RABIT, a Robot Arm Bug Intervention Tool for Self-Driving Labs

Zainab Saeed Wattoo, Petal Vitis, Ruizhe Zhu, Noah Depner, Ivory Zhang,
Jason Hein, Arpan Gujarati, and Margo Seltzer
University of British Columbia, Vancouver, BC, Canada

***Abstract*—Self-driving labs are transforming scientific research and accelerating experimentation using software-controlled lab equipment. These labs are exposed to human errors by inexperienced researchers working in the lab (e.g., setting incorrect target location could cause a robot arm to collide with an expensive piece of equipment). We present RABIT, a Robot Arm Bug Intervention Tool, which (i) allows systematically specifying safety rules across diverse devices and (ii) evaluates and enforces these rules using simulation, a low-fidelity testbed, and a production environment. We report our experience adapting RABIT for the Hein Lab, a state-of-the-art research lab that blends advanced robotics with synthetic organic chemistry.**

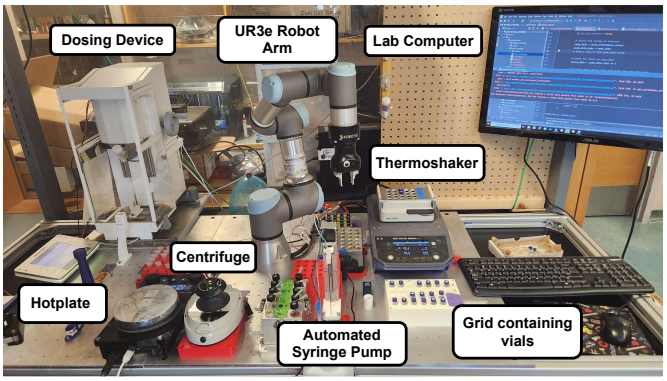
***Index Terms*—self-driving labs, rule-based anomaly detection**

I. Introduction

Manufacturing and material synthesis research involves repeated physical experiments that iterate through the parameter search space. *Self-driving labs* automate every step of this process using robot arms and software-controlled equipment, allowing researchers to *accelerate discovery*. Such labs are emerging across diverse fields, such as chemistry [18, 40], nanotechnology [39], and energy technology [20, 30]. Examples include Polybot [11] at the Argonne National Laboratory, the Matter Lab [7] at the University of Toronto, and the Hein Lab [4] at the University of British Columbia.

Fig. 1(a) illustrates a prototypical experiment deck in the Hein Lab. Each robot arm and lab device can be individually programmed using device-specific APIs. However, lab engineers typically write lightweight python wrappers over these APIs, providing an easy-to-use programming environment, as in Fig. 1(b). Hence, even students and young researchers with no experience in computer networking or hardware interfacing can quickly learn to automate their research.

While self-driving labs revolutionize research and experimentation, with increased automation there is the risk that even small, inadvertent programming errors can cause nontrivial damage to humans or expensive equipment in the lab.[1] Since self-driving labs are perennially in prototyping mode, such errors are more likely than in large-scale industrial manufacturing plants. *Our goal is to defend self-driving research labs*



(a) An experiment deck in the Hein Lab

```python
dosing_device = DosingDevice()
syringe_pump = SyringePump()
hotplate = Hotplate()

dosing_device.doseSolid(amount)
syringe_pump.doseInitialSolvent(volume)
hotplate.stirSolution(temperature)
image = recordImage()
measureSolubility(image)

while (not SolutionDissolved)
  syringe_pump.doseSolvent(amount)
  hotplate.stirSolution(temperature)
  image = recordImage()
  measureSolubility(image)
end while
```

Omitting either Line 13 or 15 can cause a collision

```python
class DosingDevice:
  def doseSolid(amount):
    open_door()
    robot.move_to_location(grid_location)
    robot.pick_up_vial()
    robot.move_to_location(dosing_device_location)
    robot.drop_vial()
    robot.move_to_location(home_location)
    close_door()
    if amount < 10: start_dosing(amount)
    else: raise Exception("Amount exceeds vial capacity")
    # Dosing stops when amount is dispensed
    open_door()
    robot.move_to_location(dosing_device_location)
    robot.pick_up_vial()
    robot.move_to_location(home_location)
    close_door()
```

(b) Python experiment script for automated solubility measurement

Fig. 1. An experiment deck and its corresponding programming environment at Hein Lab [4]. The object `dosing_device` in the main script points to the Python class `Dosing_Device`. The Python class exports convenient APIs for many common operations involving the dosing device and the robot arm (shown in the top figure) – such as `doseSolid(amount)` – while hiding low-level device communication details from the users.

against misconfigurations and programming errors that can lead to unsafe behaviors, without affecting programmer productivity and without slowing down prototyping capabilities.

Currently, the effect of potentially unsafe commands is mitigated to some extent by **(i)** device-specific thresholds embedded inside device firmware, e.g., the hotplate in Fig. 1(a) allows setting a safe temperature limit [6], and **(ii)** checks added by programmers to their experiment scripts, e.g., Lines 10-11 in Fig. 1(b) ensure that the dosing amount does not exceed 10mg. Device-specific thresholds cannot prevent unsafe behaviors that result from interaction between two devices, e.g., a robot arm may not accept coordinates that will force its arm to hit the ground, but its firmware may not prevent

[1]For example, the dosing device shown in Fig. 1(a) has a software-controlled glass door; there have been instances of the door breaking because the programmer forgot to call `open_door()`, i.e., Line 13 in the `doseSolid(amount)` definition in Fig. 1(b) was omitted. Similarly, if, say, Line 15 in the `doseSolid(amount)` definition is omitted inadvertently, the robot arm does not collect the vial back from the dosing device, which then collides with the new vial in the subsequent iteration.

TABLE I
COMPARING THE CAPABILITIES OF RABIT'S THREE STAGES

Capabilities	Simulator	Testbed	Production
Speed of exploration / testing	High	Medium	Low
Device precision and quality	Low	Medium	High
Accuracy of results	Low	Medium	High
Risk of damage	Low	Medium	High

a collision with another robot arm. In contrast, checks inside programming scripts can be more comprehensive, as programmers derive these from their holistic knowledge. Unfortunately, such checks are added ad hoc, scattered all over the program, making this approach both error-prone and cumbersome.

We present RABIT, a Robot Arm Bug Intervention Tool for systematically specifying rules, testing experiment scripts, and enforcing safe execution. We summarize RABIT's key ideas and contributions below. Although we design RABIT in consultation with the Hein Lab, the ideas apply to self-driving labs, particularly those containing diverse equipment.

Gathering rules. Identifying an exhaustive set of rules that define unsafe behavior over all possible executions across all devices is nontrivial. To address this challenge, we formulated rules using information gathered from three sources: an existing robotic arm dataset (RAD) [12], the Hein Lab experiment scripts, and researchers in the Hein Lab. Together, these allow us to identify both general-purpose rules that apply to a broad class of self-driving labs and custom rules that are specific to an individual self-driving lab, such as the Hein Lab.

Enforcing rules. As our goal is to avoid unsafe behavior, we need to actually attempt to execute such unsafe behaviors to determine if our system detects them. However, a bug in the detection could have devastating consequences. To avoid this, we use a three-stage framework for detecting rule violations: **(i)** simulation, for quick testing of individual robot arm movements; **(ii)** a low-fidelity, inexpensive testbed for testing physical actions without damaging side effects; and lastly, **(iii)** testing in the production environment. Table I provides a quick summary of their capabilities.

Contributions. Our main contribution is demonstrating how the aforementioned key ideas can be realized in a production and fully operational state-of-the-art self-driving lab. Specifically, we present a report summarizing our experience adopting RABIT in the Hein Lab (see Sections II to IV). We also discuss RABIT's usability and broader applicability to other self-driving labs (see Section V).

II. RABIT

We consider the Hein Lab's experiment deck shown in Fig. 1(a) as our production environment. It consists of a lab computer, a six-axis robot arm [16], and five automation devices: a solid dosing device [8], an automated syringe pump [13], a centrifuge [3], a thermoshaker [5], and a hotplate [5].

A. Construction of the Rulebase

We first examined the Robot Arm Dataset (RAD), which includes three months of command trace data captured in the Hein Lab [12]. We mined the dataset to identify rules implied by the sequences of commands. We identified rules that ought to apply to all self-driving labs, e.g., device doors must be opened before a robot arm can enter them, as well as rules that seemed unique to the lab from which the data were collected, e.g., solids must be added to containers before liquids.

Next, we consulted with the Hein Lab researchers, who emphasized the need to avoid collisions of robot arms with nearby equipment or people, and that the temperature of the hotplate must never exceed the specified threshold. When their safety criteria conflicted with RAD-inferred rules, we used the rules suggested by our collaborators.

Finally, we examined their experiment scripts, looking for explicit checks. We retained the distinction observed in RAD that rules fall into two categories: general-purpose rules that apply to most labs and custom rules specific to a particular lab. This design makes it easier to adapt to a new environment by describing only the items specific to that environment.

Based on the three sources discussed above, we classified each device (robot arms or software-controlled devices) into one of four types. **(1) Container:** any object that can contain a substance (solid, liquid etc.) and typically has a stopper through which the substance goes in or out. **(2) Robot Arm:** a system that moves from one location to another and has the ability to pick up, move, and place objects. **(3) Dosing System:** any system used for adding substances into a container during the experiment. **(4) Action Device:** any system with 'active/inactive' states, where the active state refers to the system performing an action, such as heating, stirring, or shaking. Both dosing systems and action devices might have doors preventing an object from entering or exiting.

For each device type, we identify *state variables* that fully describe the device, e.g., *deviceDoorStatus* indicates if a device's door is open or closed and *robotArmHolding* indicates if the robot arm gripper is holding an object. We also identify, for each device type, *actions*, which can modify the associated state variables. Each action has a set of *preconditions*, which must hold for the action to be allowed, and *postconditions*, which must hold after the action completes (e.g., see Table II). The complete set of all such descriptions constitutes the RABIT rulebase. Tables III and IV show all the general-purpose and custom, lab-specific rules (respectively).

B. Detecting Rule Violations

The algorithm in Fig. 2 describes RABIT's execution. In short, RABIT intercepts each action; if the preconditions are not met, it stops the experiment (to prevent execution of unsafe actions) and alerts the user; and after each action, checks for device malfunctions. RABIT stops an experiment preemptively based on the Hein Lab's recommendation. However, this can be dangerous at times, e.g., if a robot arm is left holding a volatile substance, a person can bump into it. In such cases, a fail-safe scenario may be recommended instead.

Lines 1-3. When RABIT starts, it acquires the initial state of all devices, $S_{initial}$, using a set of status commands. It then sets $S_{current}$, which denotes the current state, to $S_{initial}$.

TABLE II
EXAMPLE ACTIONS, PRECONDITIONS, AND POSTCONDITIONS ASSOCIATED WITH A ROBOT ARM DEVICE TYPE

Example actions associated with a robot arm device	Preconditions	Action labels	Postconditions
Moving a robot arm inside a specific device	deviceDoorStatus[device] = 1	move_robot_inside	robotArmInside[robot][device] = 1
Using a robot arm to pick up an object (a vial in this case)	robotArmHolding[robot] = 0	pick_object	robotArmHolding[robot] = 1
Using a robot arm to place an object (a vial in this case)	robotArmHolding[robot] = 1	place_object	robotArmHolding[robot] = 0

TABLE III
GENERAL RULES FOR SELF-DRIVING LABS

No.	General rules
1	Robot arm cannot move into a device whose door is closed
2	Device door cannot be closed when the robot is inside the device
3	Robot arm can move to any location not occupied by any object
4	Robot arm can pick up an object when it isn't holding something
5	Action device can perform actions when a container is inside it
6	Action device can perform actions when a container is not empty
7	A substance can be transferred from a delivering container to a receiving container when neither has a stopper on it
8	A substance can be transferred from a filled delivering container to an empty or partially filled receiving container
9	Dosing systems or action devices with doors should start dosing or performing an action, respectively, only when their doors are closed
10	The door of the dosing systems or action devices with doors should be closed when they are running
11	The action value, such as temperature or stirring speed, for a given action device should not exceed its predefined threshold

TABLE IV
CUSTOMIZED RULES FOR THE HEIN LAB

No.	Customized rules
1	Add liquid to a container only if the container already has solid
2	Place the container in the centrifuge only if the container contains both a solid and a liquid
3	Place the container in the centrifuge only if the red dot on centrifuge faces North
4	Place the container in the centrifuge only if the container has a stopper on it

Lines 5-7. The system transitions from one state to another via a single command, denoted a_{next}, which is responsible for executing an action. Each action has a precondition that must hold before the command can be executed (e.g., see Table II). If $S_{current}$ does not satisfy a_{next}'s precondition, RABIT considers the state-action pair invalid and raises an alert.

Lines 8-10. If a_{next} is a *move* command for a robot arm, RABIT checks if the robot arm can move without colliding with other devices or bumping into walls or the ground. We designed an Extended Simulator (see Section III) that models other automation devices as 3D stationary objects and checks if the robot arm's trajectory causes a collision. In the absence of such a simulator, only the target location is checked for potential collisions; the precondition for every move command requires that there be no object in the target location.

Lines 11-15. RABIT computes the expected state, $S_{expected}$, using the current state, $S_{current}$, and the action's postconditions. It then executes a_{next} and afterwards acquires the actual state of all devices, i.e., S_{actual} (using status commands, like at the initialization time). If $S_{actual} \neq S_{expected}$, RABIT assumes that at least one device malfunctioned and raises an alert.

```
1:  Input: Initial state of the self-driving lab S_initial
2:  Output: Alert, if a safety violation is detected
3:  S_current ← SetState(S_initial)
4:  while ¬SystemReboot do
5:      Fetch the next command a_next
6:      if ¬Valid(S_current, a_next) then
7:          alertAndStop("Invalid Command!")
8:      if isRobotCommand(a_next) then
9:          if SimAvailable and ¬ValidTrajectory(a_next)
        then
10:             alertAndStop("Invalid trajectory!")
11:     S_expected ← UpdateState(S_current, a_next)
12:     Execute command a_next
13:     S_actual ← FetchState()
14:     if S_actual ≠ S_expected then
15:         alertAndStop("Device malfunction!")
16:     S_current ← SetState(S_actual)
```

Fig. 2. RABIT's Execution Algorithm

C. Implementation

The lab researcher configures RABIT for their lab by instantiating their devices in the JSON files that we provide. They must categorize each device into its device type and enter its properties, including the class name that provides the device's APIs and additional properties (such as the presence and position of a door). Additionally, for each device, they should add the commands responsible for executing actions and for retrieving the device's state. They can also define lab-specific rules that become part of the custom rulebase and new device categories, if they have devices that do not belong to any of the four specified device types.

The JSON format provides a simple and standardized way to represent information, making it easy for researchers to modify and update the device information. We use the information from the JSON files to populate a state transition table, which is a two-dimensional labeled data structure similar to Table II.

We use an open-source tracing framework RATracer [22], which instruments the Python experiment scripts to intercept and trace all device commands at run time. We reconfigure RATracer such that every time it traces a command, it first checks with RABIT if the command is safe to run: if RABIT raises an alert, the experiment is halted (RATracer raises a

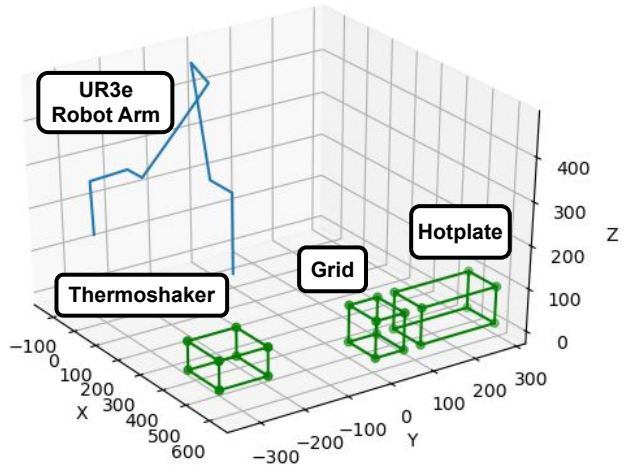


Fig. 3. Extended Simulator

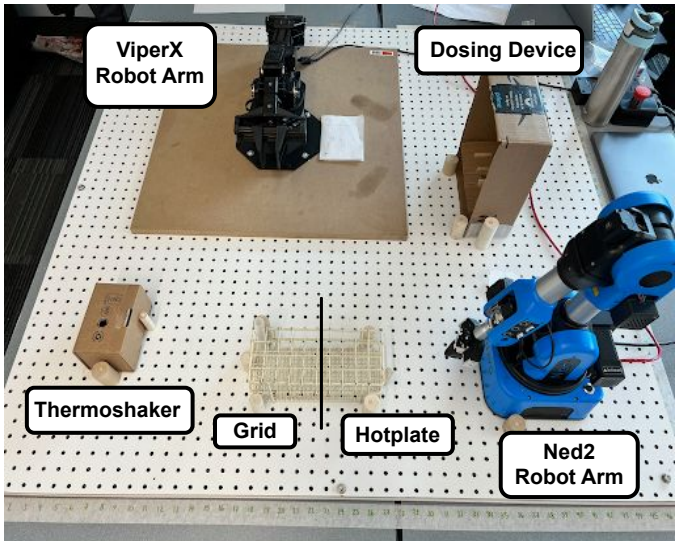


Fig. 4. Testbed

Python exception in this case); otherwise, the command is forwarded to the device and executed.

RABIT also maintains a list of device connection parameters, which are extracted from the programming scripts. It uses these parameters to fetch the state of all devices, i.e., as part of the *FetchState()* function on Line 13 in the algorithm.

We evaluated the latency overhead due to RABIT. Without the Extended Simulator, RABIT incurs approximately 0.03s overhead (1.5%), which is generally imperceptible to humans [26]. However, with the Extended Simulator, RABIT incurs approximately 2s overhead (112%). The simulator overhead arises mainly from its Graphical User Interface (GUI), which runs in a virtual machine and is invoked each time RABIT checks for collisions. The overhead is acceptable during testing, but for deployment, we plan to bypass the GUI entirely when interacting with the simulator.

III. Testing Platforms

Extended simulator. The Hein Lab uses the six-axis UR3e robot arm [16], which comes with an accurate simulator URSim [1]. However, URSim does not model other automation devices. It also does not account for collisions when the robot arm moves through its mounting platform or hits the walls.

We augmented URSim to develop an *Extended Simulator*. In the augmented version, we model each device on the experiment deck as a 3D cuboid object (as shown in Fig. 3). Further, by continuously polling the robot arm's trajectory and comparing it with the 3D objects' coordinates, the Extended Simulator can detect if the robot arm is likely to collide with one of the automation devices and alert the user.

Testbed. The testbed emulates the Hein Lab using lower precision robot arms and low-fidelity device mockups. It provides an environment for executing potentially unsafe programs, so that the chances of these programs causing a damage when deployed in a production environment are significantly reduced. The testbed also lets us experiment with intentionally unsafe workflows to check if RABIT detects them.

Our testbed setup (Fig. 4) consists of a lab computer that controls five low-fidelity objects and two robot arms: a six-axis ViperX [17] and a six-axis Ned2 [9]. Both robot arms are designed for educational and research purposes. They have the same degree of freedom as the UR3e, but limited capabilities and precision. The low-fidelity objects resemble the shapes and functionalities of their counterparts in the Hein Lab and are realized using cardboard mockups or toy devices.

The testbed allows us to test scenarios for which the simulator is insufficient, such as using multiple robot arms or checking for collisions that occur if a robot arm is holding a vial and the vial (not the arm itself) collides with a device.

IV. Evaluation

We assess whether the rules are accurately programmed in RABIT by checking if RABIT successfully detects every rule violation. We did not disable any existing safety mechanisms built into the devices or any ad-hoc safety checks added in the experiment scripts by the lab researchers; these worked in tandem with RABIT. For evaluation, we ensure that there are no intentional bugs in the JSON configurations. We first conducted controlled experiments on both the Extended Simulator and the testbed. We deliberately executed unsafe scenarios designed to trigger each rule in the rulebase. For example, in the simulator, we attempted to move UR3e inside the grid, violating rule 3 in Table III. On the testbed, we attempted to move ViperX inside the dosing device while its door was closed, violating rule 1 in Table III. **RABIT successfully detected unsafe behavior in all these scenarios.**

Next, we conducted uncontrolled experiments, where we asked one of our collaborators to modify the experiment scripts (examples are shown in Fig. 5 and Fig. 6) and introduce bugs in them, as if they were a naive programmer. They were given access only to the programming scripts on the lab computer but were not allowed to manually reposition robot arms or move equipment. Hence, they could easily change the arguments of

```
1  from workflow_utils import locations
2
3  if __name__ == '__main__':
4
5      # Set vial locations
6      ned2_grid = locations["grid"]["NW"]["ned2"]
7      viperx_grid = locations["grid"]["NW"]["viperx"]
8      viperx_dosing_device = locations["dosing_device"]["viperx"]
9
10     # Start workflow
11     dosing_device.set_door("state", "open")
12     vial.decap_vial()
13
14     viperx.arm.go_to_home_pose()
15     viperx_pick_up_object(viperx, viperx_grid, vial)   Bug C
16     viperx_place_object(viperx, viperx_dosing_device, vial)
17
18     viperx.arm.go_to_home_pose()
19
20     dosing_device.set_door("state", "closed")
21     dosing_device.run_action(delay=3, quantity=5)
22     dosing_device.stop_action(delay=0)
23     dosing_device.set_door("state", "open")   Bug A
24
25     viperx_pick_up_object(viperx, viperx_dosing_device, vial)
26     viperx_place_object(viperx, viperx_grid, vial)
27
28     random_location = [0.443, -0.010 , 0.292]   Bug B
29     ned2.move_pose(random_location)
30
31     dosing_device.set_door("state","closed")
32     viperx.arm.go_to_home_pose()
33     viperx.arm.go_to_sleep_pose()
34
35     ned2_pick_up_object(ned2, ned2_grid, vial)
36     ...
```

Fig. 5. The example (without the annotations) illustrates a safe tesbed workflow based on the automated solutbility experiment shown in Fig. 1(b). The annotated bugs represent the different categories of unsafe behavior introduced by adding, deleting, or updating one or two lines in the code. Bug A is introduced by omitting Line 23; hence, when ViperX goes to the dosing device to pick up the vial (Line 25), it collides with the door. Bug B introduces Lines 28 and 29, which asks Ned2 to move to a location close to the grid, while ViperX is also stationed there (after Line 26), causing the two robot arms to collide. Bug C is introduced by omitting Line 15; ViperX in this case continues the remaining experiment without a vial.

```
1  locations = {
2      "grid": {
3          "NW": {
4              "meta": "original vial location",
5              "viperx": {
6                  "pickup_safe_height": [0.537, 0.018,  0.23],
7                  "pickup": [0.537, 0.018, 0.12]
8              },
9              "ned2": {...},
13         },
14         "SE": {
15             "meta": "imaginary hotplate for now",
16             "ned2": {...}
20         }
21     },
22     "dosing_device": {
23         "viperx": {
24             "approach": [0.15, 0.347, 0.19],
25             "pickup_safe_height": [0.15, 0.475, 0.19],
26             "pickup": [0.15, 0.45, 0.10 0.08]   Bug D
27         },
28         "ned2": {...}
33     }
34 }
```

Fig. 6. The above snippet, which is part of a utilities file, contains hard-coded device-specific location coordinates. Bug D is introduced by changing the z-axis coordinate of the pickup location (Line 26) from 0.10 to 0.08. These coordinates are used by `viperx_place_object(...)` in Fig. 5 (see Lines 8 and 16). Since the new z-axis coordinate is close to 0, when ViperX is holding a vial, the vial crashes to the ground and breaks.

commands (e.g., enter incorrect coordinates for robot arms), delete commands (e.g., remove a command to close the door of a device), or change the order of commands (such as altering the sequence of locations to which a robot arm was supposed to move). These experiments (see [15] for details) resulted in four categories of unsafe behavior.

1. Interactions with the dosing device door. This category deals with improper handling of device doors, e.g., attempting to close the door while ViperX is still inside the device or attempting to move ViperX inside the dosing device while the door is closed (Bug A in Fig. 5 illustrates the latter scenario). *RABIT raised an alert in all such scenarios.*

2. Collisions between two robot arms. Consider Bug B in Fig. 5. ViperX is stationed just above the grid after placing a vial. The programmer moves Ned2 to a `random_location` close to the grid but different from ViperX's position. This resulted in a collision. *RABIT did not raise an alarm.*

To detect collision between two robot arms, RABIT requires a common frame of reference. Since Ned2 and ViperX are sourced from different vendors, and have varying gripper sizes and low precision, this is challenging. For example, transforming both robot arms' coordinate systems to a global coordinate system using a transformation matrix [14] resulted in an average error of 3cm between the expected and computed positions. Hence, we continue using separate coordinate systems (the *de facto* approach in the Hein Lab) but adopt a workaround for *preventing* such collisions in the first place: *we multiplex robot arm movements in either time or space.*

To multiplex in time, we ensure that, at any given time, only one robot is in motion whereas other robot arms are in their sleep position and modeled as 3D cuboid spaces (identically to other devices). For our testbed, we specify Ned2's shape and sleep position in ViperX's environment (and vice versa) and modify RABIT to add preconditions to enforce this behavior. For space multiplexing, we add a software-defined wall between the two robot arms in their environments, providing each robot with its own dedicated space in which it can move, while allowing to let them move concurrently.

Our workaround mimics common safety practices in self-driving labs. However, RABIT provides lab researchers the ability to formally express these safety practices and enforce them at runtime, while pushing for more concurrency in their experiments. External monitoring tools such as 3D cameras can also help with safety monitoring in such scenarios.

3. Experiments without a vial. Consider Bug C in Fig. 5. ViperX does not pick up the vial because the call to function `viperx_pick_up_object()` is omitted. ViperX may also not pick up the vial if there is a bug in the function definition, e.g., if commands `open_gripper()` and `close_gripper` are reordered. In both cases, *RABIT did not raise an alarm*, and the remaining experiment continued without a vial. Currently, we cannot detect if the robot arm gripper is holding an object, since we do not have a gripper pressure sensor or cameras for external monitoring.

4. Changing position coordinates. The robot arms were asked to move to seemingly infeasible locations, e.g., by replacing one or more device coordinates with a very high or a very low value. *This resulted in varied behavior.*

Bug D in Fig. 6 demonstrates one such scenario where ViperX's arm is made to collide with the platform. RABIT raised an alarm when ViperX was not holding any object. When ViperX's gripper was holding a vial, the vial collided with the platform before RABIT could raise an alarm. RABIT failed to account that a robot arm's dimensions may change if it is holding an object. We modified RABIT to account for these changes, which successfully detects such collisions.

When ViperX was moved to a very high, clearly infeasible, position, it failed to compute the trajectory and silently ignored the command. RABIT did not raise an alarm, even though silently skipping a command can be potentially unsafe.[2] With Ned2, this was not an issue as it throws an exception and halts immediately if it cannot compute the trajectory.

Summary. Our collaborator, the "naive" programmer, carried out 16 program changes with potentially unsafe consequences. Initially, RABIT detected 8 of them, resulting in a detection rate of 50%. After modifying RABIT, it successfully detected 12 scenarios, resulting in a detection rate of 75%. With the Extended Simulator on the side, we were able to detect one more scenario, improving RABIT's detection rate to 81%. While the evaluation results are encouraging, without more practical deployment experience in multiple self-driving labs and without exhaustive testing (which requires generating large bug datasets – a challenging task in itself), we do not know if these numbers are representative of what we might see in practice. Hence, RABIT's detection rate reported in the paper should not be mistaken for its likelihood to detect unsafe behavior in the wild. Importantly, throughout testing, RABIT never produced any false positives (i.e., false alarms). This is important for programmer productivity, i.e., RABIT does not run the risk of producing alarm fatigue, where researchers start ignoring alarms because they are raised frequently and unnecessarily. Table V further categorizes the introduced bugs based on increasing severity and the potential damage they could cause. It includes the total number of bugs in each category and the number of bugs detected by RABIT.

[2]Suppose ViperX needs to move from location A to B and then to C. The approach via B is intentionally chosen to avoid collision with a nearby object. However, if the location coordinates for B are accidentally changed to B' and ViperX cannot compute the trajectory from A to B', it skips this move, and proceeds to move directly from A to C, resulting in a collision. RABIT raised an alarm when this scenario was replayed in the Extended Simulator, as the Extended Simulator is able to detect collisions between robot arm trajectories and other devices, and signal it back to RABIT.

TABLE V
SEVERITY OF BUGS WITH THE TOTAL NUMBER OF BUGS IN EACH CATEGORY AND THE NUMBER OF BUGS DETECTED BY RABIT

Severity of Bugs	Total	Detected
Low: Wasting chemical materials *(e.g., spilling solid out of the vial)*	3	1
Medium-Low: Breakage of glassware *(e.g., robot arm dropping a test tube)*	1	1
Medium-High: Robot arm causing harm to the environment or inexpensive nearby objects i.e., platform it is mounted on, the nearby walls, or the grids that hold the vials *(e.g., robot arm making holes in a wall)*	6	4
High: Robot arm breaking the expensive equipment inside the lab *(e.g., robot arm breaking a dosing device)*	6	6

Due to lack of prior work on security and safety of self-driving labs, we do not have baselines against which to compare RABIT or any data on quantitative measures. Before adopting RABIT, we expect researchers in other self-driving labs to carry out a qualitative analysis of potentially unsafe scenarios in their lab, compare those to our test suite, and determine if RABIT is suitable for their environment.

V. DISCUSSION

We report on RABIT's usability based on our experience of deploying it in the Hein Lab. We discuss if RABIT can be generalized to other self-driving labs and, if customization is required, the effort needed and the problems encountered when customizing it for a specific self-driving lab. We also discuss the open challenges associated with deploying RABIT.

A. Pilot Study with the Hein Lab

We evaluate RABIT's usability via an informal pilot user study. Briefly, we provided a 30-minute one-on-one training session to one of the Hein Lab researchers, participant P. The training session included an overview of RABIT, the device configuration files (in JSON format) it uses, and the user study.

After the training session, we provided participant P with the configuration file templates and asked them to enter all details to describe their experimental platform. It took them approximately three hours to enter device-specific information and a custom rule. In addition, we spent around four hours debugging the entered information, before we could execute one of the experiment workflows using RABIT. For example, participant P accidentally entered a negative sign instead of a positive sign in a location. There were few JSON syntax errors as a result of which RABIT misinterpreted some device information. In hindsight, using a JSON-aware editor [10] could have helped avoid syntax errors, and more precise JSON schema specifications could have helped avoid sign errors.

After successfully setting up RABIT, participant P executed a series of unsafe scenarios in a controlled setting. Some of

these scenarios were suggested by us and some were done at participant P's discretion. For instance, to induce a collision with the grid, P reduced the height of the location at which UR3e is supposed to be when picking up the vial from the grid. In another scenario, P tried to have the dosing device add more solid than the vial could hold. *All unsafe scenarios attempted by P were detected successfully by RABIT.*

After completing the user study, we asked participant P a set of questions about their experience using RABIT, the strengths and limitations they identified in RABIT, and suggestions for improvements they would like to see. P described their overall experience as challenging yet rewarding. They found entering information in the JSON files challenging, mainly due to their unfamiliarity with certain action and status commands for specific devices. They informed that they would not have been able to set up RABIT without assistance. At the same time, they said, *"The set up did take a lot of work; now that it is up I imagine the maintenance is relatively simple"*.

Participant P found the capabilities of RABIT to be valuable, especially in terms of adding specific properties to devices. For instance, they highlighted the usefulness of being able to configure doors for certain devices or stoppers for containers, which can then be used to detect rule violations. They also highlighted RABIT's ability to identify potential collisions and prevent accidents before they occur, noting its usefulness in training new users and avoiding damage to components. Further, after sharing their experiences where they collided the robot arm into nearby equipment, P stated, *"I see a lot of benefit for using this when constructing new workflows or making changes or even adding new components."*

Participant P also mentioned certain limitations or missing features in the system that affected their experience. RABIT forced them to simplify certain aspects, for instance, making a choice between multiple commands for robot movement, selecting between two different commands used for dosing liquid with an automated syringe pump. P mentioned that the complexity of device shapes posed a challenge, as the shape of many devices do not comply with RABIT's cuboid specification. For example, a centrifuge resembles a hemisphere more than a cuboid and the thermoshaker has a bump at the top. They suggested that incorporating more detailed shape descriptions would enhance RABIT's flexibility.

Based on their overall experience, P stated that they would likely recommend RABIT to others. As per them, RABIT is currently suitable for users with relatively simple systems involving just robot arm movements between locations. However, before recommending RABIT to users with complex setups involving multiple components, they suggested making RABIT more adaptable: *"In its current state, I would recommend it for training new graduates on the systems without risking breakage."* They also said, *"I think it's a great tool for setting-up or changing workflows in its current iteration (especially in locations), but with more adaptability to complex systems it could later be used in full workflows."*

Evaluating RABIT's usability is at a preliminary stage. We have currently deployed RABIT in the Hein Lab, with whom we have a long-standing partnership and trust.

B. Generalizing RABIT to the Berlinguette Lab

We visited another self-driving lab – the Berlinguette Lab [2] in the University of British Columbia, which performs cutting-edge research in materials science and chemistry. Our goal was to evaluate the adaptability of RABIT to this lab, determining if we could categorize the devices in the lab according to the four predefined device types and whether the rules in our rulebase are generalizable to the workflows they run. This assessment was based on observations, discussions with personnel, and an examination of their lab setup.

The Berlinguette Lab also has a central workstation running Python experiment scripts to control robot arms and software-controlled devices for each automated experiment platform. The devices that are part of their research and development experiment platform included the UR3e robot arm, a dosing device with a door similar to that in the Hein Lab, and a decapper responsible for capping and uncapping vials. As per our categorization, the dosing device can be identified as a dosing system, while the decapper can be classified as an action device due to its specific capping and uncapping actions.

Another automated experimental platform in the lab was composed of multiple individual stations, each enclosed with walls, platforms, and ceilings, and served by a central six-axis UR5e robot arm. This arm is used for transferring vials and materials between different stations. Noteworthy stations included a precursor mixing station with an N9 robot arm and a spin coater (this can be categorized as an action device as its primary actions include starting and stopping spinning). Additionally, there was a spray coating station that has a hotplate (which can be categorized as an action device), an automated syringe pump for drawing solvent (which can be categorized as a dosing system), and ultrasonic nozzles (which can be categorized as action devices with spraying and not spraying being their primary actions). The lab also has an XRF microscopy device emitting x-rays and injecting photons onto film. We can categorize this device as a set of multiple action devices in our custom rulebase, or in future expand the definition of action devices to consider multiple actions.

During discussions with lab personnel, safety concerns were raised regarding both human and expensive equipment safety. For safety concerns, they used sensors earlier, but due to the possibility of frequent false alarms and malfunction, they do not use them anymore. Therefore, they emphasized the need for additional safety measures.

RABIT ensures equipment safety, such as collisions of robot arms with equipment, considering doors. However, in its current state, RABIT does not consider nearby humans. However, by incorporating sensors, which could be treated as a new device class, one could imagine enhancing RABIT to respond to sensor inputs that indicate a robot arm is approaching the area that is occupied.

In conclusion, we are able to categorize most of the devices as part of our four defined device types. We can generalize the rules defined as most workflows involve adding substances to

vials, drawing substances from vials, moving vials around, heating, and spraying. However, there are remaining challenges posed by the advanced nature of the lab, specifically limitations related to shape and the complexity of action devices performing multiple actions simultaneously.

C. Open Challenges

Despite RABIT's promising capabilities, there are open challenges to deploying it in more advanced labs. The primary areas for improvement are ease of use and generalizability.

Real-life, software-controlled devices come in different shapes and sizes, so we need to expand our device descriptions to easily handle objects other than cuboids. In a perfect world, we might photograph the device and use image analysis to derive its geometry. More fundamentally, our Extended Simulator is currently designed as an add-on to one specific robot arm simulator. We would like to examine other robot arm simulators to determine how best to revise the extensions to integrate with a wide range of simulators.

Another challenge arises when multiple robot arms move within the same physical space, which is common in a real-life setting. We initially addressed this issue by mapping to a common frame of reference. However, this approach proved to be impractical due to multiple sources of noise, including the large margin of errors caused by the lower precision of testbed robots and variations in their gripper sizes.

Devices might have multiple doors, for instance, for two robot arms to approach the device simultaneously. In its current state, RABIT does not handle this. Furthermore, there is a possibility that multiple commands could be used to execute a specific action. For instance, there might be two commands for moving a robot from one location to another. RABIT currently allows only one command per action.

In conclusion, despite the valuable contributions of RABIT, open challenges remain. Addressing these challenges is crucial for realizing the full potential of this technology.

VI. Related Work

The existing literature in the domain of self-driving labs focuses on making self-driving labs more automated and autonomous. There is no prior work on security and safety of self-driving labs. Hence, we discuss prior work on rule-based/specification-based intrusion detection systems (IDS) and safety monitoring in cyber-physical systems (CPS) environments that is closest to our work.

Rule-based/Specification-based IDS for CPS have been applied in various domains, including medical systems, smart grids, industrial control systems, unmanned aircraft systems [32], and network protocols [31, 41]. Specialized IDSs [19, 33–35, 37] rely on either reference models, which describe the expected behavior and properties of a system as defined by domain experts, or observed behavior, which involves capturing and analyzing real-world monitoring data to define correct system behavior. RABIT's rulebase uses both expert knowledge of self-driving lab researchers (as its reference model) and the robot arm dataset (as the observed behavior).

Mitchell and Chen [33] propose a rule-based IDS for medical CPS, converting behavior rules to state machines. They further extend their IDS to smart grids [35] and unmanned air vehicles [34] with domain-specific rules. Unlike RABIT, they do not monitor commands before execution, but evaluate only post-conditions using sensor/actuator readings. Our work is also different in that it formalizes rules for interactions between heterogeneous devices in self-driving labs.

Mitchell and Chen [33–36] also use Monte Carlo simulation for evaluating their IDS. Similarly, Pan et al. [37] test their IDS using a testbed that emulates an electrical transmission system. Testbeds and simulators are common when validating an IDS [23, 25, 27, 28], as they provide flexibility in exploring various parameters and conditions that cannot be tested in real-world environment. However, simulations and testbeds can be rather simplistic and may not fully account for the complexities of real-world scenarios. Real-world validation is hence necessary. We validate RABIT using simulation and testbed, as well as in a production environment at the Hein Lab, which subjects it to a multitude of realistic conditions.

Safety Monitoring in CPS has been explored for autonomous systems. SOTER [21] is a robotics programming framework that uses sensor data to monitor the robot and its environment at runtime, and switches the robot to a safe operating mode if a safety violation is detected. SOTER and other similar frameworks [24, 29, 38] are programmed to consider a single device, e.g., a single robot or an autonomous system. They do not consider rules involving interactions between multiple devices, such as the rule *'Action device can perform actions when a container is inside it'*.

VII. Conclusion

We presented RABIT, a tool for detecting and preventing unsafe behaviors in self-driving labs. We identified eleven general-purpose rules that can be augmented with a few custom rules to detect unsafe behavior when diverse devices in self-driving lab interact in complex ways. We introduced a three-stage deployment framework that allows researchers to safely test both their experiment workflows and RABIT's ability to prevent errors before deploying experiments in production. Our work is the first to consider rules spanning multiple heterogeneous devices in a self-driving lab, to rely on monitoring command sequences instead of sensor data, and to monitor unsafe behavior in a self-driving lab.

Acknowledgements

We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC) and the UBC Science STAIR Grant. We also acknowledge the support of Hein Lab and Berliguette Lab, especially Veronica Lai, Tara Zepel, Daniel Griffin, Jonathan Reifman, Shad Grunert, Lars P.E. Yunker, Sebastian Steiner, Henry Situ, Fan Yang, and Paloma L. Prieto at the Hein Lab for their contributions to the development of the automated solubility procedure. We thank the DSN 2024 reviewers for their feedback and our shepherd, Homa Alemzadeh, for her insightful and helpful comments.

References

[1] "UR3e Simulator," https://www.universal-robots.com/download/software-e-series/simulator-non-linux/offline-simulator-e-series-ur-sim-for-non-linux-594/.

[2] "Berlinguette Lab," https://groups.chem.ubc.ca/cberling/.

[3] "Fisher Scientific," https://www.fishersci.com.

[4] "Hein Lab," http://heinlab.com/.

[5] "IKA," https://www.ika.com.

[6] "IKA Hotplate Manual," https://www.ika.com/en/Products-LabEq/Magnetic-Stirrers-pg188/IKA-Plate-25004735/Downloads-cpdl.html.

[7] "The Matter Lab," https://www.matter.toronto.edu/.

[8] "Mettler Toledo," https://www.mt.com.

[9] "Niryo Robots," https://niryo.com/collaborative-robots.

[10] "Notepad++," https://notepad-plus-plus.org/.

[11] "Polybot," https://www.anl.gov/cnm/polybot.

[12] "Robotic Arm Dataset (RAD)," https://github.com/ubc-systopia/dsn-2022-rad-artifact.

[13] "Tecan," https://www.tecan.com.

[14] "Matrix Transformations and Coordinate Systems with Python," https://sigmoidal.ai/en/matrix-transformations-and-coordinate-systems-with-python/.

[15] "Code for unsafe test workflows on the testbed," https://github.com/ubc-systopia/dsn-2024-rabit-artifact/tree/main/testbed/unsafe_test_cases.

[16] "Universal Robots," https://www.universal-robots.com/.

[17] "Trossen Robotics," https://www.trossenrobotics.com/viperx-300-robot-arm.aspx.

[18] M. Abolhasani and E. Kumacheva, "The Rise of Self-Driving Labs in Chemical and Materials Sciences," *Nature Synthesis*, pp. 1–10, 2023.

[19] A. Carcano, A. Coletta, M. Guglielmi, M. Masera, I. N. Fovino, and A. Trombetta, "A Multidimensional Critical State Analysis for Detecting Intrusions in SCADA Systems," *IEEE Transactions on Industrial Informatics*, vol. 7, no. 2, pp. 179–186, 2011.

[20] A. Dave, J. Mitchell, K. Kandasamy, H. Wang, S. Burke, B. Paria, B. Póczos, J. Whitacre, and V. Viswanathan, "Autonomous Discovery of Battery Electrolytes with Robotic Experimentation and Machine Learning," *Cell Reports Physical Science*, vol. 1, no. 12, p. 100264, 2020.

[21] A. Desai, S. Ghosh, S. A. Seshia, N. Shankar, and A. Tiwari, "SOTER: A Runtime Assurance Framework for Programming Safe Robotics Systems," in *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2019)*.

[22] A. Gujarati, Z. S. Wattoo, M. R. Aliabadi, S. Clark, X. Liu, P. Shiri, A. Trivedi, R. Zhu, J. Hein, and M. Seltzer, "Arming IDS Researchers with a Robotic Arm Dataset," in *52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2022)*.

[23] E. Hansson, J. Gronkvist, K. Persson, and D. Nordquist, "Specification-based Intrusion Detection Combined with Cryptography Methods for Mobile Ad Hoc Networks," Command and Control Systems Technical Report, Tech. Rep., 2005.

[24] C. Harper, G. Chance, A. Ghobrial, S. Alam, T. Pipe, and K. Eder, "Safety Validation of Autonomous Vehicles Using Assertion Checking," *arXiv preprint arXiv:2111.04611*, 2021.

[25] H. M. Hassan, M. Mahmoud, and S. El-Kassas, "Securing the AODV Protocol Using Specification-based Intrusion Detection," in *Proceedings of the 2nd ACM Workshop on Q2S and Security for Wireless and Mobile Networks (Q2SWinet 2006)*.

[26] I. J. Hirsh and C. E. Sherrick Jr, "Perceived Order in Different Sense Modalities." *Journal of Experimental Psychology*, vol. 62, no. 5, p. 423, 1961.

[27] H. Lin, A. Slagell, C. D. Martino, Z. Kalbarczyk, and R. K. Iyer, "Adapting Bro into SCADA: Building Specification-based Intrusion Detection System for DNP3 Protocol," in *Proceedings of the 8th Annual Cyber Security and Information Intelligence Research Workshop on (CSIIRW 2013)*.

[28] Z. K. H. Lin, A. Slagell, and R. K. Iyer, "Using a Specification-based Intrusion Detection System to Extend the DNP3 Protocol with Security Functionalities," *Coordinated Science Laboratory Report no. UILU-ENG-12-2207*, 2012.

[29] M. Machin, J. Guiochet, H. Waeselynck, J.-P. Blanquart, M. Roy, and L. Masson, "SMOF: A Safety Monitoring Framework for Autonomous Systems," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 48, no. 5, pp. 702–715, 2016.

[30] B. P. MacLeod, F. G. Parlane, T. D. Morrissey, F. Häse, L. M. Roch, K. E. Dettelbach, R. Moreira, L. P. Yunker, M. B. Rooney, J. R. Deeth *et al.*, "Self-driving Laboratory for Accelerated Discovery of Thin-film Materials," *Science Advances*, vol. 6, no. 20, p. eaaz8867, 2020.

[31] C. McParland, S. Peisert, and A. Scaglione, "Monitoring Security of Networked Control Systems: It's the Physics," *IEEE Security & Privacy*, vol. 12, no. 6, pp. 32–39, 11 2014.

[32] R. Mitchell and I.-R. Chen, "A Survey of Intrusion Detection Techniques for Cyber-physical Systems," *ACM Computing Surveys (CSUR)*, vol. 46, no. 4, pp. 1–29, 2014.

[33] R. Mitchell and R. Chen, "Behavior Rule Based Intrusion Detection for Supporting Secure Medical Cyber Physical Systems," in *21st International Conference on Computer Communications and Networks (ICCCN 2012)*.

[34] ——, "Adaptive Intrusion Detection of Malicious Unmanned Air Vehicles Using Behavior Rule Specifications," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 44, no. 5, pp. 593–604, 2013.

[35] ——, "Behavior-rule Based Intrusion Detection Systems for Safety Critical Smart Grid Applications," *IEEE Transactions on Smart Grid*, vol. 4, no. 3, pp. 1254–1263, 2013.

[36] ——, "Behavior Rule Specification-based Intrusion Detection for Safety Critical Medical Cyber Physical Systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 12, no. 1, pp. 16–30, 2014.

[37] S. Pan, T. H. Morris, and U. Adhikari, "A Specification-based Intrusion Detection Framework for Cyber-physical Environment in Electric Power System," *International Journal of Network Security*, vol. 17, no. 2, pp. 174–188, 2015.

[38] M. Rahimi and X. Xiadong, "A Framework for Software Safety Verification of Industrial Robot Operations," *Computers & Industrial Engineering*, vol. 20, no. 2, pp. 279–287, 1991.

[39] D. Salley, G. Keenan, J. Grizou, A. Sharma, S. Martín, and L. Cronin, "A Nanomaterials Discovery Robot for the Darwinian Evolution of Shape Programmable Gold Nanoparticles," *Nature Communications*, vol. 11, no. 1, pp. 1–7, 2020.

[40] M. Seifrid, R. Pollice, A. Aguilar-Granda, Z. Morgan Chan, K. Hotta, C. T. Ser, J. Vestfrid, T. C. Wu, and A. Aspuru-Guzik, "Autonomous Chemical Experiments: Challenges and Perspectives on Establishing a Self-driving Lab," *Accounts of Chemical Research*, vol. 55, no. 17, pp. 2454–2466, 2022.

[41] C. Tseng, P. Balasubramanyam, C. Ko, R. Limprasittiporn, J. Rowe, and K. N. Levitt, "A Specification-based Intrusion Detection System for AODV," in *Proceedings of the 1st ACM Workshop on Security of ad hoc and Sensor Networks (SASN 2003)*.