CoMID: Context-based Multi-invariant Detection for Monitoring Cyber-physical Software

Yi Qin, Tao Xie, Chang Xu, Angello Astorga, and Jian Lu

***Abstract*—Cyber-physical software delivers context-aware services through continually interacting with its physical environment and adapting to the changing surroundings. However, when the software's assumptions on the environment no longer hold, the interactions can introduce errors for leading to unexpected behaviors and even system failures. One promising solution to this problem is to conduct runtime monitoring of *invariants*. Violated invariants reflect latent erroneous states (i.e., *abnormal states* that could lead to failures). In turn, monitoring when program executions violate the invariants can allow the software to take alternative measures to avoid danger. In this article, we present Context-based Multi-Invariant Detection (CoMID), an approach that automatically infers invariants and detects abnormal states for cyber-physical programs. CoMID consists of two novel techniques, namely *context-based trace grouping* and *multi-invariant detection*. The former infers contexts to distinguish different effective scopes for CoMID's derived invariants, and the latter conducts ensemble evaluation of multiple invariants to detect abnormal states during runtime monitoring. We evaluate CoMID on real-world cyber-physical software. The results show that CoMID achieves a 5.7–28.2% higher true-positive rate and a 6.8–37.6% lower false-positive rate in detecting abnormal states, as compared with existing approaches. When deployed in field tests, CoMID's runtime monitoring improves the success rate of cyber-physical software in its task executions by 15.3–31.7%.**

***Index Terms*—cyber-physical software, abnormal-state detection, invariant generation**

I. Introduction

CYBER-PHYSICAL software programs (in short as *cyber-physical programs*) integrate cyber and physical space to provide context-aware adaptive functionalities. An important class of cyber-physical programs are those that *iteratively interact* with their environments. Examples of such programs are those running on robot cars [1]–[3], unmanned aerial vehicles (UAVs) [4]–[6], and humanoid robots [7]–[9]. These program continually sense environmental changes, make decisions based on their pre-programmed logic, and then take physical actions to adapt to the sensed changes. The three steps, namely, sensing, decision-making, and action-taking, form an interaction loop between a cyber-physical program and its running environment. Each pass of such an interaction loop is referred to as an *iteration*.

To improve the productivity and cope with infinite kinds of environmental dynamics, software developers often hold certain assumptions on typical scenarios, where their cyber-physical programs are supposed to run. For example, a robot controlled by a cyber-physical program walks in an indoor environment, where the floor is supposed to be firm but not slippery, and the space is supposed not to contain any fast-moving obstacle. However, it is challenging for the developers to precisely specify what can be considered as "not firm" or "slippery". In addition, when put into an open environment that is more complex than a cyber-physical program's designed scenarios, the program itself can hardly tell when its encountered scenarios have already violated these assumptions, and thus it could be subject to various runtime errors or even system failures. As such, a cyber-physical program is easily subject to runtime errors in its deployment [11]–[15], and then suffers from misbehavior or even failure (e.g., a robot falling down and damaging itself). Therefore, there is a strong need for preventing cyber-physical programs from entering such errors, which indicate the violation of their implicit assumptions on the running environments.

One promising way is to conduct runtime monitoring of pre-specified invariants, which represent the properties that have to be satisfied during executions, to check whether a cyber-physical program's execution is safe. Being *safe* indicates that the program's execution will not lead to a failure, if no intervention is taken, but just following the logics in the program. However, specifying effective invariants is challenging. For example, one may specify invariants as the negation of failure conditions, e.g., not crashing of a UAV or falling down of a humanoid robot. However, such invariants are not that useful, because when they are violated (i.e., the corresponding failure conditions are evaluated to be true), it is already too late for a concerned program not to fail. An alternative is to specify invariants for latent erroneous states (a.k.a. *abnormal states*). Then one is potentially able to predict future failures, and prevent a concerned program from taking originally-planned actions, which would otherwise have caused failures. For example, if a robot finds its program execution violating the invariants that represent safe executions, it can decide to stop further exploring the current scenario and plan another path to its destination. This resolution action can help it avoid unexpected danger in the original scenario.

There are two major ways of specifying invariants for

Y. Qin, C. Xu and J. Lu are with the State Key Laboratory for Novel Software Technology, and the Department of Computer Science and Technology, Nanjing University, Nanjing, China (email: yiqincs@nju.edu.cn, changxu@nju.edu.cn, lj@nju.edu.cn).

T. Xie is with the Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education and the Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois (email: taoxie@illinois.edu).

A. Astorga is with the Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois (email: aastorg2@illinois.edu).

Corresponding author: Chang Xu.

detecting abnormal states: using *manually specified properties* or using *automatically generated invariants*. For the former, the developers need domain knowledge to understand what can constitute abnormal states, and derive corresponding properties. This manual process is challenging, especially when a cyber-physical program and its running environment are non-trivial [4]. On the other hand, approaches of automated invariant generation [16]–[19] provide a promising alternative. Despite varying in details, these approaches follow a general process [20] as follows. When a subject program is running, these approaches collect its execution trace in terms of program states (e.g., variable values) at program locations of interest (e.g., entry and exit points of each executed method). Then from a set of such collected safe traces (i.e., those not leading to failures), the approaches derive invariants for different program locations based on predefined templates. These invariants can then be used with runtime monitoring to predict the program's future executions to be safe (i.e., *passing*, for no invariant violation) or not (i.e., *failing*, for any invariant violation). Here, passing implies that the program runs safely with its assumptions on the environment holding, and failing implies that the program could soon fail since its assumptions on the environment no longer hold now.

However, using automatically generated invariants for runtime monitoring is still challenging. One major problem is how to balance between *general* and *specific* invariants. If an invariant for a program location is too general, using it for runtime monitoring can miss the detection of abnormal states, resulting in false negatives. For example, relaxing invariants to cater for various firm floors can accidently include firm but slippery floors, breaking the robot program's assumptions on its running environment. On the other hand, if an invariant is too specific, using it for runtime monitoring can detect many "abnormal" states even in safe executions, resulting in false positives. For example, restricting invariants to specific firm floors (e.g., in brick or wood material) can cause false alarms when the robot walks on other firm but not slippery floors, where the program's execution is still safe.

Even worse, this balancing problem can be further exacerbated by two characteristics of cyber-physical programs: iterative execution and uncertain interaction.

Iterative execution. Cyber-physical programs are featured by repeated iterations of a sensing, decision-making, and action-taking loop. Then a program location for which an invariant is generated can be executed multiple times during *multiple iterations* for dealing with *different contexts* (i.e., various situations in handling environmental dynamics). During these different iterations, a program's definition of *safe* behavior with respect to each context varies across the iterations. Overlooking these contexts, generated invariants would be overgeneralized, such that the detection of abnormal states can be missed. On the other hand, generating invariants by sticking to any specific context would also make the invariants overly fragile to other contexts of safe executions, causing false alarms.

Recently, researchers have proposed to enhance invariant generation with contexts to avoid false alarms. For example, ZoomIn [24] proposed to use program contexts to distinguish effective scopes for different invariants. However, for a cyber-physical program that iteratively interacts with its environment, only one type of context (i.e., program context) may not be sufficient for specifying an invariant's effective scope. The reason is that a cyber-physical program's behavior can also be additionally affected by its environment, even if its program context keeps similar in different iterations.

Uncertain interaction. Cyber-physical programs could also face massive false alarms due to *uncertainty* [21], when they use automatically generated invariants to detect abnormal states. For example, even if one places a robot at the same position across different iterations, its sensors can possibly report different values for its position due to uncertainty (as an inherent nature of sensing). These different input values are then propagated to a program location of interest for deriving invariants, causing this location to own variable values different from those in other safe executions also from the same position. Then, overlooking the impact of such uncertainty, runtime monitoring with the generated invariants can easily report false alarms: invariant violation is actually caused by inaccurate sensing, not due to a program's assumptions not holding on its environment.

To address these challenges, in this article, we present an approach, named Context-based Multi-Invariant Detection (*CoMID*), to automatically generating invariants for specifying developers' implicit assumptions, and checking these invariants for detecting when a cyber-physical program has entered any abnormal state at its runtime. CoMID addresses the preceding challenges with its two techniques, namely, *context-based trace grouping* and *multi-invariant detection*:

Context-based trace grouping. The first technique divides collected execution traces into different iterations, and groups them according to both program and environmental contexts. Here, *program context* refers to a program's statements executed during one iteration, and *environmental context* refers to the values of environmental attributes as sensed by the program during the iteration. The technique conducts execution trace grouping by clustering, based on the similarities of corresponding contexts between each pair of iterations. Then, for each group the technique generates invariants based only on the iterations in that group. Since the iterations in a group share a common program context and environmental context, the two contexts together specify the effective scope for the invariants generated for this group. We name this scope the group's generated *invariants' context*. Then, in the future when the cyber-physical program executes in an open environment, where different scenarios can be encountered, CoMID helps identify those iterations sharing similar contexts with the invariants that are valid to detect abnormal states. Therefore, CoMID's context-based trace grouping increases both the chance of identifying such context-sharing iterations (by shorter executions) and the accuracy of abnormal-state detection (by checking both context types)

Multi-invariant detection. The second technique addresses the robustness problem for invariants when their relied execution traces contain noisy values due to uncertainty. Instead of generating a single invariant from *all* execution traces in a group, this technique generates multiple ones,

based on *different subsets* sampled from the execution traces in the group. Then it uses an estimation function to decide the detection of abnormal states based on multi-invariant evaluation results. The function measures the ratio of violated invariants against all invariants with respect to their corresponding groups, and then takes the uncertainty in program-environment interactions into consideration, to decide whether the invariant violation indicates the detection of abnormal states or is simply caused by uncertainty. This idea has been inspired by ensemble learning [22], which uses multiple models to improve the prediction performance, in contrast to the conventional prediction based on one constituent model alone.

We evaluate our CoMID approach on three real-world cyber-physical programs: a 4-rotor unmanned aerial vehicle (4-UAV) [23], a 6-rotor unmanned aerial vehicle (6-UAV), and a NAO humanoid robot [7]. We compare CoMID with two existing approaches: *naïve*, which simply uses an invariant inference engine (i.e., Daikon [16]) to generate invariants, and *p-context*, which uses program context to enhance invariant generation and abnormal-state detection (e.g., ZoomIn [24]). The evaluation results show CoMID's effectiveness: it achieves a 5.7–28.2% higher true-positive rate and a 6.8–37.6% lower false-positive rate in detecting abnormal states for the three programs' executions; when deployed for runtime monitoring to prevent unexpected failures, CoMID improves the success rate of the three programs by 15.3–31.7% in their task executions.

In summary, this article makes the following contributions:

- The CoMID approach to automatically generating invariants and detecting abnormal states for cyber-physical programs' executions.
- The context-based trace grouping technique to refine invariant generation with respect to different contexts.
- The multi-invariant detection technique to address the impact of uncertainty in program-environment interactions on invariant-based runtime monitoring.
- An evaluation with real-world cyber-physical programs and comparison of CoMID with state-of-the-art invariant generation approaches.

The remainder of this article is organized as follows. Section II presents a program-environment interaction model for understanding a cyber-physical program's iterative execution nature, and a motivating example for explaining the challenges in generating effective invariants. Section III gives an overview of our CoMID approach and then elaborates on its two techniques. Section IV presents our evaluation of CoMID with three real-world cyber-physical programs and compares it with existing approaches. Section V discusses related work, and finally Section VI concludes this article and discusses future work.

II. Preliminaries

In this section, we introduce our program-environment interaction model and present our motivating example based on this model.

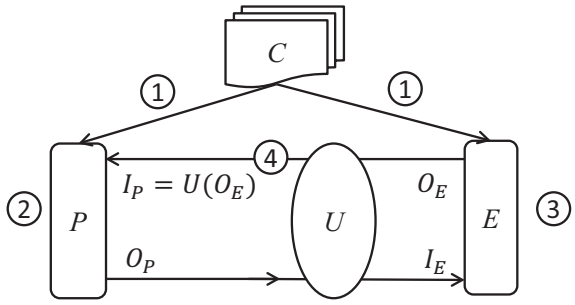


Fig. 1. PEIM's iterative reaction loop

A. Program-Environment Interaction Model

To better demonstrate the iterative execution of a cyber-physical program, we propose a *Program-Environment Interaction Model (PEIM)*. The model concerns not only the program itself, but also its environment under interaction, in contrast to traditional program models that concern programs themselves only. Note that our PEIM model is only for capturing the iterative nature of a cyber-physical program in interactions with its environment. Our CoMID approach is essentially a code-based approach.

Given a program P, we define its PEIM using a tuple, (P, E, U, C). We use P to represent the program, and E to represent the environment where the program executes. Conceptually, we consider environment E as a black-box program whose behavior can be observed by monitoring its global variables, although one may not actually know how E works. We assume that one can observe P's behavior in E (i.e., P's output) and P's obtained sensory data from E (i.e., P's input). We use C to represent P's and E's initial configuration (i.e., default startup parameter values for P and initial environmental layout for E). We use U to represent the specification of uncertainty affecting the interaction between P and E.

We define the uncertainty specification U as a function that maps environment E's output O_E to program P's input I_P. If one does not consider uncertainty, I_P would trivially equal to O_E, on their values. However, in practice, $I_P \neq O_E$ due to uncertainty. Their differences are caused by inaccurate environmental sensing (e.g., a sensed value deviates from its supposed value) or flawed physical actions (e.g., an action is taken without exactly achieving its supposed effect) [35]. Note that a complete specification of such differences may not be available. Therefore, we assume that U is a partial specification, which contains information on ranges and distributions of uncertainty on the conversion between I_P and O_E values.

As a whole, our PEIM = (P, E, U, C) works in an iterative way, as illustrated in Fig. 1. It starts with program P and environment E initialized by configuration C (Step 1). Then both P and E begin their independent executions. At the program side, P gets its input I_P from the environment's current output O_E, executes based on I_P, updates its global variables G_P, and finally returns output O_P (Step 2). At the environment side, E also takes its input I_E from the program's

(a) Walking on a wood floor (b) Walking on a brick floor

Fig. 2. A NAO robot controlled by a cyber-physical program

current output O_P, "executes" by applying I_E's effect to update its global variables G_E, and finally returns output O_E (Step 3). Once O_P or O_E is produced, E or P receives it, converts it to I_E or I_P, and puts the result in a buffer for later use. When P or E finishes its iteration, it obtains its next input I_P or I_E from the corresponding buffer using some policy, e.g., FIFO or priority-first (an input for indicating that an emergency situation can be processed first). We conceptually represent the impact of uncertainty on the conversion between P and E by $I_P = U(O_E)$ (Step 4). Steps 2 to 4 form an iterative reaction loop (i.e., *iteration*, as mentioned earlier).

B. Motivating Example

We use a motivating example to illustrate our target problem and its challenges. Consider our aforementioned NAO humanoid robot controlled by a cyber-physical program P. Its environment E, according to our PEIM, describes the robot's surrounding environment. E takes the robot's actions as input, changes its states (e.g., the position and posture of the robot), and produces P's sensory data as output. For uncertainty U, we consider only inaccurate sensing, which maps a given environment's output parameter o_E to an error range $[o_E - lower, o_E + upper]$ for P to sense. At last, the configuration C specifies the initial states of P (e.g., the initial values of P's global variables) and E (e.g., the initial position of the robot, and the layout of the obstacles).

Suppose that the robot is exploring an indoor area, as illustrated in Fig. 2. For the sake of quality and productivity, the developers can hold implicit assumptions on the scenarios where the robot is supposed to walk, e.g., a room with a firm and not slippery floor. Then, the developers proceed to design corresponding exploration strategies for the robot, e.g., walking slowly and balancing by raising its arms with certain angles. These strategies are for ensuring the robot to walk safely on a floor made of several common materials, e.g., wood, as shown in Fig. 2-a, and brick, as shown in Fig. 2-b. We next analyze what challenges the runtime monitoring with invariants can encounter, in order to prevent the robot from entering abnormal states.

Program P uses readings of two pressure sensors installed on the robot's two feet to measure whether the robot has leaned toward left or right and decide whether it has to balance the robot in its walking. The measurement is conducted by calculating the difference between the two sensors' readings, pre_{left} and pre_{right}. P then decides one of the robot's arms according to which direction the robot is leaning toward, and calculates the height the decided arm should be raised to. Suppose that variable $angle$ in P controls the height value, and then it becomes a key factor that decides whether the robot can properly balance itself in walking. The developers can design various logics to calculate the $angle$ value, but they more or less depend on the material comprising the floor.

One outstanding challenge is that the developers can hardly specify proper $angle$ values. The developers typically follow a trial-and-error process to calculate plausible $angle$ values. If lucky enough, the developers can design the calculation logics that seemingly work for several types of floor material. Even so, the users of the robot may still not be able to decide whether a specific scenario is safe for the robot to walk into (i.e., whether the calculation logics still work), or when a previously safe scenario becomes no longer safe (e.g., when the scenario gradually evolves). As mentioned earlier, runtime monitoring with invariants can play an important role to address such preceding challenge. We next explain how to generate invariants for the $angle$ variable and use them to decide whether P's execution is safe for the current scenario.

Most existing approaches of invariant generation work similarly. Consider that we generate an invariant for variable $angle$ at the entry point of method `motion.angleMove (names, angle, timeLists)`, which is the key method for deciding how to raise an arm for balancing the robot. We first collect several safe execution traces (e.g., tr_1, tr_2, and tr_3) of program P, in which $angle$'s corresponding variable-value pairs are tr_1: $\{angle = 48\}$, tr_2: $\{angle = 52\}$, and tr_3: $\{angle = 55\}$. Following a predefined template (e.g., $varX \leq C$), we can derive an invariant like "$angle \leq 55$", satisfying all the three traces. This invariant suggests that proper $angle$ values at this program location should not exceed 55. Then later when P controls the robot and finds its collected $angle$ value at the same program location to be 60, the runtime monitoring could decide that P's execution is not safe. Technically, the runtime monitoring reports that the current execution enters an abnormal state, i.e., classified as *failing*.

However, as mentioned earlier, invariant generation has to balance between general and specific invariants. The preceding invariant "$angle \leq 55$" has relaxed its condition on proper values for the $angle$ variable to cater for all the three execution traces, although these values could be from different scenarios. Then using this invariant can potentially misclassify an unsafe execution with an $angle$ value of 53 for the scenario experienced in tr_1 as passing. On the other hand, if one derives the invariant from two execution traces, tr_1 and tr_2, only (e.g., "$angle \leq 52$"), but checks it against the execution of tr_3 from another scenario. Then the runtime monitoring can be too strict and would misclassify that execution as *failing*.

The nature of cyber-physical programs exacerbates the invariant-balancing problem. For example, a cyber-physical program can encounter multiple iterations, and not all iterations share the same context. Suppose that a robot is

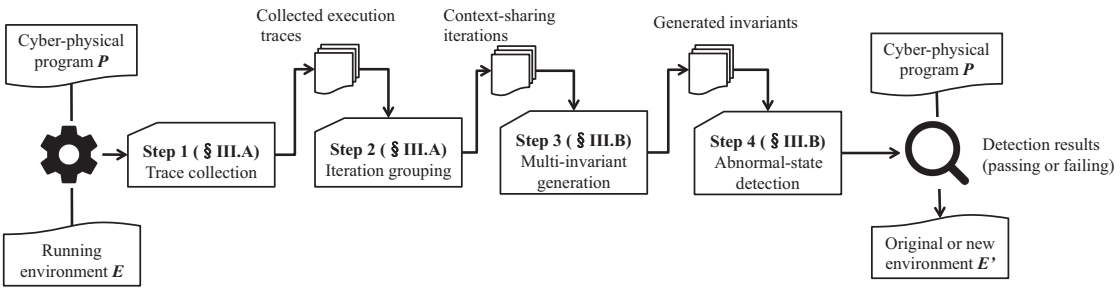


Fig. 3. CoMID's workflow

walking in a scenario connected with different types of floor material (e.g., wood and brick) and placed with different types of obstacle (e.g., high, low, and round). Such a scenario implies different values of the environment E's variables (i.e., *environmental context*). Even on the same floor, the robot may take different strategies to handle different obstacle situations. Such variety of strategies implies different execution traces of program P in the current iteration (i.e., *program context*). Without distinguishing these contexts, invariant generation can be easily over-generalized (e.g., deriving invariants to cater for all executions traces), and invariant violation can also be easily over-triggered (e.g., checking invariants in a context different from the context from which the invariants are derived).

A cyber-physical program's uncertain interactions with its environment similarly worsen the invariant-balancing problem. Uncertainty U, which might be caused by inaccurate sensing, would make derived invariants imprecise due to random noises in sensor readings. Such imprecision can cause both false-alarm and missing-warning problems. A naive way is to relax the condition in such an invariant by allowing some extent of error, e.g., a delta of ± 5 added to proper values for the $angle$ variable. However, this way is quite ad hoc, and can also easily aggravate the false-alarm and missing-warning problems.

These limitations of existing approaches on automated invariant generation motivate us to develop our CoMID approach, particularly focused on the invariant generation and runtime monitoring for cyber-physical programs. CoMID aims to distinguish different contexts for effective invariant generation and address the impact of uncertainty for effective runtime monitoring with generated invariants. We elaborate on our CoMID's methodology in the next section.

III. Context-based Multi-Invariant Detection

The input of our CoMID approach is a cyber-physical program P and its running environment E (conceptually). For the purpose of invariant generation, we assume the availability of a set of failure conditions (e.g., crashing of a UAV or falling down of a humanoid robot) for deciding whether a cyber-physical program's execution has already failed, as the existing work [24] does.

CoMID works in four steps: (1) it first executes program P in environment E to collect safe execution traces, i.e., no failure condition triggered (Step 1: *trace collection*); (2) it then groups iterations from the collected execution traces into multiple sets of *context-sharing iterations*, based on their program and environmental contexts (Step 2: *iteration grouping*); (3) after that, it generates multiple invariants for each group (Step 3: *multi-invariant generation*); (4) finally, it uses the generated invariants to detect abnormal states for program P's future executions (Step 4: *abnormal-state detection*). Fig. 3 illustrates CoMID's workflow.

In the first two steps, besides collecting traditional artifacts (e.g., arguments and return values for each executed method), CoMID also analyzes program and environmental contexts for each iteration. Regarding the program context, CoMID records what statements are executed in an iteration. Regarding the environmental context, CoMID records attribute values associated with environment E. CoMID recognizes P's system calls related to environmental sensing, and uses these calls to record attribute values at the beginning of each iteration. CoMID uses the program context to distinguish an iteration's specific strategy in handling external situations, and uses the environmental context to distinguish different situations that P is facing in a specific iteration.

In the last two steps, CoMID generates and checks multi-invariants to address the impact of uncertainty on deciding whether a specific invariant violation is a convincing indication that the current execution is no longer safe. CoMID leverages previous work (e.g., Daikon [16]) for invariant derivation by feeding different sets of sampled iterations.

We next elaborate on CoMID's details.

A. Context-based Trace Grouping (Steps 1 and 2)

Trace collection. In the first step, CoMID executes the given cyber-physical program P and collects its traces for invariant generation. For saving the cost, CoMID records values of program variables only at entry and exit points of the methods executed in each iteration. CoMID also records program and environmental contexts for each iteration, in order

to distinguish different iterations. For the program context, CoMID records the statements executed in each iteration through program instrumentation. For the environmental context, CoMID records values of environmental attributes using their involved system calls at the beginning of each iteration (i.e., once CoMID recognizes a new iteration).

CoMID extracts iterations from a collected execution trace by identifying P's input points, which indicate the start of each iteration and separate different iterations. For a cyber-physical program that iteratively interacts with its environment, it receives environmental inputs through periodically invoking system calls related to environmental sensing, e.g., reading a pressure sensor's value every 200 milliseconds, or sampling a picture from a camera per second. CoMID relies on such periodical environmental sensing and related system calls to decide such input points. For many cyber-physical programs, their system calls for environmental sensing have the same or similar invocation cycles, and this characteristic makes their inputs naturally free from being overlapping. A cyber-physical program may also conduct one-time sensing actions, e.g., reading an ultrasonic sensor's value to decide the distance to an obstacle in an ad hoc way. However, such one-time sensing actions can be easily distinguished from periodical sensing actions through analyzing their appearances in an execution trace.

Formally, we use *segment* to represent the collected information for each iteration in program P's execution. A segment abstracts P's execution state during an iteration. We use sg^i to represent P's state for its i-th iteration: sg^i = (P_{cxt}, E_{cxt}, M_1, M_2, ..., M_j), where

1) P_{cxt} represents the i-th iteration's program context, which is a set of identities (ids) of statements executed in the iteration.
2) E_{cxt} represents the i-th iteration's environmental context, which is a set of name-value pairs for sensing variables in P.
3) M_1, M_2, ..., M_j represent a sequence of methods executed in the i-th iteration, each of which contains a method's name, arguments, and return value.

CoMID conducts random testing on P and collects its execution traces. The random testing is according to P's targeted application scenarios, whose information is typically available when it is built or tested. For example, in our evaluation (Section IV), the NAO robot subject is designed to walk on wooden or brick floor, and the two UAV subjects are designed to fly on a sunny or cloudy day without strong wind. In addition, random testing has been shown to be simple, yet effective for exploring a program's diverse behaviors (e.g., Android Monkey testing [28] and Google's Waymo self-driving car testing [29]), which are useful for CoMID to generate invariants by studying these diverse behaviors from the cyber-physical program.

Then, according to P's associated failure conditions, one annotates whether a collected execution trace is safe or unsafe (i.e., whether violating any failure condition or not). We note that failure conditions can vary for different subjects, depending on their different tasks and execution environments. Still, there are three common suggestions for specifying failure conditions: (1) concerning a cyber-physical program's safety properties, e.g., a robot or UAV should never fall into the ground; (2) concerning liveness properties, e.g., a robot should not always be trapped in a small region; (3) concerning stableness properties, e.g., a UAV should not lose its height quickly in short time or lose its balance in the air. The set of safe execution traces forms the initial trace set for CoMID to learn and generate invariants from.

Iteration grouping. In the second step, CoMID groups iterations (segments) from the safe execution traces, so that each group contains only *context-sharing* ones. Here, contexts refer to program and environmental contexts recorded in the first step.

CoMID analyzes environmental contexts E_{cxt} recorded in segments to discover common patterns shared by iterations. It builds a set of all environmental contexts $ENV\_CONTEXT$, and conducts the k-means clustering algorithm [25] to form different clusters. We choose k-means clustering mainly for the performance consideration, since it is one of the most efficient clustering algorithms. For the same reason, CoMID considers only environmental attributes of numeric types in the clustering. It uses a normalized Euclidean metric to measure the distance between each pair of environmental contexts. Compared with the Euclidean metric, the normalized Euclidean metric can better measure the distance in a space whose dimensions have different scales. Since a cyber-physical program's sensing variables naturally have different scales according to involved sensors of different types, we choose the normalized Euclidean metric to measure the distance between two environmental contexts. Formally, given two environmental contexts E_{cxt}_A ($a\_A_1, a\_A_2, ..., a\_A_n$) and E_{cxt}_B ($a\_B_1, a\_B_2, ..., a\_B_n$), their distance $dis(E_{cxt}\_A, E_{cxt}\_B)$ is calculated as

$$dis(E_{cxt}\_A, E_{cxt}\_B) = \sum_{i=1}^{n} \sqrt{\frac{(a\_A_i - a\_B_i)^2}{s_i^2}},$$

where s_i^2 is the variance of all values of E_{cxt}'s i-th attributes in the $ENV\_CONTEXT$ set.

The k-means clustering algorithm [25] requires setting a suitable value for parameter k, which decides the maximal size of each formed cluster of environmental contexts. Generally, a small k value can make derived clusters more specific, but it could also increase noises in later classification [26]. Therefore, we choose the grid search [27], a traditional way of conducting parameter optimization in machine learning algorithms, to decide the most suitable value for parameter k. Intuitively, the grid search conducts cross-validation on a set of candidate values for the parameter to be optimized, and selects the one with the best performance.

We initially use 30 candidate values for parameter k, from 1% of the total number of collected environmental contexts to 30%, increasing with a pace of 1%. Then we conduct 10-fold cross-validation to decide the most suitable k value. We randomly divide the $ENV\_CONTEXT$ set into ten disjoint subsets of the same size. Nine subsets are merged for training (i.e., *training set*) and the remaining one is for validation (i.e., *testing set*). For each candidate k value, we conduct its corresponding clustering on the training set, resulting in

multiple clusters of environmental contexts. With respect to these clusters, the environmental contexts from the testing set are then classified into them. Accordingly, we calculate an average deviation value to measure the performance associated with the specific k value. Let an environmental context from the testing set be E_{cxt}_T, and its classified cluster be C (E_{cxt}_1, E_{cxt}_2, ..., E_{cxt}_j). Then context E_{cxt}_T's deviation value $div(E_{cxt}\_T)$ is calculated as

$$div(E_{cxt}\_T) = \frac{1}{j}\sum_{i=1}^{j} dis(E_{cxt}\_T, E_{cxt}\_i).$$

The *average deviation value* for k is the averaged deviation values of all environmental contexts from the testing set. One would expect this value to be minimized, and thus CoMID selects the k value with the smallest average deviation value after comparing all candidate values. In our field tests of the NAO robot and UAV subjects used later in our evaluation (Section IV), we observe that the selected k value ranges from 17% to 22% of the total number of collected environmental contexts with their corresponding performance being similar. Therefore, we select 20% of the total number as the k value used in CoMID to simplify its implementation and evaluation.

With the k value set for the k-means clustering, CoMID derives initial clusters for collected environmental contexts, and their belonging segments are also clustered accordingly. Then CoMID refines these initial clusters of segments based on their program contexts, by measuring the similarity of program contexts between segments in each cluster. CoMID uses the Jaccard similarity index [30] to calculate the Degree of Similarity (DoS) value between each pair of program contexts. Let P_{cxt}_sg be segment sg's program context (i.e., a set of statement ids). Then for two given segments sg_A and sg_B, the DoS value between their program contexts $DoS(P_{cxt}\_sg_A, P_{cxt}\_sg_B)$ is calculated as

$$DoS(P_{cxt}\_sg_A, P_{cxt}\_sg_B) = \frac{|P_{cxt}\_sg_A \cap P_{cxt}\_sg_B|}{|P_{cxt}\_sg_A \cup P_{cxt}\_sg_B|}.$$

Then the DoS value between a pair of program contexts ranges from 0 to 1. CoMID considers two segments to have the *same* program context if the DoS value of their program contexts is no less than 0.8. This reference value is set by following the existing work [24]. Nevertheless, we also study the impact of different DoS threshold values on CoMID's effectiveness in our later evaluation (Section IV).

Based on this similarity measurement on program contexts, CoMID refines the initial clusters of segments. If two segments in one cluster have the same program context, they are still together in that cluster. Otherwise, they are separated into two clusters. This separation process iterates until no cluster can be refined. Then the final result is a set of groups, each of which contains only segments with the same environmental and program contexts. We also say that each group contains *context-sharing iterations*.

Example. Consider in our robot example method `motion.angleMove(names, angle, timeLists)`. Fig. 4 illustrates CoMID's recorded information for the 8th and 12th iterations in an execution trace tr_A. The segment representing the 8th iteration, denoted as seg_A^8, is shown in the upper dashed box, and the segment representing the 12th iteration is shown in the lower box. For each segment, its

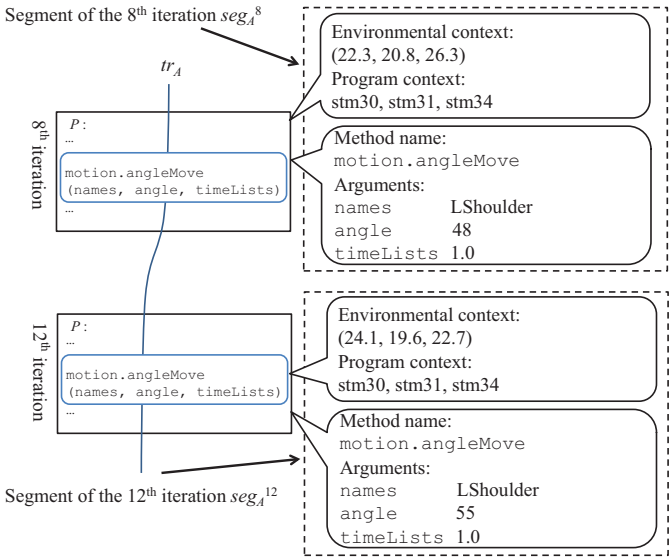
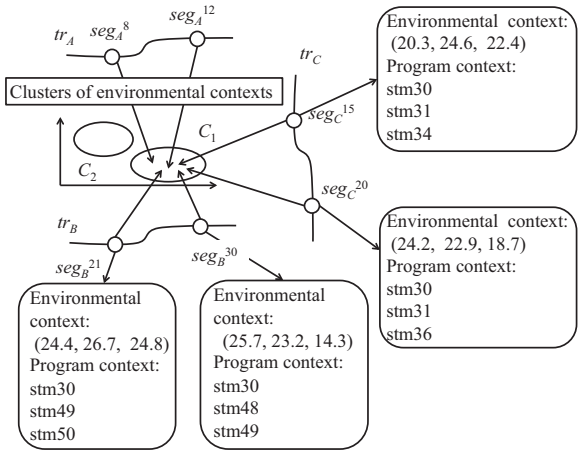
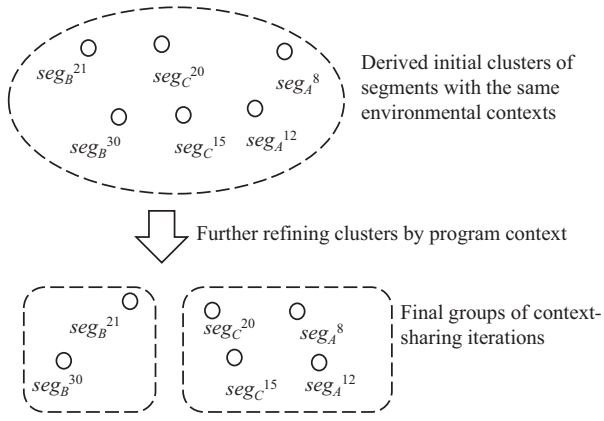


Fig. 4. Illustration of Step 1: Trace collection



(a) Deriving clusters by environmental context



(b) Refining clusters by program context to form final groups

Fig. 5. Illustration of Step 2: Iteration grouping

upper block lists the concerned iteration's environmental and program contexts, respectively, and its lower block lists the information for methods executed in this iteration (here we show one method for illustration). We use a tuple, e.g., (22.3, 20.8, 26.3), to represent the values of sensed environmental attributes, e.g., for the pressure on the robot's left foot, that on the right foot, and the robot's distance to its front-facing obstacle, respectively. We use "stm" followed by a number, e.g., "stm30", to represent the id of a statement executed in the concerned iteration. Note that this example is for the illustrative purpose and thus many aspects are simplified. For example, we list only three statements as program contexts (in reality, there can be many). As a result, a DoS threshold value of 0.8 between two program contexts is not effective to use, and one has to design more statements as program contexts to make this value useful. To make this example simple yet illustrative, we adopt another DoS threshold value of 0.5 here.

Fig. 5 illustrates how the iterations in three execution traces (tr_A, tr_B, and tr_C) are grouped according their environmental and program contexts. CoMID first derives initial clusters (Fig. 5-a) according to environmental contexts of the iterations, and cluster C_1 includes six iterations (seg_A^8 and seg_A^{12} from tr_A, seg_B^{21} and seg_B^{30} from tr_B, and seg_C^{15} and seg_C^{20} from tr_C). We show only their environmental and program contexts for illustration. CoMID then calculates DoS values for program contexts of the six iterations, and refines the C_1 cluster into two final groups (Fig. 5-b). One larger group contains four iterations (seg_A^8, seg_A^{12}, seg_C^{15}, and seg_C^{20}) from execution traces tr_A and tr_C, and the other smaller group contains two iterations (seg_B^{21} and seg_B^{30}) from trace tr_B. Such refinement result is due to their DoS calculations, e.g., *DoS*(seg_A^8, seg_C^{15}) = 1.0, *DoS*(seg_B^{21}, seg_B^{30}) = 0.5, *DoS*(seg_A^8, seg_B^{21}) = 0.2, and *DoS*(seg_B^{21}, seg_C^{15}) = 0.2, and so on.

B. Multi-invariant Detection (Steps 3 and 4)

Multi-invariant generation. After context-based trace grouping, CoMID obtains multiple groups of context-sharing iterations in terms of segments. CoMID feeds the segments in each group to the Daikon [16] engine for deriving invariants specific to this group. Note that the effectiveness of our CoMID approach is independent of the used invariant inference engine. Here we have chosen Daikon due to its wide usage and fair comparisons in our evaluation as explained later. One could also use other invariant inference engines for cyber-physical programs. In such cases, the artifacts collected in Step 1 (i.e., arguments and return values for each executed method) should be replaced by corresponding artifacts according to the actually used invariant inference engines. Nevertheless, program and environmental contexts should still be collected since they are required by our CoMID's technique of context-based trace grouping.

As mentioned earlier, CoMID needs to address the impact of uncertainty on invariant generation, so as to suppress the negative consequences of inaccurate sensing values. To do so, CoMID uses different subsets from each group of segments for deriving invariants, which are later used for collective checking in the runtime monitoring against uncertainty. Generally, one can freely decide the number of such subsets, and CoMID chooses four for avoiding high computational and monitoring overheads. The sizes of the sampled subsets can also be freely decided, and here CoMID makes the sizes of sampled subsets have equal differences (i.e., 20%, 40%, 60%, and 80% of the total number of segments in a group). We also study the impact of different sizes of sampled subsets on CoMID's effectiveness in our later evaluation (Section IV).

Then, besides the one invariant (i.e., *principal invariant*) for the universal set (i.e., a whole group of segments), CoMID generates four invariants for the four subsets, respectively. These five invariants are named as an invariant *family*, with respect to each supported invariant template and each executed method requiring invariant generation in the group. Since each invariant family is associated with a specific group of context-sharing iterations, the group's contexts are also referred as the *invariant family's context*. An invariant family's context specifies the situations under which the invariants in the family are suitable for checking, thus deciding abnormal states for concerned programs.

Abnormal-state detection. Now CoMID has generated a set of invariant families for runtime monitoring of each program location of interest. Different from the existing work [24], CoMID chooses to check only those invariant families whose contexts are the *same* as that of the current iteration in a program's execution. Here, "same" is decided by the comparisons of both program and environmental contexts: (1) the DoS value between a pair of program contexts no less than 0.8 (Section III.A), and (2) the environmental context of the current iteration is classified into the same cluster as that of the considered invariant family.

After selecting suitable invariant families for checking, CoMID then needs to decide whether an invariant violation in the runtime monitoring is simply caused by uncertainty or indicates the detection of a real abnormal state. CoMID uses an *estimation function* to ensemble the evaluation results of invariant checking across multiple iterations, in order to suppress the impact of uncertainty on the decision. The design of the estimation function is based on two *intuitions*:

1) The possibility that an invariant violation or satisfaction is caused by uncertainty relates to the number of segments that have been used for deriving the invariant under checking.
2) The impact of uncertainty on invariant checking can be suppressed by examining checking results across multiple consecutive iterations.

Based on these two intuitions, the estimation function assigns a weight to each invariant violation or satisfaction. The weight assignment is designed as follows:

1) For a violated invariant inv_1, the more segments are used for deriving it, the less possibility that inv_1's violation is caused by uncertainty, since inv_1 is inclined to be general.
2) For a satisfied invariant inv_2, the more segments are used for deriving it, the less possibility that inv_2's

satisfaction indicates the current execution to be passing, since satisfying a general invariant is natural.

Recall that CoMID makes five subsets for each group of segments (from 20% to 100% of the total size, with a pace of 20%), and generates invariants with respect to each of these subsets. Then given a subset of segments and its associated size ratio p (i.e., 20%, 40%, ..., or 100%), CoMID sets the weight assigned for the *violation* of one invariant generated from this subset to be p, and that for the *satisfaction* to be $-(1-p)$. Such a weight value intuitively models the likelihood whether an execution is failing or passing: a positive value suggests failing, while a negative value suggests passing, and its absolute value indicates the confidence.

Formally, consider an invariant family $INV = \{inv_i\}$, $1 \leq i \leq k$. Let the invariant-checking result for inv_i at iteration j be r_i^j, where 1 denotes invariant satisfaction and -1 denotes violation. Let the size ratio associated with invariant inv_i be p_i (from its corresponding segment subset). Then the estimation function returns for INV at iteration j as follows:

$$EST(INV)^j = \sum_{i=1}^{k} \begin{cases} \dfrac{p_i}{\sum_{x=1}^{k} p_x} & r_i^j = -1 \\ -\dfrac{1-p_i}{\sum_{x=1}^{k}(1-p_x)} & r_i^j = 1 \end{cases}$$

$EST(INV)^j$ calculates the sum of weighted checking results for all invariants in INV for iteration j. The estimation function then calculates the averaged result for the last w consecutive iterations (until j):

$$EST(INV)^{j-(w-1),j} = \frac{1}{w} \sum_{i=j-(w-1)}^{j} EST(INV)^i.$$

This averaged value falls in the range of $[-1, 1]$, and a value closer to 1 would be a strong indicator of a failing execution (i.e., having entered an abnormal state). Like existing work, CoMID needs to set up a threshold for this value to decide whether a monitored execution is failing. Since this value's fluctuation can be largely caused by the uncertainty, we assume that its distribution corresponds to that of the specific uncertainty type experienced by a cyber-physical program. Then based on the specific uncertainty type (i.e., its error range $[-U, U]$ and distribution D), CoMID sets up the threshold Δ by solving the uncertainty's C-confidence interval equation, i.e., $Pr(x \in [-U \times \Delta, U \times \Delta]) = C$, where $Pr(x)$ is the probability function for distribution D). For subjects such as the NAO robot and UAVs in our later evaluation, CoMID sets w = 5 and C = 90%. The former suggests 2–3 seconds before CoMID makes a decision, which is sufficient for such low-speed subjects to take new actions (customizable by application domains). The latter suggests that CoMID plans to hold a confidence level of 90% for its made decisions (also customizable by application domains). In the confidence interval equation, the probability function for most uncertainty types follow common models [31], facilitating the equation's solution. For example, if a specific certain type follows the uniform distribution, Δ would be solved to be 0.9; if it follows the normal distribution, Δ would be 0.65. By doing so, CoMID sets up the threshold Δ for deciding whether an averaged *EST* value implies the prediction of a failing execution, i.e., by checking whether the value is larger than Δ.

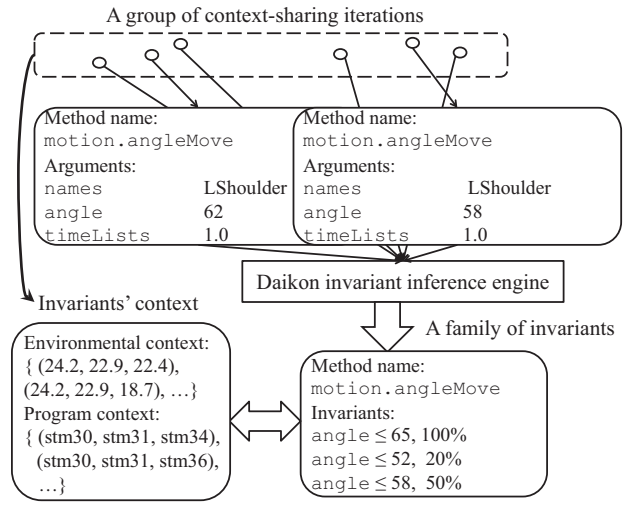


Fig. 6. Illustration of Step 3: Multi-invariant generation

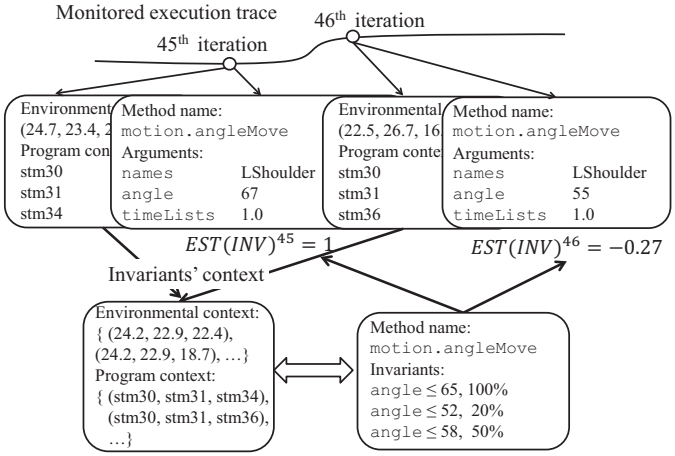


Fig. 7. Illustration of Step 4: Abnormal-state detection

Example. Consider in our robot example the variable *angle* for method `motion.angleMove(names, angle, timeLists)`. Fig. 6 illustrates an invariant family for this variable (showing three invariants for example), generated based on one group of context-sharing iterations. In this family, the principal invariant is "`angle` $\leq 65, 100\%$", indicating that the robot's arm should not be raised over 65 degrees in all cases. This invariant is generated based on all segments (i.e., 100%) in the concerned group. The other two invariants, namely, "`angle` $\leq 52, 20\%$" and "`angle` $\leq 58, 50\%$", are generated when only 20% and 50% (randomly sampled) segments are used. These invariants' context is also illustrated in Fig. 6 (from their corresponding group of segments).

Fig. 6 illustrates how CoMID uses the generated invariant family to detect abnormal states in the runtime monitoring. Consider the 45th and 46th iterations for a monitored execution trace (using two consecutive iterations for example, i.e., w = 2). Suppose that the earlier generated invariant family

(a) NAO robot (b) 4-rotor UAV (c) 6-rotor UAV

Fig. 8. Evaluation Subjects

shares the same context with both iterations. Then CoMID checks all three invariants in the family to decide whether the execution is safe or not. For the 45th iteration, its execution violates all the three invariants, and thus $EST(INV)^{45}$ is calculated to be 1 ($\frac{1}{1.7} + \frac{0.2}{1.7} + \frac{0.5}{1.7}$). For the 46th iteration, its execution violates only one invariant "$\texttt{angle} \leq 52, 20\%$", and thus $EST(INV)^{46}$ is calculated to be -0.27 ($-\frac{0}{1.3} + \frac{0.2}{1.7} - \frac{0.5}{1.3}$). So the averaged value of the estimation function for the execution consisting of the 45th and 46th iterations is 0.37 ($\frac{1-0.27}{2}$). If the uncertainty type follows the normal distribution, CoMID would solve the equation to obtain the threshold value to be 0.65, as explained earlier. Then the result (0.37) suggests that the monitored execution is still safe, and that the several invariant violations encountered in these two iterations have been possibly caused by uncertainty.

IV. Evaluation

In this section, we present the evaluation of our CoMID approach including comparing it with two existing approaches. The first approach *naïve* simply uses an invariant inference engine (i.e., Daikon [16]) to generate invariants. The second approach *p-context*, inspired by ZoomIn [24], uses program context to enhance invariant generation and abnormal-state detection. We select three real-world cyber-physical programs, namely, NAO robot (Fig. 8-a), 4-rotor UAV (Fig. 8-b), and 6-rotor UAV (Fig. 8-c), as the evaluation subjects. For the evaluation, we implement CoMID as a prototype tool in Java 8 and study the following three research questions:

RQ1: *How does CoMID compare with existing work in detecting abnormal states for cyber-physical programs in terms of effectiveness and efficiency?*

RQ2: *How does CoMID's configuration (e.g., enabling either or both built-in technique(s) for improving the generated invariants, setting up which DoS threshold value for distinguishing different program contexts in the invariant generation, and choosing which sizes of sampled subsets for multi-invariant generation) affect its effectiveness?*

RQ3: *How useful is CoMID-based runtime monitoring by invariant generation and checking for cyber-physical programs?*

A. Evaluation Subjects

We instrument the three evaluation subjects to record their program variable-value and context information during their executions. We use Daikon as the invariant inference engine for generating invariants from these subjects' execution traces. Besides the invariant templates internally supported by Daikon, we additionally add polygon invariant templates into Daikon, as suggested by existing work [4], [32] on runtime monitoring for cyber-physical programs. Note that CoMID is itself independent of the used invariant templates, and this feature makes it general to common cyber-physical programs.

The three evaluation subjects are from different companies or universities. The commercial NAO robot program contains 300 LOC (Python-based, with five methods). The two UAV programs are developed by professional electrical engineers, and contain 1,500 LOC (Java-based, with 24 methods) and 4,000 LOC (C-based, with 35 methods), respectively.

B. Evaluation Design and Setup

Execution-trace collection. In the evaluation, all invariants should be generated based on the execution traces collected from the selected evaluation subjects. For the evaluation purpose, we design various scenarios for our evaluation subjects to run with, and collect their execution traces accordingly. We test totally six scenarios and collect 1,200 execution traces (i.e., obtaining a total of 1,200 execution traces from six scenarios) for the three evaluation subjects.

We decide whether an execution trace is *safe* or not (i.e., the *oracle*) according to its corresponding program's behavior and whether its associated failure conditions have been triggered. The failure conditions discussed later seem ad hoc as they may not hold for other cyber-physical program subjects. Nevertheless, such failure conditions can hardly be general or systematic for a wide range of cyber-physical programs, as the latter can have varying requirements for being safe or functional. For example, the criterion for a NAO robot to stay balanced on the ground would be clearly different from that for a UAV to stay balanced when flying in the air. As such, failure conditions should probably be application-specific, as we design different failure conditions for the three subjects. If any failure condition is triggered, its corresponding subject program is directly decided and annotated to be *unsafe* in its execution. Based on such oracle information (safe or unsafe), we can later judge whether a specific approach under comparison gives a correct prediction or not (i.e., passing vs. safe, and failing vs. unsafe).

For the NAO robot (subject #1), we design a $3m \times 3m$ indoor area (including random obstacles and different floor materials) for free exploration. The NAO robot's failure conditions concern its safety (e.g., the robot should never fall into the ground or crash into any obstacle) and liveness (e.g., the robot should not be trapped in a small region). We collect a total of 200 execution traces, including 127 safe ones and 73 unsafe ones. We also build a simulated space with the same settings by the official NAO's emulator Webots [33], and collect 600 execution traces, which include 454 safe ones and 146 unsafe ones. We note that the Webots emulator also supports uncertain environmental sensing internally, and thus its emulated executions are accompanied with uncertainty naturally. However, both the subject program and all the approaches under comparison are unaware of such uncertainty. For ease of presentation, we use *NAO-f* and *NAO-e* to denote the two scenarios, i.e., field setting and emulation setting for the NAO robot, respectively.

For the 4-rotor UAV (subject #2), we design three field scenarios and collect 100 execution traces for each scenario due to battery constraints. In the first scenario, the UAV takes off from a starting point and lands at a remote destination. We collect 68 safe execution traces and 32 unsafe ones. In the second scenario, the UAV carries some balancing weight during its flying. We collect 71 safe execution traces and 29 unsafe ones. In the last scenario, the UAV conducts extra actions in addition to its normal flying plans, e.g., hovering and turning around. The failure conditions for the 4-rotor UAV concern its safety (e.g., a UAV should never fall into the ground or land outside a destination area) and stableness (e.g., a UAV should never lose its height quickly in short time or lose its balance in the air). We collect 64 safe execution traces and 36 unsafe ones. Similarly, we use *4-UAV-s1*, *4-UAV-s2*, and *4-UAV-s3* to denote the three scenarios, respectively.

For the 6-rotor UAV (subject #3), similarly it is scheduled to fly from a starting point to a remote destination. The 6-rotor UAV's failure conditions are the same as the 4-rotor UAV's. We collect 100 execution traces, including 76 safe executions and 24 unsafe ones. We design one field scenario for the evaluation and use *6-UAV* to denote this scenario.

Evaluation procedure. From the collected execution traces from various scenarios, all the approaches under comparison (i.e., CoMID, naïve, and p-context) generate invariants, which are evaluated for their qualities, in order to answer the three research questions. The evaluation is conducted on a commodity PC with an Intel(R) Core(TM) i7 CPU @4.2GHz and 32GB RAM. For each scenario, we run CoMID, naïve, and p-context on safe execution traces to generate invariants, respectively. Then we use safe and unsafe execution traces to validate their generated invariants in detecting abnormal states for the three evaluation subjects. We use 10-fold cross-validation in our evaluation. More specifically, for each scenario we divide the set of safe execution traces into ten subsets of the same size. One subset of safe execution traces (named the *safe set*) and the set of unsafe execution traces (named the *unsafe set*) are retained for validation. The remaining nine subsets of safe execution traces are used for invariant generation. We repeat this generation and validation process ten times and average their results as the final results for discussion.

To answer research question RQ1 (effectiveness and efficiency), we compare the invariants generated by the three approaches. For each approach, we first study the number of its generated invariants and the percentage of these invariants that can also be generated by other approaches. Since CoMID uses multi-invariant detection, we consider only its *principal invariants* for a fair comparison. We then study the effectiveness and efficiency of the invariants generated by the three approaches in detecting abnormal states for cyber-physical programs. We measure the effectiveness by the *true-positive* rate (TP, i.e., the percentage of unsafe execution traces that are predicted to be failing) for the unsafe set, and by the *false-positive* rate (FP, i.e., the percentage of safe execution traces that are predicted to be failing) for the safe set. Finally, we compare the efficiency for the three approaches by their time costs on invariant generation and checking.

To answer research question RQ2 (impact of configuration), we study CoMID's effectiveness (TP and FP) with its different configurations enabled: (1) on whether to enable one or both built-in technique(s) for improving the generated invariants, i.e., enabling context-based trace grouping only (Context), enabling multi-invariant detection only (Multi), or enabling both techniques (CoMID); (2) on how to set up a DoS threshold value for distinguishing program contexts in the invariant generation, i.e., from 0.6 to 1.0 with a pace of 0.1 (0.8 as the default setting, as explained in Section III.A); (3) on different sizes of sampled subsets for multi-invariant generation.

The first two research questions study the quality of CoMID's generated invariants based on offline execution traces that have been collected in advance. Research question RQ3 investigates how CoMID's abnormal-state detection helps to improve a cyber-physical program's safety in the runtime monitoring. Without CoMID-based runtime monitoring, the three evaluation subjects can rely on only their built-in *protection mechanisms* when their corresponding failure conditions are triggered. For example, when the robot is falling into the ground, it would control to stop walking and crouch on its knees; when a UAV is falling into the ground, it would control to stop rotating its wings. Such protection mechanisms can prevent the robot and UAVs from being damaged by the failures, but their planned tasks already fail. With CoMID-based runtime monitoring, the three evaluation subjects can use CoMID-based *recovery* in advance once CoMID detects abnormal states (i.e., predicting the current execution to be failing), and take remedy actions to prevent failure. Note that the original protection actions are invoked when failure conditions are satisfied (i.e., failures have already occurred, e.g., a robot is falling into the ground), while the remedy actions are invoked when CoMID detects any abnormal state (i.e., considering the current execution unsafe or failing). RQ3 aims to study the difference between these two setups regarding a cyber-physical program's recovery strategies (i.e., without monitoring vs. with monitoring, or original protection mechanism vs. CoMID-based remedy actions).

However, the carefully designed remedy actions are not the focus of CoMID, which focuses only on indicating when remedy actions should be invoked upon an abnormal state is detected. For comparison purposes, we adopt only very simple remedy actions for our evaluation subjects, i.e., suspending and then resuming current tasks after a short period of time. For example, the robot would stop walking, stand for two seconds, and then walk toward a different direction; a UAV would stop landing, reinitiate the flying plan, and then seek to land after two seconds. Although such remedy action can delay the subjects' planned tasks, the remedy action should be able to help avoid upcoming failures that would otherwise occur if no remedy action is taken.

To answer RQ3 (usefulness), we study how CoMID-based runtime monitoring helps the three evaluation subjects on preventing their failures. The failure data without CoMID-based runtime monitoring can be obtained from earlier collected execution traces for the three evaluation subjects in answering RQ1 and RQ2. For obtaining the failure data with

TABLE I
OVERVIEW OF THE GENERATED INVARIANTS BY THE THREE APPROACHES

	CoMID			Naïve			P-context		
	Inv	TP (%)	FP (%)	Inv	TP (%)	FP (%)	Inv	TP (%)	FP (%)
NAO-f	1,157 (33.0%)	85.9	18.3	978 (39.1%)	68.6	56.0	979 (38.0%)	78.5	43.9
NAO-e	1,313 (32.4%)	90.3	13.9	1,117 (38.1%)	79.1	44.0	1,117 (38.1%)	84.6	33.9
4-UAV-s1	860 (39.0%)	95.0	15.6	577 (58.1%)	77.5	40.0	577 (58.1%)	84.3	27.2
4-UAV-s2	802 (36.3%)	93.9	7.1	570 (51.1%)	65.7	30.7	570 (51.1%)	79.1	17.2
4-UAV-s3	933 (30.2%)	90.8	29.1	609 (46.3%)	75.5	49.4	609 (46.3%)	80.6	35.9
6-UAV	1,803 (33.0%)	92.0	12.2	1,527 (39.0%)	83.4	30.7	1,527 (39.0%)	85.9	18.9

CoMID-based runtime monitoring, we run the three evaluation subjects enabled with CoMID-based runtime monitoring and remedy mechanisms 100 times for each scenario, and average their results. Then we calculate and compare the success rates for the three evaluation subjects from the failure data. In addition, since the remedy mechanisms can delay the subjects' planned tasks, we study their impact by measuring and comparing the subjects' task-completion time (i.e., when a robot finishes its exploration task, and a UAV finishes its flying and landing tasks) for those non-failure executions.

C. Evaluation Results and Analyses

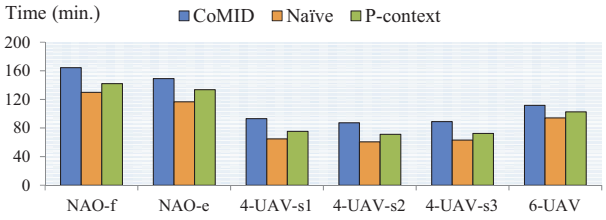
RQ1 (effectiveness and efficiency). Table I gives an overview of our evaluation results on the quality of the generated invariants by the three approaches under comparison. The table includes the number of generated invariants (Inv), true positive rate (TP) in detecting abnormal states for the unsafe set, and false positive rate (FP) in detecting abnormal states for the safe set. The percentage data in brackets after the invariant numbers give the proportions of the concerned invariants that can also be generated by other approaches. In general, CoMID generates more invariants than naïve and p-context (17.5–53.2% more, for different scenarios), even if we consider its principal invariants only. The reason is that CoMID generates different invariants to govern the program behavior for different situations by distinguishing different program and environmental contexts. Naïve and p-context generate the same numbers of invariants since they both generate the same invariants, although they check these invariants in different ways during the runtime monitoring, as shown later.

In addition, we observe that the invariants generated by CoMID are quite different from those generated by the other two approaches. For example, 30.2–39.0% of CoMID's invariants can be generated by the other two approaches, but 38.1–58.1% of the other two approaches' invariants can also be generated by CoMID. Considering that the number of CoMID's generated invariants is larger than those of the other two approaches' generated invariants, this result suggests that CoMID generates much more invariants that are unique from those generated by the other two approaches.

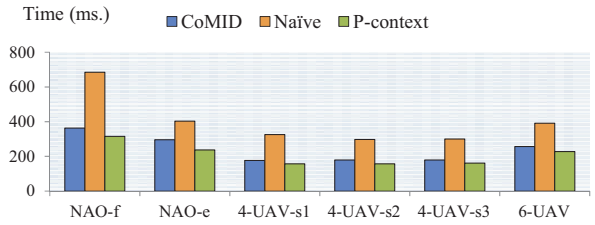
It is important to know whether these unique invariants bring the positive or negative impact on detecting abnormal states for the three evaluation subjects. We observe from Table I that these unique invariants enable CoMID to achieve a higher TP and a lower FP. For example, CoMID's TP is 8.6–28.2% higher than naïve and 5.7–14.7% higher than p-context, and at the same time, CoMID's FP is 18.6–37.6% lower than naïve and 6.8–25.5% lower than p-context. A high TP implies the ability of capturing various cases of abnormal states, and at the same time, a low FP implies that this ability is not achieved by the cost of overfitting the generated invariants to specific cases. Therefore, this result suggests that CoMID's generated invariants are of a high quality, by achieving both a high TP and a low FP. It also indicates that CoMID deserves its efforts on particularly addressing the iterative execution and uncertain interaction characteristics of cyber-physical programs. For the iterative execution, p-context partially uses program contexts to distinguish different scopes for different invariants, and thus performs better than naïve, which does not consider any context at all. For the uncertain interaction, different levels of uncertainty result in CoMID's varying leading advantages in FP for different evaluation subjects. For example, compared with p-context, CoMID achieves a 20.0–25.5% lower FP for the NAO robot, and a 6.8–11.2% lower FP for the two UAVs.

We note that CoMID's reported FP varies between different subjects (7.1–29.9%). Considering different subjects' various deployment platforms and environments, a direct comparison across different subjects may not make much sense. Nevertheless, we make a further investigation into CoMID's FP results. We find that the false positives are mainly caused by the uncertainty (e.g., inaccurate sensing) associated with these subjects. For example, in scenarios where a subject suffers more from uncertainty, e.g., the in-field scenario (NAO-f) of the NAO robot, all three studied approaches report a higher FP (4.4–12.0% higher, as shown in Table I), as compared with scenarios where a subject suffers less from uncertainty, e.g., the emulated scenario (NAO-e) of the NAO robot.

We then compare the efficiency for the three approaches in generating invariants and checking these invariants for detecting abnormal states. Fig. 9-a compares these approaches' time costs in generating invariants. We observe that CoMID spends 18.7–43.6% more time than naïve and 8.9–23.5% more than p-context in generating invariants. Naïve spends the least time due to its straightforward strategy of invariant generation by overlooking all contexts. CoMID's higher time cost is due to its constituent techniques of context-based trace grouping and multi-invariant generation for improving the quality of generated invariants. For the former, CoMID groups context-sharing iterations to make its generated invariants fitter to specific program behaviors, bringing up its TP in detecting abnormal states. For the latter, CoMID uses multiple invariants to alleviate the impact of uncertainty, bringing down its FP in detecting abnormal states.

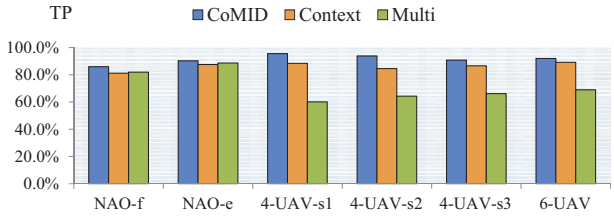


(a) For generating invariants

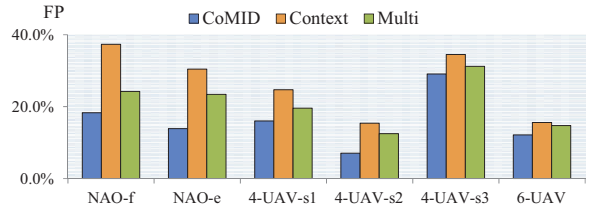


(b) For checking invariants

Fig. 9. Efficiency comparison for CoMID, naïve, and p-context



(a) TP comparison



(b) FP comparison

Fig. 10. Effectiveness comparison for CoMID, Context, and Multi

Fig. 9-b compares the three approaches' time costs in checking invariants for runtime monitoring. For the collected execution traces, which are four-minute length on average, CoMID's total time overhead is less than 400 milliseconds (176.5–363.5 milliseconds, or 241.9 milliseconds on average). Considering that CoMID checks invariants only at the end of each iteration, the time overhead is actually split into multiple pieces for each iteration, and each piece is very small. We observe that CoMID spends 36.3–88.5% less time than naïve. Although CoMID uses multiple invariants to decide abnormal states, its technique of context-based trace grouping enables it to focus on much fewer invariants specific for each iteration encountered by a cyber-physical program. naïve, instead, has to check each invariant in each iteration, resulting in its high time cost in detecting abnormal states. Regarding p-context, whose total time overhead is about 200 milliseconds (157.3–315.7 milliseconds, or 209.4 milliseconds on average), CoMID is acceptable (only slightly more time), considering that it has additionally considered environmental contexts for refining invariants and addressed the impact of uncertainty in checking invariants. Therefore, CoMID should be useful for many real-world cyber-physical programs, which include, but not limited to, the three evaluation subjects.

However, due to the variety of different cyber-physical programs, one can hardly claim that CoMID applies to all of them. We suggest characterizing CoMID's applicable cyber-physical programs according to their iteration lengths in terms of the execution time for one iteration. Since CoMID checks invariants only at the end of each iteration, a cyber-physical programs's iteration length would largely affect, if not deciding, whether CoMID's time overhead is sufficiently small and affordable. For example, if a cyber-physical program has an iteration length of about 100 milliseconds or longer (i.e., sensing its environment less than ten times per second), then CoMID is applicable (e.g., for our evaluation subjects, the iteration length for the NAO robot is about 500 milliseconds and those for the two UAV subjects are about 200 milliseconds). The reason is that CoMID takes about 0.6 millisecond for each iteration (Fig. 9-b, average total time is 241.9 milliseconds, and average iteration number is 420). Still, CoMID's own time overhead depends on how many invariants should be checked, being quite application-specific. For our evaluation subjects, the numbers of checked invariants for the three subjects are 802–1,803. For other cyber-physical programs, one can decide CoMID's applicability based on their invariant numbers accordingly.

Therefore, we answer **research question RQ1** as follows.

CoMID generates and checks invariants to detect abnormal states for cyber-physical programs effectively and efficiently. It achieves a higher TP (5.7–28.2% higher) and a lower FP (6.8–37.6% lower) than naïve and p-context. Although CoMID spends more time in generating invariants (offline), its invariant checking (online) is comparably efficient as p-context and much more efficient than naïve.

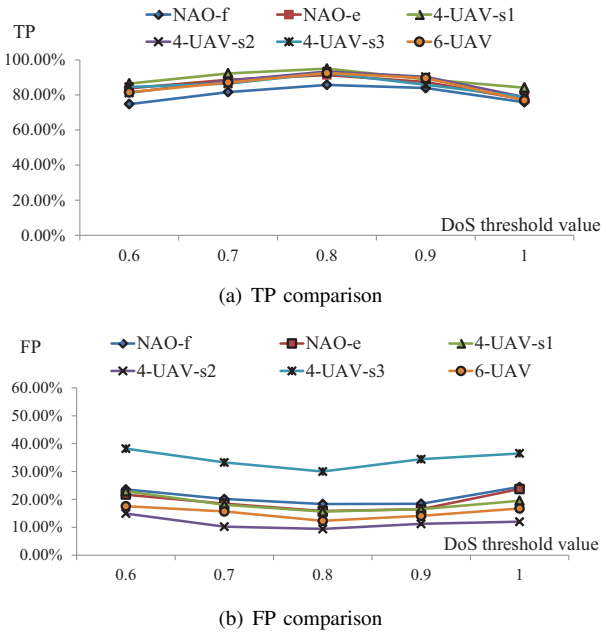
RQ2 (impact of configuration). We study the impact of configurations on CoMID's effectiveness from two aspects. First, CoMID can be configured with its two built-in techniques (context-based trace grouping and multi-invariant detection) individually enabled. Fig. 10 compares the effectiveness in terms of TP and FP for the original CoMID (CoMID), CoMID with only context-based trace grouping enabled (Context), and CoMID with only multi-invariant detection enabled (Multi). We observe that when detecting abnormal states for the unsafe set, Context performs more effectively than Multi in four UAV scenarios (20.2–28.3% higher TP), while Multi performs more effectively than Context in two NAO scenarios (0.7–1.1% higher TP). As analyzed earlier, the NAO robot suffers more from uncertainty than the two UAVs due to its complicated sensing and physical behavior, and thus Multi helps more than Context for the two NAO scenarios on suppressing the impact of uncertainty. For the four UAV-related scenarios, their uncertainty is relatively light, and thus Context exhibits more substantial advantages.

When combining the two techniques together, CoMID always produces the best results (2.7–35.4% higher TP). On the other hand, when suppressing false alarms for the safe set, Multi performs more effectively than Context in all six scenarios (0.9–13.3% lower FP). The advantages of Multi are mainly caused by the fact that uncertainty is the major reason for false alarms. Still, CoMID again produces the best results (2.1–9.5% lower FP). Considering that Context and Multi behave better in different scenarios (complementing each other) and CoMID always produces the best results, CoMID's two techniques (context-based trace grouping and multi-invariant detection) are both useful for improving its effectiveness by achieving a high TP and a low FP.

The p-context approach (inspired by the existing work ZoomIn [24]) uses program contexts to specify effective scopes for its generated invariants, but does not explicitly address the uncertainty issue. When its reported FP is compared with Context (i.e., CoMID without addressing the uncertainty), the latter obtains only a 1.4–3.3% lower FP rate than the p-context approach as shown in Table I, but CoMID with both its techniques enabled (i.e., addressing the uncertainty) obtains a 6.8–25.5% lower FP rate than the p-context approach. This result demonstrates CoMID's strengths in alleviating the impact of uncertainty to cyber-physical programs. This result also suggests that the p-context approach can still be effective for subjects with less uncertainty.

Second, CoMID can also be configured to use different DoS threshold values for distinguishing different program contexts in generating invariants. As mentioned earlier, CoMID uses a default DoS threshold value of 0.8 as suggested by the existing work [24], and here we study the impact of this value choice (from 0.6 to 1.0 with a pace of 0.1) on CoMID's effectiveness. Fig. 11 compares CoMID's effectiveness in terms of TP and FP with different DoS threshold values. We observe that in all six scenarios, CoMID with the value of 0.8 indeed behaves the best in both TP and FP. Nevertheless, the winning extents are not that large, and the extent on TP (1.8–15.6% higher) is a bit more than that on FP (0.1–7.9% lower). In addition, we observe that the impact of different DoS threshold values varies across different scenarios. For example, in scenario NAO-f, the TP for threshold 0.9 behaves slightly better than that for threshold 0.7, while in scenario NAO-e, the latter behaves slightly better than the former. This result suggests that CoMID's effectiveness might be further improved if its DoS threshold value can be tuned adaptively for specific cyber-physical programs. Currently, we make CoMID take the default value of 0.8 for simplicity, and we leave its adaptive tuning to future work.

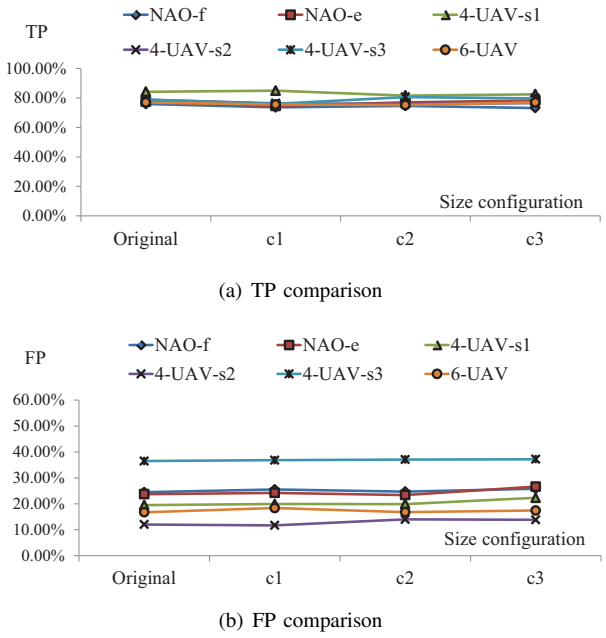
Third, CoMID samples four subsets from a group of context-sharing iterations, each containing 20%, 40%, 60%, and 80% of the total number of segments in a group, for multi-invariant generation. Now we study the impact of different sizes of sampled subsets on CoMID's effectiveness. Besides the original size configuration (**original**), we consider three other size configurations: (1) four subsets each containing 20%, 30%, 50%, and 70% of the total number of segments in a group (**c1**); (2) containing 20%, 50%, 70%, and 90% (**c2**); (3) containing 20%, 26%, 36%, and 53% (**c3**). While the first two size configurations are manually set to make their size differences not equal, the last one is randomly set for comparison.



(a) TP comparison

(b) FP comparison

Fig. 11. Effectiveness comparison for CoMID with different DoS threshold values



(a) TP comparison

(b) FP comparison

Fig. 12. Effectiveness comparison for CoMID with different size configurations for sampled subsets

Fig. 12 compares CoMID's effectiveness in terms of TP and FP with different size configurations for sampled subsets. We observe that CoMID's average effectiveness with the original size configuration is the best among the four compared configurations in both TP (78.7%) and FP (22.2%). Nevertheless, the wining extents are not that large (0.8–1.9% higher TP, and 0.5–1.7% lower FP). This result suggests that

changing to another size configuration can have only limited impact on CoMID's effectiveness. In addition, we observe that although CoMID with the original size configuration achieves the best average effectiveness, CoMID with other size configurations can achieve the best effectiveness for specific scenarios. For example, considering TP, c1 performs the best in scenario 4-UAV-s1, c2 performs the best in scenario 4-UAV-s3, and c3 performs the best in both scenarios NAO-e and 6-UAV. Similar to the DoS threshold value setting, the result also suggests that CoMID's effectiveness might be potentially further improved if its adopted sizes of sampled subsets for multi-invariant generation can be tuned adaptively for specific cyber-physical programs.

Therefore, we answer **research question RQ2** as follows.

CoMID's configurations affect its effectiveness. First, CoMID's two built-in techniques are both useful. When the uncertainty affecting the three evaluation subjects is relatively light, CoMID with only context-based trace grouping enabled already behaves quite well. When the uncertainty is relatively heavy, CoMID with only multi-invariant detection enabled behaves better. In either way, combing both techniques (i.e., a full-fledged CoMID) produces the best results. Second, CoMID's settings of its DoS threshold value for distinguishing different program contexts, as well as sizes of its sampled subsets for multi-invariant generation, also affect its effectiveness, but not substantially. Its current configuration (i.e., DoS threshold value set to 0.8, and sizes of sampled subsets set to 20%, 40%, 60%, and 80% of the total number of segments in a group) already makes it work satisfactorily for the three evaluation subjects.

RQ3 (usefulness). Finally, we study how CoMID-based runtime monitoring helps the three evaluation subjects on preventing their potential failures. Fig. 13 compares the success rate for the three evaluation subjects in the six scenarios, based on their failure data with ("with CoMID") and without ("without CoMID") CoMID-based runtime monitoring. We observe that CoMID indeed helps improve the success rate by 15.3–31.7% (avg. 23.1%) across different scenarios. This result echoes our earlier evaluation results on CoMID's high TP and low FP performance. In addition, as mentioned earlier, the CoMID-based runtime monitoring and remedy mechanisms can delay the three evaluation subjects' planned tasks, thus trading for higher safety (i.e., fewer failures). So we study such impact. Fig. 14 compares the average task-completion time for non-failure executions of the three evaluation subjects with ("with CoMID") and without ("without CoMID") CoMID-based runtime monitoring. We observe that CoMID indeed increases the subjects' task-completion time by 8.8–35.2% (avg. 26.8%). We consider such slowdown extent acceptable for subjects that require high safety assurance. In fact, the delay is largely due to the safety control before reinitializing the tasks (e.g., a robot stands for two seconds and then restarts walking, and a UAV restarts to land after two seconds), customizable by different application domains.

Therefore, we answer **research question RQ3** as follows.

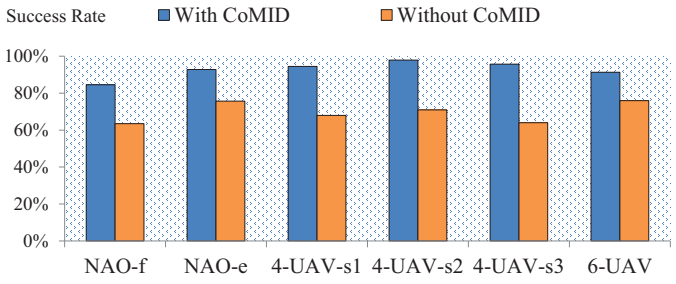


Fig. 13. Success rate for monitored cyber-physical programs

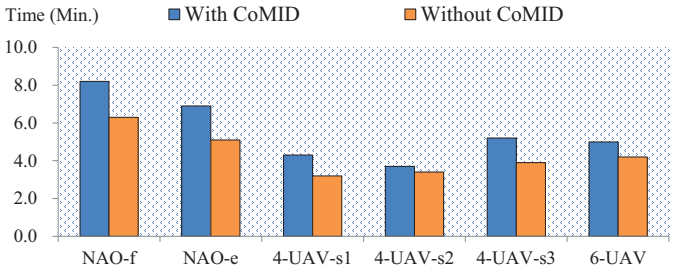


Fig. 14. Average task-completion time for monitored cyber-physical programs

CoMID's capability of generating and checking invariants for runtime monitoring can effectively prevent the three evaluation subjects from entering potential failures. CoMID helps improve the subjects' success rate in their task executions by 15.3–31.7%, with a cost of 8.8–35.2% longer task-completion time.

D. Threats to Validity

One major concern on the validity of our empirical conclusions is the selection of evaluation subjects in our evaluation. We select only three evaluation subjects, which may not allow our conclusions to be generalized to more other subjects. Nevertheless, a comprehensive evaluation requires the support of suitable environments for experimentation, which should be both observable and controllable. This requirement restricts our choice of possible evaluation subjects. To alleviate this threat, we try to make our subjects *realistic* by selecting real-world cyber-physical programs. In addition, we make the subjects *diverse* by requesting them to cover different functionalities (e.g., automated area exploration, planned flying, and smart obstacle avoidance), and to run on different platforms (e.g., Python-based NAO robot, Java-based UAV, and C-based UAV). By doing so, we try to alleviate as much as possible potential threat to the external validity of our empirical conclusions. Still, evaluating CoMID on more comprehensive cyber-physical programs and platforms deserves further efforts.

Another concern is about relating the detection of an abnormal state to an execution's failure result; such factor may pose threat to internal validity of our empirical conclusions on an approach's TP and FP performance. The reason is that when an abnormal state is detected by an approach, one seems not able to clearly relate the detection to the current execution's

upcoming failure, considering that their time interval can vary. To address this problem, we particularly design to measure TP for unsafe executions and FP for safe executions only: (1) for an unsafe execution, if an approach never detects any abnormal state, such result suggests its weakness (it should detect), and so we choose to check whether the approach reports the detection of any abnormal state, i.e., TP; (2) on the other hand, for a safe execution, if an approach reports the detection of any abnormal state, such result also suggests its weakness (it should not detect), and so we directly check whether the approach produces such false alarms, i.e., FP. In addition, to further alleviate the potential threat, we additionally study in research question RQ3 whether CoMID-based runtime monitoring indeed helps prevent the three evaluation subjects from entering failures, i.e., by measuring and comparing their success rates in task executions. All together, we strive our best efforts to evaluate CoMID's empirical and practical usefulness for cyber-physical programs.

Last but not least, the failure conditions used for annotating safe/unsafe execution traces may threat the validity of our empirical conclusions. Failure conditions' not being satisfied does not necessarily indicate that the current execution is passing (i.e., should be a candidate to be annotated as a "safe" one) at this moment. What we can assure is that when failure conditions are satisfied, the current execution is indeed failing (i.e., should be annotated as an "unsafe" one). If not, one has not yet observed any evidence showing that the current execution will necessarily fail in the future. Therefore, we consider that the execution is still passing at this moment. Note that this treatment applies to all the approaches under comparison, and therefore should not affect much our empirical conclusions.

V. Related Work

In this section, we discuss representative related work on testing cyber-physical programs, generating program invariants, and runtime monitoring, respectively.

Testing cyber-physical programs. Cyber-physical programs are featured with context-awareness, adaptability, and uncertain program-environmental interactions, which bring substantial challenges to their quality assurance. To address this problem, various approaches have been proposed for effective testing of such programs. For example, Fredericks et al. [34] use utility functions to guide the design and evolution of test cases for cyber-physical programs. Xu et al. [13] propose monitoring common error patterns at the runtime of cyber-physical programs, to identify defects in their adaptation logics when interacting with uncertain environments. Ramires et al. [35] explore specific combinations of environmental conditions to trigger specification-violating behaviors in adaptive systems. Yi et al. [36] propose a white-box sampling-based approach to systematically exploring the state space of an adaptive program, by filtering out unnecessary space samplings whose explorations would not contribute to detecting program faults. These preceding approaches exploit different observations to strengthen their testing effectiveness, but rely mostly on human-written or domain-specific properties for defining abnormal or error states in executing programs. Our CoMID approach complements these preceding approaches by assisting their fault-detection capabilities from checking trivial failure conditions (e.g., system crashes) to comprehensive errors (e.g., various types of error state) with automatically generated invariants.

Generating program invariants. Dynamically inferring invariants is spearheaded by the Daikon [16] approach. The approach instantiates several pre-defined property templates to produce candidate properties, and uses test runs to discard candidate properties that are violated. The remaining set of candidate properties are maintained as the likely invariants. DySy [37] is an approach that combines test runs with symbolic execution. Like Daikon, DySy uses test runs but simultaneously performs symbolic execution to collect path conditions and symbolic constraints for a method's return value and the receiver object's instance variables. From these path conditions and symbolic constraints, DySy derives the method's preconditions and postconditions. PreInfer [38] also combines test runs with symbolic execution but, unlike DySy, PreInfer conducts pruning and template-based abstraction for loops to infer concise quantified invariants. Jiang et al. [4] derive invariants by observing messages exchanged between system nodes, and specify operational attributes for robotic systems based on these messages. Zhang et al. [39] use symbolic execution as a feedback mechanism to refine the set of candidate invariants generated by Daikon. Carzaniga et al. [40] propose cross-checking invariant-alike oracles by exploiting intrinsic redundancy of software systems. Different from these preceding approaches, our CoMID approach additionally considers the impact of contexts on invariant generation (to restrict invariants' effective scopes) and that of uncertainty on invariant checking (to suppress false alarms), specially catered for the characteristics of cyber-physical programs.

Runtime monitoring. By means of invariant checking, one is able to detect abnormal states or anomalous behaviors in a program's execution. Detecting abnormal states early can allow the program to execute alternative actions to avoid danger. Zheng et al. [41] mine predicate rules that specify what must hold at certain program points (e.g., branches and exit points) for runtime monitoring. Raz et al. [42] derive constraints on values returned by data sources, and identify abnormal values based on the derived constraints. Pastore et al. [24] use the statement-coverage information in a program's execution to improve the precision of abnormality detection. Nadi et al. [43] extract configuration constraints from program code, and use the constraints to enforce expected runtime behaviors. Xu et al. [44] collect the calling contexts of method invocations, and use the contexts to distinguish a program's different behaviors under different scenarios. The preceding approaches share a common assumption that a program execution's anomalous behaviors can be discovered by checking newly collected execution data against earlier derived constraints from assumed normal executions. While this assumption is generally correct, cyber-physical programs' two characteristics, i.e., iterative execution and uncertain interaction as discussed earlier, make the preceding approaches less effective. The main reason is that different iterations

in a cyber-physical program's execution can face different situations and undertake different strategies to handle these situations. Then a straightforward invariant-checking approach can easily generate false alarms when the derived invariants' scopes differ and the impact of uncertainty is overlooked. Our CoMID approach specifically addresses this problem and thus complements existing work on effective runtime monitoring.

VI. Conclusion and Future work

In this article, we have presented a novel approach, CoMID, for effectively generating and checking invariants to detect abnormal states for cyber-physical programs. CoMID distinguishes different contexts for invariants and makes them context-aware, so that its generated invariants can be effective for varying situations and at the same time robust to uncontrollable uncertainty faced by cyber-physical programs. Our evaluation with real-world cyber-physical programs demonstrates CoMID's effectiveness in improving the true-positive rate and reducing the false-positive rate in detecting abnormal states, as compared with two state-of-the-art invariant generation approaches.

CoMID still has room for improvement. For example, it currently records the values of program variables at entry and exit points of all executed methods, and uses these variable values to generate invariants. Monitoring all executed methods greatly increases the time overhead of CoMID, and makes it less effective when applied to a time-critical cyber-physical program (e.g., a program whose iteration length is less than 100 milliseconds, as discussed in Section IV-C). One promising way is to restrict the invocations of Daikon to important methods only, as suggested by other Daikon-based work [45]. In addition, CoMID currently uses the default DoS threshold value of 0.8 as suggested by existing work [24]. In our evaluation, we observe the opportunities in which different threshold values can bring higher quality of runtime monitoring for different scenarios. Therefore, it is also worth exploring how to design adaptive DoS threshold tuning for further refined invariant generation and checking, as our future work.

CoMID also brings new research opportunities. Once CoMID detects abnormal states, one has to correct the monitored cyber-physical program's current execution, in order to prevent it from reaching a failure. In our evaluation, we use a straightforward strategy to design the remedy actions, since remedy is not the focus of this article. Considering the open environment surrounding cyber-physical programs, it is very challenging to design such simple yet effective remedy actions. One possible way is to exploit the invariant-violation information. When CoMID reports an invariant violation, it not only detects the anomalies of the variable values of a cyber-physical program, but also describes the program's internal and external situation through program and environmental contexts. By checking the program's safe executions under similar situations, one could possibly interpret the situation with the program's present violation, and map this information to proper remedy actions. This direction deserves further effort to investigate.

Acknowledgment

The authors would like to thank the editor and anonymous reviewers for their valuable comments on improving this article. This work was supported in part by National Key R&D Program (Grant #2017YFB1001801) and National Natural Science Foundation (Grants #61690204) of China. The authors would also like to thank the support of the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China. This work was supported in part by National Science Foundation under grants no. CNS-1513939, CNS-1564274, CCF-1816615, and a GEM fellowship.

References

[1] W. Yang, C. Xu, Y. Liu, C. Cao, X. Ma, and J. Lu, "Verifying self-adaptive applications suffering uncertainty," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE'14, 2014, pp. 199-210.

[2] R. Okuda R, Y. Kajiwara, and K. Terashima, "A survey of technical trend of ADAS and autonomous driving," in *Proceedings of Technical Program International Symposium on on InVLSI Technology, Systems and Application*, ser. VLSI-TSA'14, 2014, pp. 1-4.

[3] R. Pahuja, and N. Kumar, "Android mobile phone controlled bluetooth robot using 8051 microcontroller," in *International Journal of Scientific Engineering and Research*, ser. IJSER, vol. 2, 2014, pp. 15-24.

[4] H. Jiang, S. Elbaum, and C. Detweiler, "Reducing failure rates of robotic systems though inferred invariants monitoring," in *Proceedings of the 2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, ser. IROS'13, 2013, pp. 1899-1906.

[5] V. Roberge, M. Tarbouchi, and G. Labonte, "Comparison of parallel genetic algorithm and particle swarm optimization for real-time UAV path planning," in *IEEE Transactions on Industrial Informatics*, vol. 9, no. 1, 2013, pp. 132-141.

[6] M. Orsag, C. Korpela, and P. Oh, "Modeling and control of MM-UAV: Mobile manipulating unmanned aerial vehicle," in *Journal of Intelligent and Robotic Systems*, 2013, pp. 1-14.

[7] https://www.ald.softbankrobotics.com/en/cool-robots/nao.

[8] Audren, Herv, J. Vaillant, A. Kheddar, A. Escande, K. Kaneko, and E. Yoshida, "Model preview control in multi-contact motion-application to a humanoid robot," in *Procedding of the 2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, ser. IROS'14, 2014, pp. 4030-4035.

[9] Z. Liu, C. Chen, Y. Zhang, and C.P. Chen, "Adaptive neural control for dual-arm coordination of humanoid robot with unknown nonlinearities in output mechanism," in *IEEE Transactions on Cybernetics*, vol. 45, no. 3, 2015, pp. 507-518.

[10] E. Fredericks, B. DeVries, and Betty H. C. Cheng, "Towards run-time adaptation of test cases for self-adaptive systems in the face of uncertainty," in *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, , ser. SEAMS'14, 2014, pp. 17-26.

[11] D. Kulkarni and A. Tripathi, "A framework for programming robust context-aware applications," *IEEE Transactions on Software Engineering*, vol. 36, no. 2, 2010, pp. 184-197.

[12] M. Sama, S. Elbaum, F. Raimondi, D. S. Rosenblum, and Z. Wang, "Context-aware adaptive applications: Fault patterns and their automated identification," *IEEE Transactions on Software Engineering*, vol. 36, no. 5, Sep. 2010, pp. 644-661.

[13] C. Xu, S. C. Cheung, X. Ma, C. Cao, and J. Lu, "Adam: Identifying defects in context-aware adaptation," *Jounral of Systems and Software*, vol. 85, no. 12, 2012, pp. 2812-2828.

[14] A. J. Ramirez, A. C. Jensen, and B. H. C. Cheng, "A taxonomy of uncertainty for dynamically adaptive systems," in *Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, ser. SEAMS'12, 2012, pp. 99-108.

[15] B. H. e. a. Cheng, "Software engineering for self-adaptive systems," in *Software Engineering for Self-Adaptive Systems: A Research Roadmap*, LNCS vol. 5525, 2009, pp. 1-26.

[16] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *IEEE Transactions on Software Engineering*, vol. 27, no. 2, 2001, pp. 99-123.

[17] T. Nguyen, D. Kapur, W. Weimer, and S. Forrest, "Using dynamic analysis to generate disjunctive invariants," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE'14, 2014, pp. 608-619.

[18] L. Zhang, G. Yang, N. Rungta, S. Person, and S. Khurshid, "Feedback-driven dynamic invariant discovery," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA'14, 2014, pp. 362-372.

[19] S. Bensalem, M. Bozga, B. Boyer, and A. Legay, "Incremental generation of linear invariants for component-based systems," in *Proceedings of 13th International Conference on Application of Concurrency to System Design*, ser. ACSD'13, 2013, pp. 80-89.

[20] C. D. Nguyen, A. Marchetto, and P. Tonella, "Automated oracles: An empirical study on cost and effectiveness," in *Proceedings of the 21th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, ser. FSE'13, 2013, pp. 136-146.

[21] N. Esfahani, E. Kouroshfar, and S. Malek, "Taming uncertainty in self-adaptive software," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE'11, 2011, pp. 234-244.

[22] D. Opitz, and R. Maclin, "Popular ensemble methods: An empirical study," in *Journal of Artificial Intelligence Research*, vol. 11, 1999, pp. 169-198.

[23] http://www.bspilot.com/.

[24] F. Pastore, and L. Mariani, "ZoomIn: Discovering failures by detecting wrong assertions," in *Proceedings of the 37th International Conference on Software Engineering*, ser. ICSE'15, 2015, pp. 66-76.

[25] J. A. Hartigan, and M. A. Wong, "Algorithm AS 136: A k-means clustering algorithm," in *Journal of the Royal Statistical Society*, Series C (Applied Statistics) 28, no. 1, 1979, pp. 100-108.

[26] B.S. Everitt, S. Landau, M. Leese and D. Stahl, "Miscellaneous Clustering Methods," in *Cluster Analysis*, 5th Edition, 2011, John Wiley & Sons, Ltd, Chichester, UK.

[27] C. C. Chang and C. J. Lin, "LIBSVM: A library for support vector machines," in *ACM Transactions on Intelligent Systems and Technology*, vol. 2, no. 3, 2011, pp. 1-27.

[28] UI/Application Exerciser Monkey, https://developer.android.com/studio/test/monkey.

[29] Google Auto Waymo Disengagement Report for Autonomous Driving. https://www.dmv.ca.gov/portal/wcm/connect/946b3502-c959-4e3b-b119-91319c27788f/GoogleAutoWaymo_disengage_report_2016.pdf?MOD=AJPERES, 2016.

[30] M. Levandowsky, and D. Winter, "Distance between sets." in *Nature*, vol. 234, no. 5323, 1971, pp. 34-35.

[31] W. Yang, C. Xu, M. Pan, X. MA, and J. Lv, "Improving verification accuracy of CPS by modeling and calibrating interaction uncertainty," in *ACM Transactions on Internet Technology*, vol. 18 no. 2, 2018, Article 20.

[32] Cachera, David, and Florent Kirchner. "Inference of polynomial invariants for imperative programs: A farewell to Grebner bases," in *Science of Computer Programming*, vol. 93, 2013, pp. 89-109.

[33] http://www.cyberbotics.com/webots.php.

[34] E. M. Fredericks, B. DeVries, and B. H. C. Cheng, "Towards runtime adaptation of test cases for self-adaptive systems in the face of uncertainty," in *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, ser. SEAMS'14, 2014, pp. 17-26.

[35] A. J. Ramirez, A. C. Jensen, B. H. C. Cheng, and D. B. Knoester, "Automatically exploring how uncertainty impacts behavior of dynamically adaptive systems," in *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE'11, 2011, pp. 568-571.

[36] Y. Qin, C. Xu, P. Yu and J. Lu, "SIT: Sampling-based interactive testing for self-adaptive apps," in *Journal of Systems and Software*, vol. 120, 2016, pp. 70-99.

[37] C. Csallner, N. Tillmann, and Y. Smaragdakis, "DySy: Dynamic symbolic execution for invariant inference," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE'08, 2008, pp. 281-290.

[38] A. Astorga, S. Srisakaokul, X. Xiao, and Tao Xie, "PreInfer: Automatic inference of preconditions via symbolic analysis", in *Proceedings of the 48th IEEE/IFIP International Conference on Dependable Systems and Networks*, ser. DSN'18, 2018, pp. 678-689.

[39] L. Zhang, G. Yang, N. Rungta, S. Person, and S. Khurshid, "Feedback-driven dynamic invariant discovery," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser ISSTA'14, 2014, pp. 362-372.

[40] A. Carzaniga, A. Goffi, A. Gorla, A. Mattavelli, and Mauro Pezze, "Cross-checking oracles from intrinsic software redundancy," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE'14, pp. 931-942.

[41] W. Zheng, M. Lyu, and T. Xie, "Test selection for result inspection via mining predicate rules," in *Proceedings of the 31th International Conference on Software Engineering*, ser. ICSE'09, 2009, pp. 215-225.

[42] O. Raz, P. Koopman, and M. Shaw, "Semantic anomaly detection in online data sources," in *Proceedings of the 24th International Conference on Software Engineering*, ser ICSE'02, 2002, pp. 302-312.

[43] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki, "Where do configuration constraints stem from an extraction approach and an empirical study," in *IEEE Transactions on Software Engineering*, vol. 41, no. 8, 2015, pp. 820-841.

[44] K. Xu, K. Tian, D. Yao, and B. G. Ryder, "A sharper sense of self: probabilistic reasoning of program behaviors for anomaly detection with context sensitivity," in *Proceedings of the 46th IEEE/IFIP International Conference on Dependable Systems and Networks*, ser DSN'16, 2016, pp. 467-478.

[45] L. Mariani, M. Pezze, and M. Santoro, "Gk-tail+ an efficient approach to learn software models," in *IEEE Transactions on Software Engineering*, vol. 43, no. 8, 2017, pp. 715-738.

Yi Qin received his doctoral degree in computer science and technology from Nanjing University, China. He is an assistant professor with the State Key Laboratory for Novel Software Technology and Department of Computer Science and Technology at Nanjing University. His research interests include software testing and adaptive software systems.

Tao Xie received his doctoral degree in computer science from the University of Washington at Seattle, USA. He is with the Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education and the Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois. He was the ISSTA 2015 Program Committee Chair and will be an ICSE 2021 Program Committee Co-Chair. He is a co-Editor-in-Chief of the Software Testing, Verification and Reliability (STVR) journal. He has been an Associate Editor of the IEEE Transactions on Software Engineering (TSE) and the ACM Transactions on Internet Technology (TOIT), along with an Editorial Board Member of Communications of ACM (CACM). His research interests are in software engineering. He is an ACM Distinguished Scientist and an IEEE Fellow.

Chang Xu received his doctoral degree in computer science and engineering from The Hong Kong University of Science and Technology, Hong Kong, China. He is a full professor with the State Key Laboratory for Novel Software Technology and Department of Computer Science and Technology at Nanjing University. He participates actively in program and organizing committees of major international software engineering conferences. He co-chaired the MIDDLEWARE 2013 Doctoral Symposium, FSE 2014 SEES Symposium, and COMPSAC 2017 SETA Symposium. His research interests include big data software engineering, intelligent software testing and analysis, and adaptive and autonomous software systems. He is a senior member of the IEEE and member of the ACM.

Angello Astorga received his bachelor degree in Computer Science and Engineering with Magna Cum Laude Honors from The Ohio State University, USA. He is currently working toward his doctoral degree at the Department of Computer Science at the University of Illinois at Urbana-Champaign, USA. His research interests include software testing, machine learning, and program synthesis.

Jian Lu received his doctoral degree in computer science and technology from Nanjing University, China. He is the Director of the State Key Laboratory for Novel Software Technology. He is a full professor with the Department of Computer Science and Technology at Nanjing University. He has served as a Vice Chairman of the China Computer Federation since 2011. His research interests include software methodologies, automated software engineering, software agents, and middleware systems.