Programming of Automation Configuration in Smart Home Systems: Challenges and Opportunities

SHEIK MURAD HASSAN ANIK*, Virginia Tech, United States and Auburn University at Montgomery, United States
XINGHUA GAO, Virginia Tech, United States
HAO ZHONG, Shanghai Jiao Tong University, China
XIAOYIN WANG, University of Texas at San Antonio, United States
NA MENG, Virginia Tech, United States

As the innovation of smart devices and internet-of-things (IoT), smart homes have become prevalent. People tend to transform residences into smart homes by customizing off-the-shelf smart home platforms, instead of creating IoT systems from scratch. Among the alternatives, Home Assistant (HA) is one of the most popular platforms. It allows programmers (i.e., home residents or smart-home creators) to smartify homes by (S1) integrating selected devices into the system, and (S2) programming YAML-based software to control those devices. Unfortunately, due to the diversity of devices and complexity of automatic configurations, many programmers have difficulty correctly creating YAML files. Consequently, their smart homes may not work as expected, causing frustration and concern in people.

This paper presents a novel study on issues of YAML-based automation configuration in smart homes (issues related to S2). We mined the online forum Home Assistant Community for discussion threads related to programming of automation configuration. By manually inspecting 190 threads, we revealed 3 categories of concerns: implementation, optimization, and debugging. Under each category, we classified discussions based on the issue locations and technical concepts involved. Among debugging discussions, we further classified discussions based on users' resolution strategies; we also applied existing analysis tools to buggy YAML files, to assess the tool effectiveness. Our study reveals the common challenges faced by programmers and frequently applied resolution strategies. There are 129 (68%) examined issues concerning debugging, but existing tools can detect at most 14 of the issues and fix none. It implies that existing tools provide limited assistance in automation configuration. Our research sheds light on future directions in smart home programming.

CCS Concepts: • **General and reference** → **Empirical studies**; • **Computer systems organization** → *Embedded and cyber-physical systems*; • **Software and its engineering** → *Software creation and management*.

Additional Key Words and Phrases: Empirical, smart home, automation, programming, YAML debugging

*The majority of the work for this paper was completed at Virginia Tech.

Authors' Contact Information: Sheik Murad Hassan Anik, murad@vt.edu, Virginia Tech, Blacksburg, Virginia, United States and Auburn University at Montgomery, Montgomery, Albama, United States; Xinghua Gao, xinghua@vt.edu, Virginia Tech, Blacksburg, Virginia, United States; Hao Zhong, zhonghao@sjtu.edu.cn, Shanghai Jiao Tong University, Shanghai, China; Xiaoyin Wang, xiaoyin.wang@utsa.edu, University of Texas at San Antonio, San Antonio, Texas, United States; Na Meng, nm8247@vt.edu, Virginia Tech, Blacksburg, Virginia, United States.

Manuscript submitted to ACM

ACM Reference Format:
Sheik Murad Hassan Anik, Xinghua Gao, Hao Zhong, Xiaoyin Wang, and Na Meng. 2018. Programming of Automation Configuration in Smart Home Systems: Challenges and Opportunities. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 24 pages. https://doi.org/10.1145/1122445.1122456

1 Introduction

A **smart home** is a residence that uses internet-connected devices to remotely monitor and manage appliances/systems. According to Fortune Business Insights, the global Smart Home Market size is projected to reach USD 338.28 billion by 2030, at a Compound Annual Growth Rate (CAGR) of 20.1% during the forecast period 2023–2030 [49]. As explained by researchers, the increasing number of internet users, surging disposable income of consumers within emerging economies, the growing significance of home monitoring in remote areas, and the increasing demand for low-carbon emission and energy-saving-oriented solutions drive the market competencies [36].

A **smart home platform** is a software framework that controls and manages multiple devices from multiple manufacturers, usually through a smartphone or tablet app. Various smart home platforms are available: some are commercial systems and close-source (e.g., Samsung SmartThings [66]); some are free and open-source (e.g., openHAB [64]). *Among the alternatives, Home Assistant (HA) has become one of the most widely used platforms* [37–39] mainly because it is free, open-source, and designed specially for local control as well as privacy. Up till March 2024, HA has got over 332 thousand active installations; HA creators estimated the actual number of HA users (e.g., residents or creators of HA-based smart homes) to be 3 times of this number (i.e., about 1 million) [45]. The widespread usage of HA motivated us to do software engineering research on HA-based smart homes, because (1) the program and data from these systems can represent many smart homes, and (2) our research findings will help shape the future of smart home programming.

To create a smart home with HA, programmers need to (1) integrate components (e.g., devices) into the system and (2) create YAML files to automatically control those components. YAML [50] is a human-friendly data serialization language for all programming languages. With YAML, HA users or programmers of smart homes can define an automation rule by specifying a **trigger**, an **action**, and (optionally) a **condition**; such a rule expresses that HA performs an action when a trigger event occurs and optionally the specified condition is met. For instance, the rule in Fig. 1 means to turn on office lights, when someone moves (i.e., the sensor detects motion) and the outside is dark (i.e., the solar elevation angle is less than four degrees).

With some initial inspection of the online forum Home Assistant Community [46], we found lots of questions concerning automation configuration. On the forum, over 10,000 questions were tagged with "automation" and over 1,000 questions were tagged with "configuration" [48]. This may be because many programmers have insufficient domain knowledge and automatic control is challenging. As a result, wrongly programmed smart homes may misbehave [34], waste energy [4], frustrate users [35], or jeopardize home safety.

Inspired by our initial observations, we conducted a novel empirical study on automation configuration issues in HA-based smart homes, to understand the technical challenges and identify research opportunities. Specifically, we crawled the discussion threads of "Configuration" during 03/2022–08/2022 using tag "automation", and retrieved 438 potentially relevant threads. Then we manually analyzed all threads to identify the root cause and resolution strategy of each issue under discussion, and filtered out 248 threads because they lack necessary information. We explored the following research questions (RQs) and observed interesting phenomena:

- **RQ1:** *What challenges do developers face when programming automation configuration?* Among the 190 threads, 129, 52, and 9 discussions separately focus on debugging, implementation, and optimization. It means that

```yaml
automation:
  - alias: "Turn on office lights"
    trigger:
      - platform: state
        entity_id: sensor.office_motion_sensor
        to: "on"
    condition: "{{ state_attr('sun.sun', 'elevation') < 4 }}"
    action:
      - service: scene.turn_on
        target:
          entity_id: scene.office_lights
```

Fig. 1. An exemplar YAML file for home automation [42]

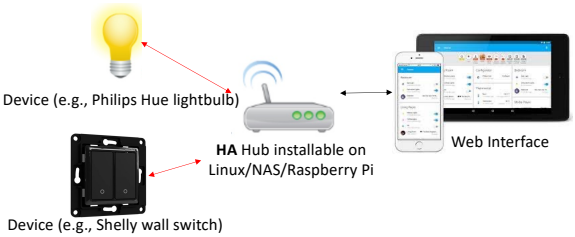

Fig. 2. The consumer pattern in Home Assistant

developers get stuck with debugging more often than other issues. We also observed significant concept commonality/similarity among threads (e.g., data specification for matching), which implies developers' strong need for help or tool support in implementing, debugging, or optimizing some features.

- **RQ2:** *How do developers address challenges in automation configuration?* Developers frequently applied eight strategies to address common challenges. Two of the strategies correct formats (i.e., quotes and indentation); two conditionally call various services; one replaces the trigger type; one correctly accesses or calculates data; one correctly specifies data for matching; one handles a group/list of same-typed entities. All strategies imply desired tool support.
- **RQ3:** *How effectively do existing tools detect or fix buggy YAML files?* We searched online for YAML file validators. By applying the 6 publicly available tools to 129 buggy files, we observed all tools to report only 16–20 files/snippets as invalid ones, and 1–5 of these reports are false positives. All tools achieved high precision, very low recall, and very low F-score when detecting bugs; their error-reporting mechanism is poor. No tool fixes bugs.

In this paper, we made the following research contributions:

- We conducted a novel and comprehensive empirical study to characterize programming issues of automation configuration in HA-based smart homes. No prior work did that.
- We revealed the common root causes and recurring resolution strategies for HA-related automation issues; many of our findings are revealed for the first time.
- We novelly applied 6 publicly available tools to 129 buggy YAML files/snippets, and surprisingly found all tools ineffective in detecting or fixing bugs.

HA is one of the most widely used smart home platforms, and commonly shares the automation configuration mechanism (if-this-then-that rules) with many other platforms (e.g., IFTTT [47]). Thus, our study will shed light on future research in smart homes, help with programming or software engineering in smart home development, and enlighten potential ways of improving smart-home quality. Our dataset is available at https://figshare.com/s/7aa8ea9f4af98c371114

2 Background

This section introduces terms relevant to Home Assistant.

HA can be installed on various devices, from full Linux systems to some network-attached storage (NAS) environment or even a Raspberry Pi. As shown in Fig. 2, users can access HA through a dashboard or web-based user interface by using companion apps for smart phones, or by using web browsers for tablets and PCs. Once the HA software is installed, it acts as a hub—a central control system for home automation. Without any mandatory dependence on vendor-specific cloud services, devices, or mobile apps, the HA hub can have local control of the IoT devices, software, applications, and services that are supported by modular integration components. **Home Assistant Community**

(HAC) [46] is an online forum dedicated for HA users and programmers to discuss as well as resolve issues in smart home systems. HAC contains discussion threads. Each thread has one question post, and zero or more answer posts; at most one answer in each thread is the accepted answer.

Integrations are pieces of software that allow HA to connect to other software and platforms. For example, a product by Philips called Hue smart light (see Fig. 2) can be included into a smart home via the Philips Hue integration, which integration allows HA to talk to the hardware controller Hue Bridge, so that any HA-compatible physical devices connected to Hue Bridge appear in HA as logical devices (i.e, virtual objects) and can be controlled by the hub.

Entities are the basic building blocks to hold data in HA. An entity represents a sensor, actor, or function in HA, which can monitor physical properties or control other entities. Each entity has a **state** to hold information of interest (e.g., whether a light is on or off); an entity's state only holds one value at a time. Entities can store **attributes** related to its state, such as the brightness of a turned-on light. **Sensors** return information about an object, such as the level of water in a tank. **Devices** are logical groups for entities. A device may represent a physical device with one or more sensors; the sensors are entities associated with the device.

Automation is a set of repeatable actions that can be run automatically. **Automation Configuration** is about programming YAML files to specify automations, after HA is installed and all components are integrated. In a YAML file, an automation rule has three key segments:

(i) Trigger describes what starts an automation [43]. An automation can be triggered by an event (i.e., signal emitted when something happens), certain entity state (e.g., when a light is on), or a given time. Multiple triggers can be specified simultaneously for one automation. When any of the automation's triggers becomes true (i.e, trigger fires), HA will validate the conditions if any, and call the action. For the example in Fig. 1, the trigger is an event: when the state of a motion sensor is changed to "`on`".

(ii) Condition is optional; it describes the **predicates** (i.e., tests) that must be met before actions get run [42]. After a trigger occurs, all conditions are checked. The action is run when conditions are satisfied. The condition in Fig. 1 tests whether the solar elevation angle is less than four degrees.

(iii) Action describes what is executed when a rule fires [41]. It can interact with anything via services or events. **Service** carries out a task, such as turning on the light in the living room. A service can have a target and data. For instance, entity `scene.office_lights` in Fig. 1 is a **scene** that prescribes a series of actions, with each action setting an entity's state. The defined action interacts with this scene by calling service `scene.turn_on`, to turn on lights and set their states as prescribed by the scene.

Scripts are also repeatable actions, similar to automations. The difference is that scripts do not have triggers: scripts cannot automatically run unless they are used in an automation.

Templates are used for formatting outgoing messages to present to users or processing incoming data from entities. They are expressed with Jinja2 [19]—a general-purpose templating language. For instance, `"{{state_attr('sun.sun', 'elevation') < 4}}"` in Fig. 1 is a template, which gets the solar elevation degree from entity `sun.sun` and compares that value with 4 to define a predicate for condition checking. Templates enable smart homes to flexibly respond to varying entity states. To define templates, people must surround single-line templates with double quotes (") or single quotes ('). When defining multi-line templates, people must use `{% %}` to enclose the program structures (e.g., loop) in templates.

3 Methodology

To understand the challenges and opportunities in the programming of home automation, we explored the following three research questions:

- **RQ1:** *What challenges do developers face when programming automation configuration?* What kind of automation-related questions do programmers ask? Are there questions frequently asked? What are the root causes of the frequently asked questions?
- **RQ2:** *How do developers address challenges in automation configuration?* Namely, when people ask similar or identical questions, is there any answer repetitively suggested? Do answers present resolution strategies?
- **RQ3:** *How effectively do existing tools detect or fix buggy YAML files?* When programmers have difficulty debugging erroneous YAML files, can existing tools reveal or fix those errors?

This section first introduces our procedure of data collection, and then explains our method of exploring RQs.

3.1 Data Collection

The program data related to HA-based smart home systems can be majorly found on two websites: GitHub [44] and HAC [46]. We decided to crawl HAC for two reasons. First, HAC organizes discussion threads based on categories and tags. Such an organization enables us to quickly locate automation configuration-related program data. Second, many questions on HAC are resolved, with some resolutions well explained and finally accepted by askers. Such a high availability of technical solutions and rationale explanation enables us to rigorously characterize questions and answers.

Specifically, we manually went through the discussion threads under category "Configuration" of HAC, to locate candidates tagged with "automation" during 03/01/2022 - 08/31/2022. That six-month period was chosen because we collected data in September 2022. As HAC ranked threads based on timestamps of the latest activity on each thread (e.g., question posting or answer updating), we successfully retrieved a dataset of 438 candidate discussion threads purely based on the HAC ranking and timestamp range. Next, we manually inspected each thread to record (1) ID of the question post, (2) the URL, (3) the original YAML file provided by the asker, and (4) the suggested YAML file mentioned, referred to, or implied by the accepted answer. In this process, we filtered out a thread if

(i) the question asker does not present a YAML file,
(ii) no answer was explicitly marked as accepted by the asker,
(iii) the accepted answer does not suggest or lead to any concrete version of YAML file, or
(iv) the discussion is too confusing for us to understand.

We defined the four filters mentioned above, to ensure the high quality of our data analysis results. With more details, based on our experience so far, when people asked questions at HAC, they sometimes failed to provide YAML files to clearly show the programming context or automation scenarios. Consequently, analyzing such questions can be very challenging and time-consuming, as it is hard to tell what is the major issue and what types of resolutions askers inquiry for. To avoid such confusion, we introduced filter (i) to guarantee that each covered question post provides sufficient details on the programming context, including but not limited to askers' automation needs, the attempts they made, the technical issues they face, and their specific requests for help. When processing each thread, we manually inspected all posts involved in the discussion. As long as the asker provided their YAML files in any of the posts, we kept those threads.

Filter (ii) ensures that there is a correct answer for us to refer to, when characterizing people's concern and resolution for each covered thread. Filter (iii) makes sure that the suggested resolution is clear and concrete. Namely, we do not have to speculate on the correct version of YAML files, and thus will not commit mistakes in speculation. In reality, however, an asker might accept an answer that provides no concrete YAML files, while the concrete version was mentioned in a different post of the same thread. To tolerate such inaccuracy in askers' labeling of accepted answers,

we read all posts in each thread. As long as a concrete YAML file is suggested by an accepted answer or any answer related to that one, we counted in the thread. In this way, we made our dataset as representative as possible. Filter (iv) ensures that we have high confidence in our interpretation of threads. After applying all filters mentioned above, we kept 190 threads for further analysis.

3.2 Data Analysis

For RQ1, we took an open-coding approach to classify threads, because we had no prior knowledge of people's concerns on automation configuration. Specifically, two authors manually inspected all threads to identify keywords/phrases, which characterize each thread in terms of the (1) issue type, (2) automation component under discussion, and (3) involved technical concepts. Then they separately defined an initial taxonomy and categories by clustering recurring or similar keywords/phrases. Next, they held a meeting to compare and discuss their initial results to refine the taxonomy and improve thread classification. Using the classification labels they agreed upon, author A rechecked all threads to label categories and to reveal overlooked or wrong categories. Afterwards, author B examined A's labels for all threads; whenever he disagreed upon any labels, he had discussions with author A until reaching a consensus. This procedure sometimes involved multiple iterations of thread labeling, and lasted until both authors agreed upon all labels.

For RQ2, we also took an open-coding approach to identify frequently adopted resolution strategies. With more details, inside each category, author A summarized the resolution suggestion for each issue-under-discussion, and then clustered issues if their resolutions are identical or similar in certain aspect. In this procedure, A also defined resolution strategies based on the common summaries. Next, author B manually inspected all strategies as well as related threads, and initiated discussion with A for any disagreement. The discussion lasted until all disagreements were addressed.

For RQ3, we first defined a ground-truth dataset of buggy YAML files/snippets, according to the classification results in RQ1. In this dataset, we also included the fixes suggested by accepted answers. Next, we applied all state-of-the-art YAML validation tools to the dataset, to study how effectively existing tools detect and fix bugs.

4 Experiment Results

This section presents and explains our results for RQ1–RQ3.

4.1 RQ1: Challenges in Handling Automation Configuration

Fig. 3 shows our thread classification based on issue types, issue locations, and involved technical concepts.

4.1.1 Classification Based on Issue Types. We observed three types of technical issues: (1) debugging, (2) implementation, and (3) optimization. Debugging involves scenarios where users share erroneous YAML files; they request help in diagnosing the root cause of the errors and fixing the issues. Implementation refers to cases where users describe their program context and automation requirements, seeking guidance to create YAML-based configurations that fit their specific scenarios and achieve desired functionality. These discussions typically focus on translating requirements into working code from scratch or extending existing automation configurations to include new features. The developer queries for implementation tasks typically include "How to do something", such as "*How to exclude entities in automation?*" [33].

Debugging questions differ from implementation as they often include failed attempts by the user to achieve certain automation goals, with buggy YAML files as a starting point. The focus here is not on how to implement a requirement from the ground up but rather on identifying and resolving the mistakes in existing configurations. The

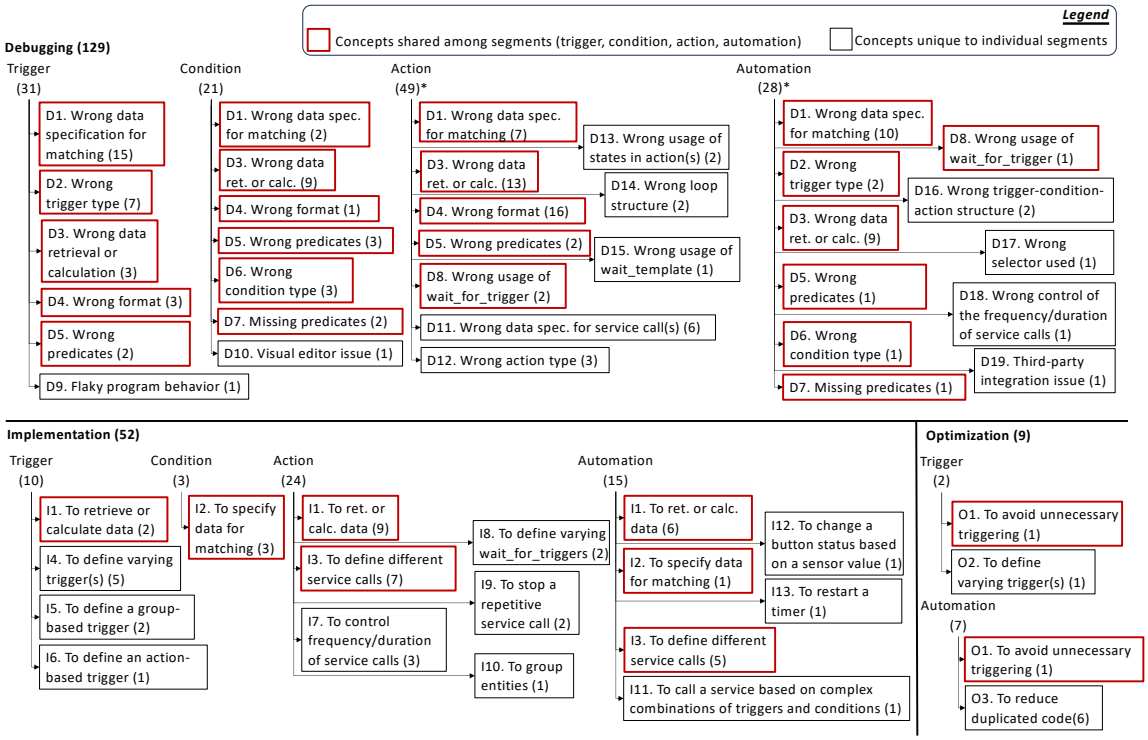

Fig. 3. The taxonomy of 190 threads, where * means some issues belong to more than one category

debugging queries typically state that the user is trying to do something, however it is not functioning as expected. One representative example is "*Please help me fix (long) automation*" [26].

Optimization discussions occur when users present YAML files that are already functional and meet specific requirements but seek advice on improving the efficiency, readability, or maintainability of the configuration. Unlike debugging or implementation, these questions aim to refine the automation programs further, addressing concerns such as performance improvement, redundancy reduction,, or compatibility enhancement. For example, the following is an optimization query: "*I'm trying to figure out if is it possible to combine some of my automations to simplify things*" [12].

Our dataset includes 129 debugging issues, 52 implementation issues, and 9 optimization issues. It means that most askers had difficulty debugging YAML files, while fewer askers were bothered by implementation and optimization issues. This may be because automation configurations have relatively simple control logic. It is not quite difficult for developers to get started and have initial implementation done. However, it is harder for programmers to successfully debug initial implementation, in order to deliver high-quality automation configurations. Consequently, even fewer programmers bother to optimize correct configurations for refinement.

Finding 1: *68% (129/190) of the examined issues are about debugging, implying the significant challenge of addressing buggy YAML files.*

4.1.2 Classification Based on Segments Involved. Under each issue category, we classified threads based on where issues occurred. Namely, if an issue resided in purely the trigger, condition, or action segment of an automation rule, we labeled it accordingly. Otherwise, if an issue resided in multiple segments or we could not clearly tell which specific segment an issue was about, then we labeled the issue with "Automation".

As shown in Fig. 3, for both debugging and implementation issues, "Action" is the largest among four categories. This implies that actions are harder to develop than triggers and conditions. One reason is that HA defines the script syntax [31], which allows people to define complex control structures (e.g., if-then and loop), or special behaviors in actions (e.g., `wait_for_trigger` to wait for a trigger before calling a service). The syntax poses challenges for people to understand and properly use all syntactic components. Additionally, there are more threads on triggers than conditions. It implies that triggers are harder to develop, probably because there are more alternative ways of defining triggers, and more delicate constraints on trigger definition. Finally, automation issues widely exist in all three categories. It means that programmers have difficulty in developing multiple segments for consistency, or in creating code that can exist in any of the segments.

Finding 2: *There are 73, 50, 43, and 24 issues separately about action, automation, trigger, and condition, implying that automation and action segments are more challenging to create or improve.*

4.1.3 Classification Based on Technical Concepts. We further classified issues based on the technical concepts involved. In this procedure, whenever we observed similar issues in different segments (i.e., trigger, condition, or action), we used the same terms to capture the similarity, and to identify segment-specific concepts as well as segment-agnostic ones. In Fig. 3, **the 129 debugging issues cover 19 unique concepts; 8 concepts are shared among segments (D1–D8)**.

D1. Wrong data specification for matching means that data (e.g., state or attribute) was wrongly provided for predicate-value matching in trigger, condition, action, or the whole automation, making the automation run abnormally. For instance, lines 2–6 in Listing 1 show a buggy snippet from a debugging thread [9]. This snippet defines a state trigger, to start an automation when an entity `input_boolean.ecobee_fan_on_off` changes from the `'off'` state to `'on'`. However, as both states are misspelled, HA is case-sensitive and does not recognize those states. Consequently, the automation is never started because the prescribed state transition never happens.

D2. Wrong trigger type: There are multiple types of triggers (e.g., state trigger and device trigger) usable to specify the triggering logic of an automation. However, when developers wrongly choose a trigger type, the specified logic does not work because the chosen type does not support that logic. For instance, the buggy version in Listing 3 defines a device trigger, to fire when sensor `binary_sensor.v4b01_motion` detects no motion for the number of minutes developers specified via `input_number.timeout_offices`, a numeric input box in GUI. However, device triggers do not support templating (see line 11), so the buggy version fails.

D3. Wrong data retrieval or calculation means that some data used in an automation is wrongly accessed or calculated, causing automation fail to run normally. For instance, lines 2–6 in Listing 2 show a buggy snippet from a debugging thread [23], which defines an action to send a notification message via mobile phone. This action calls the service `notify.mobile_app_huawei_p20`, with a dynamically generated message to incorporate the reading of a temperature sensor `sensor.temperatur_gefrierschrank`. However, as developers wrongly accessed the sensor value via `sensor.temperatur_gefrierschrank`, the automation does not run and an error message is produced: *"Error rendering data template: UndefinedError: 'sensor ist undefined"*.

D4. Wrong format means that a YAML file is malformed, by violating the formatting rules defined by either YAML or HA. Typical violations include wrong line indentation and wrong quote usage.

D5. Wrong predicates means automation fails because the predicates used in automation have flaws: the defined predicates or predicate combinations do not enable action execution as expected. For instance, Listing 4 shows a buggy condition that combines two predicates `before:sunrise` and `after:sunset`. The former predicate corresponds to the time period between midnight and sunrise, while the latter corresponds to the period between sunset and midnight. These

```yaml
# Bug: Wrong states are specified for the to: and from: options
trigger:
- platform: state
  entity_id: input_boolean.ecobee_fan_on_off
  to: 'On'     # Fix: 'On' should be 'on'
  from: 'Off' # Fix: 'Off' should be 'off'
# Implementation: To properly specify an unwanted target state with the not_to: option
trigger:
- platform: state
entity_id:
- sensor.pc1_printjob
- sensor.pc2_printjob
- sensor.pc3_printjob
- sensor.pc4_printjob
not_to: 'unknown'
```

Listing 1. Two YAML snippets to show the common challenge of "data specification for matching" [9, 13]

```yaml
# Bug: Wrong way of retrieving the sensor state value
action:
- service: notify.mobile_app_huawei_p20
  data:
    title: Temperature Warning
    message: 'Temperature is: {{ sensor.temperatur_gefrierschrank }}'
# Fix: sensor.temperatur_gefrierschrank => states("sensor.temperatur_gefrierschrank")
# Implementation: To include sensor value into the notification
action:
  - device_id: c593fadfd2c2d134bc507567b588b2ae
    domain: mobile_app
    type: notify
    message: Energy production ended. Energy produced today: {{states('sensor.solaredge_current_power')}}
```

Listing 2. Two YAML snippets to show the common challenge of "data retrieval or calculation" [23, 28]

```yaml
# Bug: A template is involved in a device trigger definition
trigger:
  - type: no_motion
    platform: device
    device_id: e0954ea41d7a6da69baeff2e9558ed13
    entity_id: binary_sensor.v4b01_motion
    domain: binary_sensor
    id: '1'
    for:
      hours: 0
      minutes: ''{{states('input_number.timeout_offices') |int(0) }}''
      seconds: 0
# Fix: Replace the device trigger with a state trigger to support template usage
trigger:
  - platform: state
    entity_id: binary_sensor.v4b01_motion
    id: '1'
    to: 'off'
    for:
      minutes: "{{ states('input_number.timeout_offices') | int(0) }}"
```

Listing 3. A bug and fix related to D2 [11]

```yaml
# Bug: Wrong combination of predicates
condition:
  - condition: sun
    before: sunrise
    after: sunset
```

Listing 4. A buggy snippet of "wrong predicates" (D5) [20]

```yaml
condition:
  - condition: template
    value_template: '{{ trigger.from_state.state | is_number }}'
```

Listing 5. A condition injected to address D7 [24]

two predicates should never get used together to define a period "*after the sunset of Day N and before the sunrise of Day (N+1)*", as the two predicates are always interpreted as two separate periods: "*after the sunset of Day N and before the sunrise of Day N*". No time point satisfies both predicates at the same time.

D6. Wrong condition type is similar to D2. There are multiple types of conditions. However, using some condition type (e.g., device condition) can limit automation expressiveness (e.g., no template allowed), making some logic infeasible. Namely, this category of errors are semantic errors, where condition types are wrongly used to express logic that they do not support.

D7. Missing predicates means one or more predicates are not checked before an automation executes certain action. For example, an automation is defined to send a notification message on the phone when the washing machine has finished a job [25]. The automation has a trigger defined as "when power consumption of the washer is below 0.1 watts for more than 2 minutes", but has no condition defined. Consequently, the automation gets triggered again and again, sending repeated notifications every few minutes after the power consumption is <0.1 watts. The major reason is reading interruptions: there are some short moments without reading, after which, when everything comes back to normal, the automation re-triggers. To eliminate re-triggers caused by intermittent sensor readings, developers are recommended to insert a condition (see Listing 5), which ensures that the trigger is fired by a transition from a state with numeric reading, instead of from other states (i.e., states without any reading).

D8. Wrong usage of `wait_for_trigger`: Action `wait_for_trigger` allows automation to wait for a trigger being fired before doing anything. Some developers messed it up with (1) another specialized action `wait_template`, which waits for a

```yaml
alias: Routine - Sluit ochtend af
sequence:
# Bug: wait_for_trigger is used
  - wait_for_trigger:
      - platform: template
        value_template: >-
          {{ states('sensor.buitensensor_achter_illuminance') | float >
          states('input_number.lichtdrempel') | float }}
# Fix: wait_template should be used
  - wait_template: "{{ states('sensor.buitensensor_achter_illuminance') | float(0) >
↪  states('input_number.lichtdrempel') | float(0) }}"

    timeout: '3600'
  - service: scene.turn_on
    data: {}
    target:
      entity_id: scene.alle_lichten_uit
```

Listing 6. A bug and fix related to D8 [40]

```yaml
# Bug: A service call with wrongly provided data
action:
  - service: tts.google_say
    data:
      entity_id: media_player.living_room_speaker
      message: Dryer has started
      cache: true
      cache_dir: /tmp/tts
```

Listing 7. A buggy snippet of D11 [21]

```yaml
alias: Boiler off after 30 minutes
description: ""
trigger:
  - platform: state
    entity_id:
      - input_boolean.boiler_manual
    to: "on"
# The unsuccessful configuration
    for:
      hours: 0
      minutes: 30
      seconds: 0
# The successful configuration
    for: "00:30:00"
```

Listing 8. An instance of D9 [27]

```yaml
 # Bug: A state is wrongly used in action
action:
  - platform: state
    entity_id: switch.valve_2
    state: 'off'
# Fix: The action should call a service to turn
↪  off the switch and change the state
action:
  - service: switch.turn_off
    entity_id: switch.valve_2
```

Listing 9. An example of D13 [7]

template to evaluate to true before running automation, or (2) the trigger segment. As shown in Listing 6, a buggy morning automation leverages `wait_for_trigger` to express that the script should wait until the light (i.e., sensor reading of `'sensor.buitensensor_achter_illuminance'`) passes a specific value stored in a variable (i.e., `states('input_number.lichtdrempel')`). However, a correct automation should use `wait_template`. This is because `wait_for_trigger` monitors for a state transition that turns the template evaluation from false to true, while `wait_for_trigger` waits for the template to evaluate to true. The two actions work differently when the initial template evaluation is true: `wait_template` immediately continues with the true value, while `wait_for_trigger` waits until an evaluation transition from false to true.

D9. Flaky program behavior means two semantically equivalent implementations of the same logic have divergent outcomes: one automation succeeds and the other fails. As shown in Listing 8, the automation is expected to be triggered 30 minutes after a boolean variable `input_boolean.boiler_manual` is set to "on". Both of the 30-minute configurations are legitimate. However, developers observed the second one to work perfectly, while the first one does not. This may be due to issues inside HA.

D10. Visual editor issue means a problematic YAML file is generated due to the usage of a visual editor in HA IDE. This may be due to issues in the IDE implementation.

D11. Wrong data specification for service calls means an action calls service(s) using wrong data, making the entire automation fail. Namely, if we treat service calls as analogous to method calls in Java programming, then wrongly specified service data is analogous to wrongly provided parameter values. For instance, Listing 7 shows a service call, which intends to make the Google speaker announce an audio message ``Dryer has started''. However, lines 7–8 are not needed by the service call. When both key-value pairs are provided, the automation fails and an error message is generated: *extra keys not allowed @data['cache_dir']*.

D12. Wrong action type is similar to D2 and D6. It happens when developers wrongly select a type to implement some unsupported logic.

D13. Wrong usage of states in action(s) means developers try to define an action by specifying entity states. However, action can be only defined via interactions with services or events (e.g., service calls). As shown in Listing 9, an action

```
# Bug: The action field in automation does not support templating
{% for entityId in area_entities("living_room") if entityId.startswith("light.") %}
    ... ...
    - service: light.turn_on
    data:
      color_temp: ... ...
    target:
      entity_id: {{ entityId }}
  {% endfor %}
# Fix: Use the repeat action to repetitively apply a sequence of actions to a group of
↪ entities
repeat:
  for_each: |
    {{ area_entities(trigger.event.data.area) }}
  sequence:
    - condition: template
      value_template: '{{ repeat.item.startswith("light.") }}'
    - service: light.turn_on
      data:
        color_temp: ... ...
      target:
        entity_id: '{{ repeat.item }}'
```

Listing 10. A bug and fix related to D14 [29]

```
# Bug: two wait_templates are specified while
↪ only one is needed
- wait_template: '      - wait_template: ''{{
↪ is_state(...) or is_state(...) or
↪ is_state(...) }}'''
# Fix: remove the unnecessary wait_template
- wait_template: |-
    {{ is_state(...) or is_state(...) or
↪ is_state(...) }}
```

Listing 11. An example of D15 [10]

```
alias: Push - Reminder to Arm Home
description: ''
# Bug: No trigger is specified
trigger: []
condition: []
action:
  - service: notify.notify
    data:
      message: Alarm System has not been armed.
```

Listing 12. A bug of D16 [22]

intends to close a valve `switch.valve_2`. However, `platform: state` is for triggers; when it is put in action to execute or update a state, the action does not work. The action should instead call service `switch.turn_off` to update entity states.

D14. Wrong loop structure means developers wrongly use the loop construct to define a malformed automation. For example, as shown in Listing 10, an automation tries to turn on all lights in an area at once (i.e., `light.turn_on`). As different light models have different ranges of temperatures, the automation attempts to set a valid temperature to each light (i.e., `color_temp`) depending on the ranges and lighting requirements. The buggy automation wrongly uses the `for`-loop structure of templates to enumerate and operate each light, and the usage leads to a syntactic error *Message malformed: expected dictionary @ data['action'][0]*. This is because the action field does not support templating. Alternatively, a `repeat` action should be used in the action field to define the loop structure, with the `for-each` form adopted to iterate over items.

D15. Wrong usage of `wait_template` means developers incorrectly use this action. For instance, Listing 11 shows a wrong usage of `wait_template`, where the action is specified twice. After developers removed one `wait_template` and slightly adjusted the related content, the automation works well.

D16. Wrong trigger-condition-action structure means developers wrongly define the trigger-condition-action structure to implement an automation, like defining an action without trigger. For instance, a developer wants to define an automation to send a push notification reminder at night, concerning an alarm system. However, as shown in Listing 12, because no trigger is specified in that automation, the notification is not sent as expected (i.e., only at night).

D17. Wrong selector used: Selectors are used to specify what values are acceptable by automation, or define how the input is shown in GUI. D19 is similar to D2, D6, and D12. Basically, when multiple selectors are available, developers did not choose the right one to specify the intended logic.

D18. Wrong control of the frequency/duration of service calls is similar to D2, D6, D12, and D17. There are multiple means to control the frequency or duration of service calls, and these means differ in expressiveness. If developers choose a wrong mean, then they are unable to express the intended logic.

D19. Third-party integration issue means when an automation depends on a third-party software for device control, developers may have difficulty fixing issues due to that software usage.

The 52 implementation issues cover 13 distinct concepts. Three concepts are shared among segments (I1–I3).

```yaml
alias: Turn on patio light at sunset.
description: ' '
trigger:
  - platform: sun
    event: sunset
condition: []
action:
# Requirement: To turn on the lightly differently based on the date
  - service: light.turn_on
    data:
      brightness_pct: 100
      color_name: >
        {% set holidays = { '03-17': 'green', '07-01': 'blue',
                            '10-31': 'orange', '12-25': 'red',
                            '01-01': 'yellow' } %}
        {% set today = (now().date()|string)[5:] %}
        {{ holidays[today] if today in holidays.keys() else 'white' }}
    entity_id: light.outside_2
mode: single
```

Listing 13. An instance of I3 [1]

```yaml
# Requirement: To define a trigger based on varying sensor states
trigger:
  - entity_id: group.all_motions
    platform: template
    value_template: >
      {{ expand('group.all_motions')
         | selectattr('last_changed', 'lt',
now()-timedelta(minutes=15))
         | list | count == 0 }}
```

Listing 14. An example of I4 [3]

```yaml
trigger:
  - platform: state
    entity_id: group.brad_and_lauren
    to: not_home
    for: "00:05:00"
```

Listing 15. An instance of I5 [18]

I1. To retrieve or calculate data is similar to D3, because it also reflects developers' concerns on appropriate data access or calculation. For instance, lines 9–13 in Listing 2 show a correct snippet from an implementation thread [28], which defines an action to send a notification message via mobile phone. The message is formulated based on the value of a sensor that tracks energy production `sensor.solaredge_current_power`. This snippet was recommended because some developers could not retrieve the sensor value. Note that I1 is different from D3, as I1 covers implementation questions while D3 is about debugging questions.

I2. To specify data for matching is similar to D1, as it also reflects developers' concerns on data specification for predicate-value matching. For instance, lines 8–15 in Listing 1 show a snippet from an implementation thread [13]. The snippet defines a state trigger, which monitors four sensors (`sensor.pcX_printjob`), and starts automation when any sensor (1) changes its state and (2) the to-state is not `'unknown'`. This snippet was suggested because some developers could not specify an unwanted state for the trigger.

I3. To define different service calls is about the correct way of calling distinct services based on the fired triggers, satisfied conditions, or collected data. For instance, a developer asks for an automation that turns on a patio light every night at sunset, and changes the default light color if the date is a holiday. To fulfill the requirement of date-based color change, as shown in Listing 13, a suggested automation defines a dictionary `holidays` in template that maps dates to colors. It also defines a variable `today` to save the date of the current day. By checking whether the current date corresponds to any entry in the dictionary, the automation decides whether to use a predefined special color. Based on the determined color, the automation calls service `light.turn_on` differently.

I4. To define varying trigger(s) means to define one or more changeable triggers (i.e., triggers with variables or templates), whose firing events vary with the surrounding environment, time, or inputs. For instance, a developer wants to turn off all lights if all motion sensors detect no movement for at least 15 minutes. To implement the needed trigger that depends on varying states of multiple sensors, Listing 14 defines a template trigger to (1) enumerate each member in a predefined sensor group, (2) examine the attribute `last_changed`, and (3) compare that timestamp with the timestamp 15 minutes before now. If none of the `last_changed` timestamp is later than 15 minutes ago, then the trigger fires because no motion has been detected for at least 15 minutes.

I5. To define a group-based trigger(s) means to define triggers based on the state of a sensor group. For instance, a developer wants to define a triggering event as when both occupants are away from the home. To satisfy this requirement, Listing 15 checks whether a group state is `not_home`, where the group is defined to combines sensors of both people and gets the state `not_home` assigned if both sensors have the state `not_home`.

```yaml
# Requirement: To define a
trigger:
  - platform: event
    event_type: mobile_app_notification_action
    event_data:
      action: "SILENCE"
```

Listing 16. An instance of I6 [16]

```yaml
- service: sonos.join
  data:
    master: media_player.sonos_beam
    entity_id: media_player.keuken_sonos_2
- service: media_player.select_source
  data:
    source: '{{ (state_attr("sensor.sonos_favorites",
↪  "items").values() | list)[3]}}'
  target:
    device_id:
      - 22cd279ccc8cd7ff6463c532ba66932f
```

Listing 17. An instance of I10 [8]

```yaml
repeat:
  until:
    - condition: or
      conditions:
        - condition: state
          entity_id: binary_sensor.water_leak_sensors
          state: 'off'
# Requirement: to stop the repeated alarm if the most recent button press is
↪  within 15 seconds
          - "{{ now() - states('input_button.clear_alarm') | as_datetime <
↪  timedelta(seconds=15) }}"
  sequence:
    - service: media_player.play_media
      data:
        media_content_id: /local/audio/internal_alarm.wav
        media_content_type: music
        enqueue: play
      target:
        entity_id: media_player.house_announcement
    ... ...
```

Listing 18. An example of I9 [30]

I6. To define an action-based trigger means to define a trigger, which fires based on which GUI button users click. For instance, a developer wants to trigger an automation with an action named "SILENCE". Namely, once the triggering action is taken, the automation executes. To define such a trigger, Listing 16 defines an event trigger, which fires when the SILENCE action is received by HA.

I7. To control frequency/duration of service calls is similar to D18, because it also concerns the correct way of controlling the frequency or duration of service calls.

I8. To define varying `wait_for_trigger`*(s)* is similar to D8, as it also focuses on the correct usage of `wait_for_trigger`.

I9. To stop a repetitive service call means when developers call a service in a loop structure, they need help in defining the loop so that the service is called repetitively, and the loop condition is properly evaluated before automation terminates. For instance, a developer wants to implement an action that breaks an alarm loop created via a Repeat Util loop once a button is clicked. To implement that logic, as shown in Listing 18, a template condition is added under `until`, to specify that the most recent button click of `input_button.clear_alarm` is within a short period of time (e.g., 15 second). In this way, loop iterations stop when the condition is satisfied.

I10. To group entities means to group a set of entities (e.g., speakers) so that they work unanimously just like one entity. Specifically, a developer wants to create an automation, which starts a radio stream via Sonos favorites (i.e., a predefined playlist) and plays it on all Sonos speakers in a house. To realize that requirement, Listing 17 groups all speakers with service `sonos.join`, and calls service `media_player.select_source` on of the speakers so that all speakers play the same music unanimously.

I11. To call a service based on complex combinations of triggers and conditions means developers have complex predicates/tests to meet, before a service is called. Thus, they specify their automation needs and all predicates, asking for a well-structured automation that properly puts predicates in triggers and conditions.

Both *I12* and *I13* are self-explanatory, so we skip the explanation for succinctness.

The nine optimization issues cover three unique concepts: to avoid unnecessary trigger firings, to efficiently define changeable triggers, and to reduce duplicated code in automation. **One of the concepts is shared among segments (O1)**.

O1. To avoid unnecessary triggering: These issues arise when an automation is triggered by unintended events, leading to inefficiencies or undesired behaviors. Developers seek ways to refine triggers by incorporating additional predicates or adjusting existing ones to ensure that automations only activate when truly necessary. For example, Listing 19

```yaml
# An unoptimized version, which has one automation to turn on a
↪ light at 3pm exactly, and the other automation to turn off a
↪ light at 5pm exactly
- alias: TurnOnBulb
  trigger:
    - platform: time
      at: "15:00:00"
  action:
    - service: light.turn_on
      data:
        entity_id:
          - light.mybulb

- alias: TurnOffBulb
  trigger:
    - platform: time
      at: "17:00:00"
  action:
    - service: light.turn_off
      data:
        entity_id:
          - light.mybulb
```

Listing 19. An instance of O1 [14]

```yaml
# An optimized version that defines a single automation to turn on
↪ and off a light, according to the predefined time periods
- alias: TurnOnOffBulb
  trigger:
    - id: 'on'
      platform: time
      at: '15:00:00'
    - id: 'off'
      platform: time
      at: '17:00:00'
    - id: 'on'
      platform: state
      entity_id: light.mybulb
      from: 'unavailable'
  condition: "{{ today_at('15:00:00') <= now() <
↪ today_at('17:00:02') }}"
  action:
    - service: 'light.turn_{{ trigger.id }}'
      target:
        entity_id: light.mybulb
```

Listing 20. An optimized version of Listing 19

shows an unoptimized implementation for a simple logic: to turns on a bulb at 3pm and off at 5pm, every day. The implementation has two issues. First, the two automations are similar but only different in certain values, so they are redundant or duplicated. Second, the bulb sometimes goes offline at 2:59pm and goes online at 3:02pm; this automation will not turn on the bulb as expected because it was offline. To turn on the bulb immediately after it got online during 3pm-5pm, a naïve solution can be creating a time pattern trigger, which checks the bulb status during that period once every minute and turns on the bulb if it is still `off`. However, such a frequent status checking is not an elegant solution. To address both issues with an optimized solution, Listing 20 defines three distinct triggers in one automation. The first two triggers separately fire at 3pm and 5pm, getting assigned with ids `'on'` and `'off'`. Action uses both ids to call services `light.turn_on` or `light.turn_off`, depending on the fired trigger. The third trigger checks if a bulb's state is transitioned from `unavailable`, to decide whether the bulb once got offline. If so, the automation further examines the condition predicate, i.e., whether the current timestamp is between 3pm and 5pm. If so, the bulb is also turned on.

O2. To define varying triggers: These issues focus on dynamically adapting triggers based on environmental factors, user preferences, or changing system states. It is similar to I4, but is viewed from an optimization perspective, where developers aim to make automations more flexible and efficient. For instance, instead of defining static triggers for different times of the day, a developer might use templates or variables to adjust trigger conditions dynamically, ensuring that automations respond optimally to real-time changes.

O3. To reduce duplicated code: Developers sometimes define multiple automations that involve repetitive structures, to implement similar but different logic. This concept focuses on automation restructuring to eliminate redundancy. Although in the original question [14], developers only asked for an optimized solution to the bulb's intermittent availability issues; the automation combination shown in Listing 20 demonstrates one way of duplication reduction. Namely, the triggers of similar automations can be combined into a bigger trigger component; the actions of similar automations can also get combined into a bigger action component; each triggering event gets assigned with a unique id, so that the id later get leveraged by action to call the corresponding service.

We use **segment-agnostic concepts** to refer to the concepts shared among distinct segments, and use **segment-specific concepts** to refer to the remaining. The segment-agnostic concepts appear across different segments, regardless of their specific focus area. For example, "D1: Wrong data specification for matching" can be found in any segment of the program (trigger, condition, or action). Fig. 3 shows one (O1), three (I1–I3), and eight (D1–D8) segment-agnostic concepts

found in optimization, implementation, and debugging issues respectively. These imply that, in spite of working in distinct areas, developers tend to ask similar or related questions, reinforcing the idea that certain problem-solving patterns transcend individual segments.

Additionally, across all categories, issues related to trigger cover in total five segment-specific concepts (i.e., D9, I4–I6, O2). Meanwhile, issues related to condition, action, and automation separately cover one, nine, and eight segment-specific concepts. These observations imply that the issues related to trigger and condition are less diverse than those related to action and automation; action and automation are harder to develop and maintain.

Finding 3: *One, three, and eight segment-agnostic concepts are separately identified in optimization, implementation, and debugging issues. This implies the wide existence of common challenges.*

Between the two categories—debugging and implementation, we observed similarity among identified concepts: (i) D1 vs. I2, (ii) D3 vs. I1, (iii) D8 vs. I8, (iv) D18 vs. I7, (v) D11 vs. I3. For instance, both D1 and I2 focus on "data specification for matching"; however, the former describes a root cause of recurring bugs and the latter describes a frequent implementation request.

Between implementation and optimization, we also observed a commonality: to define varying triggers (I4 vs. O2). All such similarity or commonality implies that some concepts are challenging and popular; they confuse people no matter whether people implement, debug, or optimize automations.

To facilitate understanding, we use representative examples to show two common challenges. Listing 1 has a buggy snippet from a debugging thread of D1, and a correct snippet from an implementation thread of I2. In the first snippet, developers committed mistakes by misspelling states (e.g., `'On'` instead of `'on'`), and the automation never started. The second snippet was suggested because some developers could not specify an unwanted state. Both snippets evidence that developers have difficulty in specifying data for matching. Listing 2 shows another buggy snippet and another correct snippet. In the former one, developers made mistakes when accessing a sensor value; the latter one was recommended because some developers asked for help in getting a sensor value. Both snippets imply that developers have difficulty in retrieving or calculating data.

Finding 4: *Five concepts commonly exist in implementing, debugging, and optimization issues; these concepts are challenging no matter whether developers implement, debug, or optimize automations.*

4.2 RQ2: Frequent Resolution Strategies

We realized that the 190 examined issues were resolved in diverse ways; no dominant strategy was applied to resolve the majority. However, we managed to identify some strategies repetitively applied to resolve multiple issues (≥ 4). Table 1 lists each strategy in terms of its focus, the relevant technical concept(s), the number of issues it addressed, and the strategy content. We ranked the eight strategies in ascending order of the number of relevant concepts. This section explains all strategies in detail.

R1 corrects quote usage. It was frequently applied when askers posted YAML files with wrong formats. For instance, Listing 21 shows a representative bug and fix. The buggy snippet tries to set a counter `counter.configure` using the value of another counter `counter.aux_ac_pieza`, but the quote usage violates a domain-specific constraint in HA: when there are quotes inside and outside a template, differentiate the outer and inner ones by using different quote types. The buggy code nests single quotes. Thus, the fix is to replace the inner pair of single quotes with double quotes.

R2 corrects indentation. As with R1, this was also adopted to fix wrong formats. The basics of YAML syntax are block collections and mappings containing key-value pairs. Indentation is important for specifying relations among

Table 1. The representative resolution strategies (i.e., each strategy occurs at least four times in our dataset)

Idx	Focus	The Relevant Technical Concept(s)	# of Issues Resolved	Strategy Content
R1	Quote usage	Wrong format	10	Add or remove quotes to properly use String literals, variables, and templates
R2	Indentation usage	Wrong format	6	Add or remove white spaces to properly align statements
R3	Picking a trigger type	Wrong trigger type	4	Replace the device trigger with a (numeric) state trigger
R4	Reading an entity's state	Wrong data retrieval or calculation, to retrieve or calculate data	9	Use template `{{states('ENTITY')}}` to read the state of ENTITY
R5	Specifying states for matching	Wrong data specification for matching, to specify data for matching	8	Look up all possible states of the entity in Developer Tools -> States, and use one of the states with case sensitivity.
R6	Different service calls based on fired triggers	To reduce duplicated code, to avoid unnecessary triggering, wrong data specification for matching	5	Define multiple triggers, specify a unique ID for each trigger, and call services differently based on the fired trigger.
R7	Different service calls based on entity values	To reduce duplicated code, to define different service calls, wrong data specification for matching, wrong loop structure	7	Define a dictionary to map entity values to distinct service calls or service data, and call services based on the retrieved entity values.
R8	Handling a group or list of same-typed entities	To reduce duplicated code, to define different service calls, to define varying trigger(s), wrong data retrieval or calculation, wrong algorithm design	5	Use `expand()` to enumerate a group or list of entities, and define filters to pick elements that satisfy certain requirements.

```yaml
service: counter.configure
data_template:
  value: '{{states('counter.aux_ac_pieza')}}'
# Bug: A pair of single quotes enclose another pair of single quotes
# Fix: Replace the inner quotes with double quotes, as below
  value: '{{states(''counter.aux_ac_pieza'')}}'
```

Listing 21. A bug and fix related to the quote usage (R1) [32]

```yaml
action:
  - repeat:
      while:
        - condition: state
          entity_id: binary_sensor.group_door_sensor_at_night
          state: 'on'
        - condition: time
          before: '05:00:00'
          after: '23:30:00'
sequence:
  - service: notify.mobile_app_iphone
# Bug: The while-loop does not have "sequence" aligned with "while"
# Fix: To indent the "sequence" block
    sequence:
      - service: notify.mobile_app_iphone
```

Listing 22. A bug and fix about indentation (R2) [6]

collections, mappings, and their items. For instance, the buggy version in Listing 22 tries to repetitively (1) check whether the door is open at night (between 23:30pm and 5am next day), and (2) send a notification if so. Such a logic can be realized with a `while`-loop, which encloses two blocks—a `while` block and a `sequence` block—with keyword `repeat`. However, as the buggy version did not indent the `sequence` block properly (see lines 10–11), the automation fails.

R3 replaces device triggers with triggers of numeric_state or state, because device triggers are too limited to express triggering events. For instance, the buggy version in Listing 3 defines a device trigger, to fire when sensor `binary_sensor.v4b01_motion` detects no motion for the number of minutes developers specified via `input_number.timeout_offices`, a numeric input box in GUI. However, device triggers do not support templating (see line 11), so the buggy version fails. The fix is to replace that trigger with a state trigger (lines 14–20).

R4 is about state access. To correctly read the state of an entity (see Listing 2), people are suggested to call function `states(...)` with that entity's ID, such as `states('sensor.solaredge_current_power')`.

R5 fixes misspelled states, by suggesting developers to look up valid states of a given entity in Developers Tools of HA IDE and properly specify states. For example, `input_boolean.ecobee_fan_on_off` in Listing 1 is a boolean variable representing an input box in GUI, whose state value is `'on'` or `'off'`. Developers must specify states accordingly.

R6 defines alternative triggers in one automation rule, and calls services differently depending on the fired trigger. It establishes correspondence between triggers and service calls by (1) assigning distinct IDs to triggers, and (2) referring to those IDs in service calls. For instance, suppose that an HA user wants to turn on a wifi bulb at 3pm and off at 5pm

```yaml
- alias: TurnOnOffBulb
  trigger:
    - id: 'on'
      platform: time
      at: '15:00:00'
    - id: 'off'
      platform: time
      at: '17:00:00'
    - id: 'on'
      platform: state
      entity_id: light.mybulb
      from: 'unavailable'
  condition: "{{ today_at('15:00:00') <= now() < today_at('17:00:02') }}"
  action:
    - service: 'light.turn_{{ trigger.id }}'
      target:
        entity_id: light.mybulb
```

Listing 23. Different service calls based on fired triggers (R6) [15]

```yaml
- id: '1631649587197'
  alias: Alert - Bin collection tomorrow
  description: ''
  variables:
    waste: >
      {% set t = (now() + timedelta(days=1)).date() %}
      {% set x =
 { strptime(states('sensor.rubbish_bin_collection'),'%a %d %b %Y').date():
'Rubbish', strptime(states('sensor.recycling_bin_collection'),'%a %d %b
%Y').date(): 'Recycling', strptime(states('sensor.garden_bin_collection'),'%a
%d %b %Y').date(): 'Garden' } %}
      {{ x[t] if t in x.keys() else 'nothing' }}
  trigger:
  - platform: time
    at: '20:00:00'
  condition: "{{ waste != 'nothing' }}"
  action:
  - service: notify.admin_devices
    data:
      message: "{{ waste }} bin collection tomorrow!"
  mode: single
```

Listing 24. Different service calls based on entity values (R7) [2]

every day; the bulb should be also turned on whenever it gets online during 3pm-5pm after being offline, because the bulb sometimes goes offline at 2:59pm due to connectivity issues. To define both turning-on and turning-off behaviors of the bulb in the same automation rule, lines 2–12 in Listing 23 define three alternative triggers, and assign two IDs to those triggers separately: `'on'` and `'off'`; line 15 calls the corresponding service (i.e., `light.turn_on` or `light.turn_off`) by composing a service name using the `trigger.id` value.

R7 reads values of multiple entities (e.g., sensors), and calls services differently depending on those values. It typically defines a dictionary to map entity values with service calls or data. Given an entity value, it looks up the dictionary to take an action. For instance, Listing 24 uses three self-defined sensors to pull bin collection dates (rubbish, recycling, garden) from a website, and sends notification messages the night before a collection is due. In the automation, the Jinja variable `x` is defined as a dictionary (line 7), to map pulled dates of bin collection to bin types. The Jinja variable `t` holds the date of next day (line 6). Variable `waste` is initialized as `'nothing'` if the next day's date `t` does not match any bin collection date, or as a bin type if `t` finds a match (line 9). Finally, the service is called with the value set to `waste` (line 17).

R8 defines pipelines to uniformly handle a group or list of same-typed entities. Each pipeline typically starts with the Jinja function call `expand()` to enumerate all elements in a group/list, and then invokes Jinja built-in filters (e.g., `map()`) to refine or process elements. For instance, suppose that an HA user wants to set all Google speakers at home to the same volume, by referring to the most recent volume setting among all speakers. Listing 25 satisfies the automation need through three steps. First, to identify all turned-on speakers, it enumerates speakers (lines 4–7), rejects turned-off

```yaml
action:
  - variables:
      speakers_on: >
        {{ expand("media_player.google_home_bedroom",
        "media_player.google_nestmini_office",
        "media_player.google_nesthub_living_room",
        "media_player.google_nestmini_kitchen")| rejectattr('state', 'eq', 'off')
        | map(attribute='entity_id') | list }}
      recent_volume: >
        {{ expand("sensor.volume_google_speaker_bedroom",
        "sensor.volume_google_speaker_office",
        "sensor.volume_google_speaker_living_room",
        "sensor.volume_google_speaker_kitchen" )
        | sort(attribute='last_changed', reverse=true)
        | map(attribute='state') | first | float(0) }}
  - service: media_player.volume_set
    data:
      volume_level: "{{ recent_volume }}"
    target:
      entity_id: "{{ speakers_on }}"
```

Listing 25. Handling a group or list of same-typed entities (R8) [5]

entities (line 7), gathers values of attribute `'entity_id'` (line 8), and lists those values. Second, to acquire the most recent volume setting, it enumerates all volume sensors (lines 10–13), sorts them in descending order of their latest change timestamps (line 14), gathers values of attribute `'state'` (i.e., timestamps), gets the first one in that list (i.e., the mosts recent one), and converts the value to a floating-point number. Third, the automation sets all turned-on speakers to the same volume.

Notice that R1–R3 address debugging issues; R4–R5 resolve debugging and implementation issues; R6 fixes optimization and debugging issues; R7–R8 handle issues of all three categories. These strategies provide two insights for future tool support of smart home development. First, developers repetitively commit certain mistakes, and such mistakes present clear bug patterns; it is promising to create new tools to detect and fix repetitive bugs. Second, some developers cannot (1) define diverse actions depending on the fired triggers or entity values, or (2) define uniform processing for elements in a group/list, although such automation needs are not uncommon. Therefore, it will be helpful to create tools, which generate automation implementations to at least partially satisfy those needs.

Finding 5: *The eight resolution strategies imply that domain experts addressed recurring issues by following certain principles; it is promising to automate such principles.*

4.3 RQ3: Current Debugging Tool Support

To study the existing tools for debugging YAML files or YAML-based automation configuration, we searched online with keywords "YAML validator" and found five publicly available tools: YAML Validator by Code Beautify [51], YAML Lint [63], YAML Checker [55], YAML Validator by JSON formatter [57], and ONLINEYAMLTOOLS [67]. Additionally, as HA provides an IDE to help automation development, we also included its checker into the tool list for evaluation. As shown in Table 2, for simplicity, we assigned a unique ID to each tool, and will consistently use T1–T6 to refer to these tools. In our experiments, **because we found no tool to suggest any bug fix, this section focuses on our results about tools' capabilities of bug detection.**

4.3.1 Metrics for Automatic Bug Detection. We defined three metrics to evaluate the bug detection capabilities of tools:

Recall (R) measures among all known bugs, how many of them are reported by a tool:

$$R = \frac{\text{\# of known bugs reported}}{\text{Total \# of known bugs}} \tag{1}$$

To compute the recall of a tool, we first intersected the tool-reported bugs with known 129 bugs. Then we computed the count ratio between this intersection and the known bug set.

Precision (P) measures among all bugs reported by a tool, how many of them are real bugs:

$$P = \frac{\text{\# of true bugs reported}}{\text{Total \# of bugs reported}} \tag{2}$$

The 129-bug set does not include all bugs existent in given YAML files/snippets, as developers sometimes omitted discussion on trivial bugs. To compute a tool's precision, we manually inspected all bug reports. If a report B_r clearly describes a known bug, we consider B_r a true positive. Otherwise, if B_r is confusing, we applied the tool which originally output B_r also to the developer-fixed version. If (1) that tool reported nothing for the fixed version or (2) our domain knowledge confirms the bug reported by B_r, we consider B_r a true positive; otherwise, B_r is a false positive.

F-score (F) combines P with R, to measure the overall accuracy of bug detection as below:

$$F = \frac{2 \times P \times R}{P + R} \tag{3}$$

F is the harmonic mean of P and R. All three metrics have their values vary within [0, 1]. The higher, the better.

Table 2. The bug detection capabilities of current tools

ID	Tool Name	# of Bugs Reported		P	R	F
		Correct	Incorrect			
T1	YAML Validator by Code Beautify [51]	15	3	83%	9%	15%
T2	YAML Lint [63]	14	3	82%	7%	13%
T3	YAML Checker [55]	15	5	75%	8%	14%
T4	YAML Validator by JSON formatter [57]	15	4	79%	8%	14%
T5	ONLINEYAML TOOLS [67]	15	4	79%	8%	14%
T6	HA IDE Checker [17]	15	1	94%	8%	14%

4.3.2 Tool Effectiveness. As shown in Table 2, the tools behaved similarly to each other. All tools achieved high precision (75%–94%), low recall (7%–9%), and low F-scores (13%–15%). They reported bugs in 14 common YAML files/snippets, and detected no bug in 106 common files/snippets. High precision means that the six tools report bugs with low false positives. Namely, if a tool reports a bug for a given YAML file/snippet, the file or snippet is very likely to be buggy. Meanwhile, low recall means that these tools poorly reveal known bugs. Among the 129 bugs we distilled from HA discussion, only 9–11 bugs were covered by the reports generated by each tool; the combination of all tools' outputs only revealed 14 known bugs. Due to such low recall rates, the measured F values are also low.

Among the tools, T6 got the highest precision; T1 got the highest recall and F-score; T4 and T5 acquired identical values for all metrics. Between T4 and T5, we found a significant overlap in tool-generated bug reports; both tools described all errors in almost identical ways. This interesting observation implies that T4 and T5 may share the same core implementation. Additionally, we once hypothesized T6 to outperform other tools in all metrics, because T6 was specially designed to reveal HA-related issues while all other tools are general YAML file checkers. Surprisingly, we found T6 to work similarly with other tools, and only outperformed the other tools in precision.

The results indicate that all six tools exhibited similarly low recall rates. This suggests that existing tools primarily focus on syntax validation rather than detecting semantic or logical errors in YAML configurations. Notably, none of the tools provided automated bug-fixing recommendations. This indicates a gap in current YAML debugging capabilities, emphasizing the need for more advanced debugging techniques that extend beyond syntax validation.

Finding 6: *For bug detection, existing tools achieved high precision, low recall, and low F-scores.*

4.3.3 Characterization of Tool-Generated Bug Reports. Among the bug reports output by different tools, we observed two phenomena. First, *the bug reports are purely about wrong formats.* These issues may involve wrong quote usage, bad indentation, and wrong tag (i.e., token) usage. However, as mentioned in Section 4.1, the 129 known bugs cover 19 distinct concepts; "wrong format" is just one of them, and its bugs seem easier to detect because formatting rarely requires for advanced program analysis. Unfortunately, existing tools cannot detect other types of bugs. Even within the 20 known bugs of "Wrong format", we still observed 11 cases missed by all tools. For three cases, T3–T5 failed to interpret the legal notation "!" and incorrectly reported it as wrong tag usage.

Finding 7: *Existing tools only revealed a relatively simple type of bugs—wrong format; even for the detection of such bugs, current tools suffer from significant false positive and false negative issues.*

Second, *the error messages in many bug reports are very confusing.* They may incorrectly pinpoint the bug location, wrongly describe an error, or provide meaningless hints that mislead developers. For instance, Listing 21 shows a buggy snippet to misuse quotes at line 3. Essentially, single quotes should not be nested. However, T1 and T2 output confusing messages to complain about unexpected characters/scalar at line 3 (see Table 3), without clarifying which character/scalar is unexpected. T3 and T6 reported bad indentation, but no indentation issue exists at all. T4 and

Table 3. The confusing tool-generated reports for Listing 21

ID	Error Message
T1	Error: Unexpected characters near "counter.aux_ac_pieza')}}'". Line : 3 value: '{{states('counter.aux_ac_pieza')}}'
T2	Unexpected scalar at node end at line 3, column 21
T3	bad indentation of a mapping entry (3:21)
T4	Error: can not read an implicit mapping pair; a colon is missed at line 3, column 46
T5	YAMLException: can not read an implicit mapping pair; a colon is missed at line 3, column 46
T6	bad indentation of a mapping entry (3:21)

T5 mentioned a missing colon, which does not reflect the wrongly used quotes, either. Among the 109 bug reports we examined in total, 45 reports contain confusing error messages. T1 produced the biggest number of confusing reports—12, while T3–T6 output the fewest—6.

Finding 8: *In our experiment, 41% (45/109) of the examined bug reports are confusing, which evidences the need of improving the error-reporting mechanism.*

5 Threats to Validity

Threats to External Validity. All our observations are based on the experimental dataset. Although the study analyzes 190 threads in depth, the sample size, while substantial, may not fully represent the diverse range of experiences and challenges encountered by all developers. In the future, we will add more data to our dataset, and conduct further analysis to draw more generalized conclusions about automation configuration issues in smart homes.

Threats to Internal Validity. Our manual analysis for the collected data and tool outputs is subject to human bias, and is limited to our domain knowledge. To mitigate the problem, we had two authors independently examine the data in multiple iterations. When they disagreed upon certain data labels or classification criteria, they had discussion until coming to an agreement and enforced the same labeling/classification mechanism in the next iteration.

6 Lessons Learned

Below are actionable items we learned from this study.

For Tool Builders and SE Researchers: Our study shows that programmers do not have sufficient domain knowledge or good tool support for home automation development. Tool builders and SE researchers can create tools to (1) better detect or fix bugs in YAML-based automation scripts or (2) generate automation implementation from scratch. We suggest three future directions.

First, syntax checkers and fixers. Developers often use quotes and indentation in wrong ways (Sections 4.1 and 4.2), and the current tool support is poor (Section 4.3). HA defines a set of grammar rules on top of the basic YAML rules (e.g, `while`-loop). To enforce these domain-specific rules, future work can create parsers to analyze YAML files, locate HA-specific keywords as well as structures, comprehend automation rules based on the extracted information, and create syntax trees. In the tree-creation procedure, parsers can detect malformed content by reporting any violation of grammar rules; they can further suggest fixes by observing grammar rules and program context.

Second, semantic checkers and fixers. As described by Section 4.2, HA defines semantic rules on segment-specific concepts (e.g., device triggers do not support templating), and segment-agnostic ones (e.g., sensor states should be accessed via the function `state(...)`). To enforce these rules, future work can create analyzers to traverse the parsing trees mentioned above, gather semantic information like the definition or usage of variables/templates/entities, compare the gathered information with predefined bug patterns, and report a bug for each found pattern-match. The analyzers can also automate or suggest fixes based on predefined bug-fixing patterns, or data analysis of correct automations.

Third, generators of automation configurations. As mentioned in Section 4.2, some developers have similar automation needs (e.g., to call distinct services based on the fired triggers), and the code to satisfy those needs share commonality (e.g., defining and using trigger IDs). Future work can use data-driven approaches like machine learning and large language models (LLMs) to (1) infer the correspondence between automation needs and YAML code, and (2) generate YAML code given automation specification. Such tools can be used in combination with the bug detectors and fixers mentioned above, to iteratively refine or optimize automations.

For HA Creators or Maintainers: HA users or programmers are often confused about the correct usage of some program constructs, built-in functions, or templates (Section 4.1). The error messages generated by the existing HA checker seem not quite helpful (Section 4.3). To help developers better adopt HA and further broaden the platform's impact, HA creators or maintainers may need to improve documentation, and to combine existing concrete YAML examples with a more systematic and comprehensive concept explanation. They may also need to enhance the error-reporting mechanism, to clarify the root causes or even fixes of reported bugs.

For Programmers: Carefully read the available HA documentation to follow best practices and avoid the well-known pitfalls. When asking questions on HA, provide all relevant information (e.g., automation specification, unsuccessful trials, error messages, and help request) to benefit most from the community wisdom.

7 Related Work

The related work includes empirical studies on Internet-of-Thing (IoT) systems, and bug detection in those systems.

7.1 Empirical Studies on IoT Systems

Studies were recently performed to characterize issues or problems in IoT systems [52, 53, 56, 59, 65, 69, 71]. For instance, Fernandes et al. [56], Alrawi et al. [52], and Zhou et al. [71] analyzed the security properties of IoT platforms and systems. In contrast, our research focuses on coding issues in IoT automation configuration, covering three categories: (1) implementing new features, (2) debugging, and (3) optimization. He et al. [59] did an online survey with 72 users of smart home systems, to learn their negative user experiences. The participants reported fears of breaking the system by writing code, and struggles of diagnosing or recovering from system failures. Similarly, Makhshari and Mesbah [65] did interviews and surveys with IoT developers; they also found testing and debugging as the major challenges. However, neither study examines any widely used smart home platform to characterize the bug patterns or fixing strategies, let alone to provide concrete actionable advices to tool builders. Our study is motivated by and complement both studies.

Brackenbury et al. [53] focused on the trigger-action programming (TAP) model. They systematized the temporal paradigms through which TAP systems could express rules, and classified TAP programming bugs into three categories: control logic, timing, and inaccurate user expectation. As with Brackenbury et al., we also identified bugs related to these three general categories (e.g., wrong loop structure and wrong data specification for matching). However, our taxonomy is finer-grained and more comprehensive, as we derived bug patterns from real-world bugs instead of speculation on the TAP model. Our observations reflect the real-world bug distribution among patterns; we also characterized (1) bugs violating syntactic rules (e.g., wrong formats) and more diverse semantic rules (e.g., data access), (2) developers' bug fixes, (3) recurring implementation requests, and (4) frequent optimization needs.

Wang et al. [69] inspected 330 device integration bugs mined from HAC to characterize any root cause, fix, trigger condition, and impact of those bugs. Our study is closely related to the work by Wang et al., as we also examined data

mined from HAC. However, our study is irrelevant to device integration; instead, it focuses on the coding issues in automation configuration.

7.2 Automatic Software Bug Detection in IoT Systems

Tools were created to automatically detect bugs in IoT systems [54, 58, 60–62, 68, 70]. Specifically, VulHunter [70] detects new vulnerabilities by analyzing the known vulnerability patch packs in Industry IoT. SOTERIA [54] verifies whether IoT apps adhere to the identified safety, security, and functional properties via model checking. Liang et al. [62] and Fu et al. [58] separately created tools to reveal bugs in IoT operating systems (OSes).

Some tools [60, 61, 68] detect conflicting interactions between smart home IoT applications, because these conflicts can result in undesired actions like locking a door during a fire. For instance, Trimananda et al. [68] studied 198 official and 69 third-party apps on Samsung SmartThings, and found 3 major categories of app conflicts. Based on their observations, the researchers created a conflict detector that uses model checking to detect up to 96% of the conflicts. Li et al. [61] categorized conflicts in a totally different way. They also invented a graph structure named *IA graph* to represent the controls in each IoT app and event schedules. With that representation, they created an efficient algorithm to leverage first-order logic and SMT solvers to detect conflicts.

Our study complements all tools mentioned above; it focuses on a different type of IoT bugs—bugs in automation configuration or implementation. Such implementation bugs are less relevant to security, IoT kernels, or inter-app conflicts. However, they are still important as these bugs can prevent end-users from realizing the desired automation rules or achieving high-quality automations.

8 Conclusion

As the growing prevalence of smart homes, we believe that they will become crucial to lower utility costs, improve people's life, and protect people's properties. Wrongly configured home automations can waste utilities, jeopardize people's life, and compromise home safety/security. Our study characterizes the challenges and opportunities in HA-based smart homes, enlightening future research to improve end-user programming and smart-home quality.

Acknowledgment

We thank anonymous reviewers for their valuable comments on our earlier version of the paper. This work was partially supported by NSF-1845446, CCI-P4TLGZ22, CCI-PMH7EUMV, and the National Natural Science Foundation of China No. 62272295.

References

[1] 2020. Schedule light color based on holidays. https://community.home-assistant.io/t/schedule-light-color-based-on-holidays/256020.
[2] 2021. Automation - reference condition template variable in action notification. https://community.home-assistant.io/t/automation-reference-condition-template-variable-in-action-notification/339858.
[3] 2021. How to check if doors were opened or closed last 15 minutes? https://community.home-assistant.io/t/how-to-check-if-doors-were-opened-or-closed-last-15-minutes/331897.
[4] 2022. Automation - house thermostat based on phone location - not working. https://community.home-assistant.io/t/automation-house-thermostat-based-on-phone-location-not-working/401942.
[5] 2022. Automation Action Templating. https://community.home-assistant.io/t/automation-action-templating/409680.
[6] 2022. Automation door alarm use template. https://community.home-assistant.io/t/automation-door-alarm-use-template/439744.
[7] 2022. Automation not close the valve after timer finished. https://community.home-assistant.io/t/automation-not-close-the-valve-after-timer-finished/415726.

[8] 2022. Automation: play radio to a Sonos speaker group. https://community.home-assistant.io/t/automation-play-radio-to-a-sonos-speaker-group/440662.
[9] 2022. Boolean triggering script. https://community.home-assistant.io/t/boolean-triggering-script/404465.
[10] 2022. Can someone help me a bit with my alarm script (siren not repeating for some reason). https://community.home-assistant.io/t/can-someone-help-me-a-bit-with-my-alarm-script-siren-not-repeating-for-some-reason/442911.
[11] 2022. Cant use template in for field of trigger automation. https://community.home-assistant.io/t/cant-use-template-in-for-field-of-trigger-automation/400475.
[12] 2022. Combining automations? https://community.home-assistant.io/t/combining-automations/415637.
[13] 2022. Condition on trigger.entity_id. https://community.home-assistant.io/t/condition-on-trigger-entity-id/442838.
[14] 2022. Exceptions on automation. https://community.home-assistant.io/t/exceptions-on-automation/409661.
[15] 2022. Exceptions on automation. https://community.home-assistant.io/t/exceptions-on-automation/409661.
[16] 2022. Help adding a “Silence notification” style button to a push notification (iOS). https://community.home-assistant.io/t/help-adding-a-silence-notification-style-button-to-a-push-notification-ios/399163.
[17] 2022. Home Assistant. https://www.home-assistant.io.
[18] 2022. How to create an automation when 2 ppl are away at the same time. https://community.home-assistant.io/t/how-to-create-an-automation-when-2-ppl-are-away-at-the-same-time/406896.
[19] 2022. Jinja. https://palletsprojects.com/p/jinja/.
[20] 2022. Lights on only after sunset and before sunrise. urlhttps://community.home-assistant.io/t/lights-on-only-after-sunset-and-before-sunrise/307761.
[21] 2022. My automation generates an error message. https://community.home-assistant.io/t/my-automation-generates-an-error-message/434970.
[22] 2022. Not getting actionable button in push notificaiton. https://community.home-assistant.io/t/not-getting-actionable-button-in-push-notificaiton/446658.
[23] 2022. Notify - How to display sensor value in message. https://community.home-assistant.io/t/notify-how-to-display-sensor-value-in-message/376646.
[24] 2022. Numeric state trigger is not reliable when used as trigger. https://community.home-assistant.io/t/numeric-state-trigger-is-not-reliable-when-used-as-trigger/445265.
[25] 2022. Numeric state trigger is not reliable when used as trigger. https://community.home-assistant.io/t/numeric-state-trigger-is-not-reliable-when-used-as-trigger/445265.
[26] 2022. Please help me fix (long) automation. https://community.home-assistant.io/t/please-help-me-fix-long-automation/433663.
[27] 2022. Problem with automation - time delay trigger. https://community.home-assistant.io/t/problem-with-automation-time-delay-trigger/454590.
[28] 2022. Putting information from sensor into mobile notification. https://community.home-assistant.io/t/putting-information-from-sensor-into-mobile-notification/264000.
[29] 2022. Reliably set light temperature of different light models in one area. https://community.home-assistant.io/t/reliably-set-light-temperature-of-different-light-models-in-one-area/430578.
[30] 2022. Repeat until input_button is pressed. https://community.home-assistant.io/t/repeat-until-input-button-is-pressed/429300.
[31] 2022. Script Syntax - Home Assistant. https://www.home-assistant.io/docs/scripts/.
[32] 2022. Set counter value equals to other counter in automation. https://community.home-assistant.io/t/set-counter-value-equals-to-other-counter-in-automation/442485.
[33] 2022. SOLVED: Exclude entities from a variable. https://community.home-assistant.io/t/solved-exclude-entities-from-a-variable/420452.
[34] 2022. Strategies for dealing with restarts in automations. https://community.home-assistant.io/t/strategies-for-dealing-with-restarts-in-automations/400922.
[35] 2022. Time Before and After condition, I’m going crazy. https://community.home-assistant.io/t/time-before-and-after-condition-i-m-going-crazy/416170.
[36] 2022. With 21.1% CAGR, Smart Home Market Worth USD 380.52 Billion in 2028. https://www.globenewswire.com/en/news-release/2022/06/28/2470249/0/en/With-21-1-CAGR-Smart-Home-Market-Worth-USD-380-52-Billion-in-2028.html.
[37] 2023. How Popular Is Home Assistant Compared To The Competition? https://whatsmarthome.com/how-popular-is-home-assistant/.
[38] 2023. If starting again what smart home platform would people go with ? https://www.reddit.com/r/smarthome/comments/16sk3qg/if_starting_again_what_smart_home_platform_would/.
[39] 2023. The best home automation systems: Compare SmartThings, Apple HomeKit, Amazon Alexa, and more. https://www.zdnet.com/home-and-office/smart-home/best-home-automation-system/.
[40] 2023. Wait for trigger timing out even when condition is met. https://community.home-assistant.io/t/wait-for-trigger-timing-out-even-when-condition-is-met/425310.
[41] 2024. Automation actions - Home Assistant. https://www.home-assistant.io/docs/automation/action/.
[42] 2024. Automation Conditions - Home Assistant. https://www.home-assistant.io/docs/automation/condition/.
[43] 2024. Automation Trigger - Home Assistant. https://www.home-assistant.io/docs/automation/trigger/.
[44] 2024. GitHub. https://github.com.

[45] 2024. Home Assistant Analytics. https://analytics.home-assistant.io.
[46] 2024. Home Assistant Community. https://community.home-assistant.io.
[47] 2024. IFTTT - Automation for business and home. https://ifttt.com.
[48] 2024. Tags - Home Assistant Community. https://community.home-assistant.io/tags.
[49] 2024. The global smart home market size was valued at $80.21 billion in 2022 & is projected to grow from $93.98 billion in 2023 to $338.28 billion by 2030. https://www.fortunebusinessinsights.com/industry-reports/smart-home-market-101900.
[50] 2024. YAML. https://yaml.org.
[51] CodeBeautify 2023. Accessed October 11, 2023. *CodeBeautify - YAML Validator.* https://codebeautify.org/yaml-validator
[52] Omar Alrawi, Chaz Lever, Manos Antonakakis, and Fabian Monrose. 2019. Sok: Security evaluation of home-based iot deployments. In *2019 IEEE symposium on security and privacy (sp)*. IEEE, 1362–1380.
[53] Will Brackenbury, Abhimanyu Deora, Jillian Ritchey, Jason Vallee, Weijia He, Guan Wang, Michael L. Littman, and Blase Ur. 2019. How Users Interpret Bugs in Trigger-Action Programming. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI '19)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3290605.3300782
[54] Z Berkay Celik, Patrick McDaniel, and Gang Tan. 2018. Soteria: Automated {IoT} Safety and Security Analysis. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 147–158.
[55] YAML Checker. Accessed October 11, 2023. *YAML Checker.* https://yamlchecker.com/
[56] Earlence Fernandes, Jaeyeon Jung, and Atul Prakash. 2016. Security analysis of emerging smart home applications. In *2016 IEEE symposium on security and privacy (SP)*. IEEE, 636–654.
[57] JSON Formatter. Accessed October 11, 2023. *JSON Formatter - YAML Validator.* https://jsonformatter.org/yaml-validator
[58] Lirong Fu, Shouling Ji, Kangjie Lu, Peiyu Liu, Xuhong Zhang, Yuxuan Duan, Zihui Zhang, Wenzhi Chen, and Yanjun Wu. 2021. CPscan: Detecting Bugs Caused by Code Pruning in IoT Kernels *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 794–810. https://doi.org/10.1145/3460120.3484738
[59] Weijia He, Jesse Martinez, Roshni Padhi, Lefan Zhang, and Blase Ur. 2019. When Smart Devices Are Stupid: Negative Experiences Using Home Smart Devices. In *2019 IEEE Security and Privacy Workshops (SPW)*. 150–155. https://doi.org/10.1109/SPW.2019.00036
[60] Bing Huang, Dipankar Chaki, Athman Bouguettaya, and Kwok-Yan Lam. 2023. A Survey on Conflict Detection in IoT-based Smart Homes. *ACM Comput. Surv.* 56, 5, Article 122 (nov 2023), 40 pages. https://doi.org/10.1145/3629517
[61] Xinyi Li, Lei Zhang, and Xipeng Shen. 2020. DIAC: An Inter-app Conflicts Detector for Open IoT Systems. *ACM Trans. Embed. Comput. Syst.* 19, 6, Article 46 (oct 2020), 25 pages. https://doi.org/10.1145/3391895
[62] Hongliang Liang, Qian Zhao, Yuying Wang, and Haifeng Liu. 2016. Understanding and detecting performance and security bugs in IoT oses. In *2016 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*. IEEE, 413–418.
[63] YAML Lint. Accessed October 11, 2023. *YAML Lint.* https://www.yamllint.com/
[64] Purdy Lounge. Retrieved September 12, 2022. *OpenHAB Vs Home Assistant: Detailed Comparison By An Expert in 2022.* https://purdylounge.com/openhab-vs-home-assistant/
[65] Amir Makhshari and Ali Mesbah. 2021. IoT bugs and development challenges. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 460–472.
[66] SmartThings. 2024. *SmartThings.* https://www.smartthings.com/
[67] Online YAML Tools. Accessed October 11, 2023. *Online YAML Tools.* https://onlineyamltools.com/validate-yaml
[68] Rahmadi Trimananda, Seyed Amir Hossein Aqajari, Jason Chuang, Brian Demsky, Guoqing Harry Xu, and Shan Lu. 2020. Understanding and automatically detecting conflicting interactions between smart home IoT applications. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1215–1227.
[69] Tao Wang, Kangkang Zhang, Wei Chen, Wensheng Dou, Jiaxin Zhu, Jun Wei, and Tao Huang. 2022. Understanding device integration bugs in smart home system. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 429–441.
[70] Fu Xiao, Le-Tian Sha, Zai-Ping Yuan, and Ru-Chuan Wang. 2020. VulHunter: A Discovery for Unknown Bugs Based on Analysis for Known Patches in Industry Internet of Things. *IEEE Transactions on Emerging Topics in Computing* 8, 2 (2020), 267–279. https://doi.org/10.1109/TETC.2017.2754103
[71] Wei Zhou, Chen Cao, Dongdong Huo, Kai Cheng, Lan Zhang, Le Guan, Tao Liu, Yan Jia, Yaowen Zheng, Yuqing Zhang, Limin Sun, Yazhe Wang, and Peng Liu. 2021. Reviewing IoT Security via Logic Bugs in IoT Platforms and Systems. *IEEE Internet of Things Journal* 8, 14 (2021), 11621–11639. https://doi.org/10.1109/JIOT.2021.3059457