RobotBT: Behavior-Tree-Based Test-Case Specification for the Robot Framework

Sven Peldszus
Ruhr University Bochum
Germany

Noubar Akopian
Ruhr University Bochum
Germany

Thorsten Berger
Ruhr University Bochum and
Chalmers | University of Gothenburg
Germany and Sweden

ABSTRACT

The Robot Framework is a popular and widely used test automation framework that abstracts test case specifications toward natural language specifications. This makes it well suited for implementing high-level test cases, at least as long as the functions provided by Robot can support the intended functionality. For more complicated test cases, custom and often deeply nested functionality specifications are required, and the readability of Robot test cases tends to decrease. We present RobotBT, a library for the Robot framework that addresses these shortcomings by adding support for specifying test cases using behavior trees. Behavior trees are a comprehensive method for specifying complex behaviors based on a control flow model that orchestrates the execution of functionality. We evaluated RobotBT on a test suite for GUI testing from G DATA CyberDefense AG and interviewed their engineers about the usability, readability, and applicability of RobotBT. Our results show that BTs improve the expressiveness and readability of Robot Framework test cases and are applicable to practical problems.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; **Domain specific languages**; **Control structures**.

KEYWORDS

Test Case Specification, Robot Framework, Behavior Tree

1 INTRODUCTION

The Robot Framework [10] is an open-source framework for automating end-to-end, system, and UI tests. Based on the principles of Acceptance-Test Driven Development [7] and Behavior Driven Development [13], one of its main goals is to provide a simple and intuitive interface for writing and executing test cases that eliminates the need for extensive programming knowledge, making it suited for non-programmers, such as domain experts. To this end,

the Robot Framework provides a simple domain-specific language (DSL) [3] to ease writing comprehensible test code.

The benefit of providing domain-specific and natural-language-like test specifications comes at the cost of expressiveness. The Robot Framework is limited to simple test cases that do not require complex logic to actuate the units under test. While the logic of test case specifications should generally be simple, GUI testing in particular often requires declaring more complex behaviors in the test case. Imagine a test case that involves opening an application from the Windows tray, where the icon to click can be in different locations, e.g., directly on the tray or in the grouped icons. Furthermore, notifications may overlap this area and need to be closed first. To address such issues, G DATA CyberDefense AG, a leading provider of antivirus solutions, which uses Robot Framework for GUI testing of its antivirus software, had to abstract the entire logic of the complicated decision processes in its test suite to several external specifications written in high-level programming languages. This adds a new level of complexity to the test suite, which is still difficult to read.

Similar problems in expressing complicated behavior have been successfully addressed by the introduction of *Behavior Trees* (BTs) in the robotics domain [5, 9]. A BT is a model that describes the switching between a finite set of tasks, allowing developers to create complex logic composed of simple tasks [4, 5]. It is constructed as a hierarchical set of nodes that control decision making and is used to control the behavior of autonomous systems, such as robots. BTs are often used to model and control the decision-making processes of these systems and can be applied to a wide range of applications.

We use behavior trees to facilitate the specification of test case behavior by improving the readability and understandability of test cases. Specifically, behavior trees can improve the expressiveness and flexibility of test cases by allowing developers to specify complex behaviors in a clear and concise manner. Behavior trees also foster reusability, since behavior can easily be extended and customized by adding new nodes to the three or by modifying existing ones. Behavior trees can be especially helpful when a test case becomes complicated due to many decisions, which threatens the readability and maintainability of Robot test code.

We present RobotBT, a behavior tree library for Robot Framework. With RobotBT, we assessed the feasibility of behavior trees for the specification of Robot test cases. We show that it can actually improve the specification of test cases, making them more concise and extensible, among others. Our integration of behavior trees with the Robot Framework was guided by two research questions:

RQ1 – Suitability: Are behavior trees suitable to express real-world Robot test cases?

RQ2 – Comprehensibility: Do behavior trees improve the readability and understandability of Robot test cases?

Our contributions comprise the *Robot Framework Behavior Tree Library (RobotBT)*, a library for the Robot Framework that enables the use of behavior trees for test case specification, and an evaluation of RobotBT on a real-world Robot test suite from G DATA CyberDefense AG. The source code of RobotBT, the full real-world test case specification used as an example in this work, artificial examples, and raw data are publicly available in our GitHub repository [2]. We also provide a video demonstration of RobotBT.[1]

2 BACKGROUND

We now briefly introduce the Robot Framework and our running example from G DATA's test suite, as well as behavior trees as a DSL for modeling behavior.

2.1 Robot Framework

The Robot Framework [10] provides a DSL called *Robot* for specifying test cases. The DSL mainly comprises expressions, which are composed of functions (in Robot simply called *keywords*) that are either provided by the Robot framework or are user-defined. Statements in the form of variable assignments are also possible. The concatenation of the functions, together with the domain-specific syntax, allows forming natural-language-like sentences, making the test-case specification more readable and intuitive. These functions (i.e., *keywords*) can be implemented in Python or Java, or they can be compositions of other keywords. However, while being intuitive and domain-specific, the DSL is limited to simple test cases, leading to less-readable and inconcise expressions for complicated test cases, which include multiple decisions to effectively actuate the unit under test, such as for the interactions required for GUI testing.

Listing 1 shows a simple excerpt of a Robot test case that is part of a GUI test case of G DATA CyberDefense AG. This test case tests the opening of the user interface of their antivirus software, called the G DATA Security Center, from the Windows system tray. As part of this, the shown excerpt defines a composite keyword that locates the G DATA Security Center icon in the Windows tray and stores the location in a variable that is used to open the Security Suite in further parts of the test case.

The excerpt calls in line 1 of Listing 1 the G DATA custom keyword "Run And Return Status", which takes another keyword ("Wait Until Succeeds") as input and stores the result of its execution in a variable, here $_exists_sys_tray. "Wait Until Succeeds" calls a third keyword ("Elem Should Exist") up to a given number of tries (10 in the example) and waits a given amount of time (100ms) between calls. It returns true on the first success. "Elem Should Exist" checks if an item exists in Windows at the given location ($SYS_TRAY_GD_ICON). In lines 2 to 6, if the icon exists, the keyword writes the location to a global variable and exits.

Similarly, the keyword definition in Listing 1 proceeds in the following lines for the other possible locations of the Security Center icon. If the icon is not on the system tray, it searches in lines 7 to 10 for the Windows grouping icon and fails if it is not present. If the grouping icon is present, the keyword opens the group in line 11, searches the Security Center icon in the group in line 12, and evaluates the results like before. If the icon was not found in the group either, in lines 19 to 21, the test case fails with an error message.

---
[1]Tool demonstration on YouTube: https://youtu.be/zPK8RdMmFaM

Listing 1: Example test case specified in the Robot DSL

```
1  ${_exists_sys_tray}  Run And Return Status  Wait Until Succeeds
          10x    100ms  Elem Should Exist  ${SYS_TRAY_GD_ICON}
2  IF  ${_exists_sys_tray} == ${True}
3    ${NotGrouped}  Set Var  ${True}
4    Set Global Var  ${NotGrouped}
5    Return From Keyword
6  END
7  ${_status}    Run And Return Status    Elem Should Exist
          ${SYS_TRAY_GROUP}
8  IF  ${_status} == ${False}
9    Fail  msg="Tray icon not present!"
10 END
11 Open Notification Overflow Area
12 ${_exists_not_ovfl}  Run And Return Status  Wait Until Succeeds
          15x  200ms  Elem Should Exist  ${N_OVERFLOW_GD_ICON}
13 IF  ${_exists_not_ovfl} == ${True}
14   ${NotGrouped}    Set Var  ${False}
15   Set Global Var  ${NotGrouped}
16   Close Notification Overflow Area
17   Return From Keyword
18 END
19 IF  ${_exists_sys_tray} == ${False}  and
          ${_exists_not_ovfl} == ${False}
20   Fail  msg="G~DATA tray icon not found."
21 END
```

Although, the shown keyword is the simplest part of the G DATA CyberDefense AG test suite, it already starts to show the impact of the required control flow on test case readability. Yet, it does not even use more complicated control flow concepts such as loops.

2.2 Behavior Trees

Behavior trees are executable models that allow the specification of behavior [8]. They are a popular alternative to state machines, especially in the robotics domain [9]. Behavior trees consist of a hierarchical tree structure, with nodes representing either control flow or executed actions [4, 5]. The nodes are connected by edges that define the order of node execution. This tree structure allows behavior trees to be easily visualized and understood, making them a useful tool for designing and implementing complex behaviors.

One of the key benefits of behavior trees is their flexibility and extensibility [4]. They can be easily modified or extended by adding new nodes or modifying existing ones, allowing developers to adapt the behavior of the system to changing requirements or environments. Behavior trees are also efficient and scalable because they can handle large and complex behavior models without incurring significant overhead. They are also robust and resilient as they can continue to function even if some of the nodes fail.

To give an impression of BTs, Fig. 1 shows the robot test specification from our example in the notation of the popular graphical behavior tree framework BehaviorTree.CPP [6]. For simplicity, the
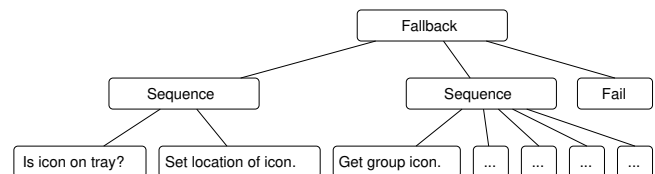

Figure 1: Behavior Tree example

actions to be performed are given in natural language. In practice, these action nodes would trigger the invocation of functions, e.g. written in Python, comparable to keywords in Robot. To realize the control flow, in this example we use two different control flow nodes of BTs. First, a *Fallback* node that executes all child nodes from left to right until the first child executes successfully. If a child executes successfully, this node returns success to its own parent or false otherwise. Second, a *Sequence* node, which again executes all children from left to right, but all children must execute successfully.

In addition to the control nodes shown, behavior trees also support more complicated control, such as loops and parallel execution that are also supported by RobotBT. Overall, behavior trees are an expressive notation for controlling the behavior of autonomous systems. They provide a clear and concise way to specify complex behaviors, and their flexibility, efficiency, and robustness make them suitable for a wide range of applications.

2.3 Behavior Trees in Testing

While behavior trees have been successfully used in various domains [4], in testing, to the best of our knowledge, behavior trees have mainly been used as a specification of a system under test in model-based testing [12]. For example, Lindsay et al. encode requirements in behavior trees and automatically derive test cases that are checked against this specification [11]. More related to our work, Yan and Ma use behavior trees to orchestrate simulation test executions and increase reuse among tests [14].

3 ROBOT TEST CASE SPECIFICATION USING BEHAVIOR TREES

With RobotBT we provide an internal DSL [3] for Robot that allows to specify Robot keywords using behavior trees. We leverage the comprehensive API of Robot Framework that allows developers to create custom libraries and extend the functionality of the framework. This API provides a set of Python classes and methods that can be used to interact with the Robot Framework run-time and create custom keywords. To integrate behavior trees into Robot Framework, RobotBT wraps the Python library *py_trees* [1] for implementation of the behavior tree functionality and exposes its functionality to Robot Framework through a set of keywords. This approach allows developers to leverage the existing behavior tree implementation, while still being able to use Robot Framework's keyword-driven approach for test case development.

To represent Robot Framework test cases as behavior trees, we map each keyword to an action or condition node in the tree. The behavior tree structure reflects the logical flow of the test case. For example, an action node means executing a keyword without considering possible return values, such as "Open Notification Overflow Area" in the example. Similarly, if a test case contains a keyword that checks for the presence of a particular item in the Windows system tray, this keyword could be represented as a condition node in the behavior tree. That is, the status of the keyword, Fail or Pass, is passed to the parent node in the behavior tree that continues the execution according to its control behavior, e.g., calling the next child node of a sequence or failing itself. This allows even complex test cases to be specified in a clear and intuitive way, which can improve the readability of the test case. In addition, behavior trees can

Listing 2: Running example specified using RobotBT

```
1     One Should Pass
2     ... - All Should Pass
3     ... - - Wait Until Succeeds 10x 100ms Elem Should Exist
         \${SYS_TRAY_GD_ICON}
4     ... - - Set Global Var  $NotGrouped  \${True}
5     ... - All Should Pass
6     ... - - Elem Should Exist  \${SYS_TRAY_GROUP}  msg="Tray
       icon not present!"
7     ... - - Open Notification Overflow Area
8     ... - - Wait Until Succeeds 15x 200ms  Elem Should Exist
         \${N_OVERFLOW_GD_ICON}
9     ... - - Set Suite Var  $NotGrouped  \${False}
10    ... - - Close Notification Overflow Area
11    ... - Fail  msg="G DATA tray icon not found."
12
```

be extended and modified in a flexible and maintainable way, which can improve the extensibility and maintainability of the test case.

Listing 2 expresses the example using the syntax of RobotBT. For realizing the control flow nodes from Figure 1 we use the two keywords "One Should Pass" and "All Should Pass" that are provided by RobotBT. As Robot does not support keywords that cover multiple lines, the behavior tree is internally represented as a single line. The line breaks only serve for structuring the Robot code visually, which is indicated by "..." in Robot. The indentation of the BT nodes is expressed by dashes in RobotBT. In the leafs, arbitrary Robot code can be used for implementing the actions. Accordingly, the code on the leafs is comparable to the code in the original Robot code. For example, the leafs in lines 3 and 4 of Listing 2, are identical to the lines 1, 3, and 4 in Listing 1. The if-statement in line 2 of Listing 1 is not necessary in RobotBT as the "All Should Pass" node in Listing 2 only continues with line 4 when the keyword in line 3 returned success.

The source code of RobotBT, the full real-world test case from G DATA CyberDefense AG used as example in this work, and artificial examples are publicly available in our GitHub repository [2].

4 EVALUATION

To answer the research questions, we evaluate RobotBT on a real-world test suite from G DATA CyberDefense AG. Their Robot Framework test suite targets GUI testing of their retail antivirus software for major bugs, especially after Windows beta updates. The test suite consists of 10 end-to-end Robot test cases of which the first test case is contained in the RobotBT repository [2].

4.1 RQ 1 – Suitability of RobotBT

To answer RQ 1, we validate whether behavior trees are suitable to express real Robot test cases. We focus on the expressiveness of behavior trees to support real-world test cases, whether the functional behavior of the test cases remains the same, and whether there is an impact on execution times.

4.1.1 Setup. The second author of the publication translates the 10 Robot test cases of G DATA CyberDefense AG into RobotBT test cases. Thereby, we consider two ways to translate the test cases. First, we check whether we can express all parts of the test cases using only behavior trees. Second, to consider the combination of traditional and RobotBT code, as it will most-likely be used in

practice, we selectively translated the parts of the test cases that are most suited for behavior trees. After the translation, we executed all test cases and compared the test results with those of the original test suite as well as the time needed for test execution.

4.1.2 Results. This experiment shows that RoboBT is suitable to express real-world Robot test cases without changing the test behavior nor negatively impacting the execution times.

Expressiveness. We were able to express all test cases using only behavior trees, but also in a combination with traditional Robot code, using behavior trees where they are most appropriate. When translating existing test cases into RobotBT test cases, it is sometimes necessary to rewrite keywords to suit the BT context. For example, the keyword `"Elem Should Exist"` originally returns True or False based on existence, but in BTs a pass or fail of the keyword execution is required instead. Comparing the two translation approaches, we noticed that while it is possible, especially for simple robot code, always using BTs adds overhead. However, the selective use of BTs allows effective specification of test cases.

Correctness. The two created RobotBT test suites can be successfully executed and show no failing test cases. Further, we manually forced test cases to fail, e.g., in the example by executing the test suite when the G Data Security Center is not on the system tray. Also in these cases, the original Robot test suite and RobotBT behaved identical and failed as expected.

Run Time Efficiency. To validate that RobotBT has no negative impact on the execution times of the test cases, we executed the test suite five times using the original Robot test suite and the test suite adapted to RobotBT and measured the average execution times. We observed no significant difference in run-times between the original Robot test cases (17.1s) and the RobotBT test cases (17.06s). Thereby, the standard derivation is relatively low with 0.167 and 0.158 respectively. In conclusion, using RobotBT has no negative impact on the run-time efficiency of the Robot Framework.

4.2 RQ 2 – Comprehensibility

To answer RQ 2 on the comprehensibility of RobotBT, we investigated the practical usability and industry acceptance of RobotBT.

4.2.1 Setup. We conducted one-on-one interviews with 5 developers from 3 different R&D teams of G DATA. The Robot Framework experiences of the participants varied, with one being new to Robot framework tests, while others had several years of experience.

We started with an exploratory code review where we showed the developers the original Robot test suite and the RobotBT versions. We then asked them to provide feedback on the readability, applicability, understandability, and overall effectiveness of using BTs with the Robot framework. Finally, we asked them to rate the readability of RobotBT compared to traditional Robot on a scale from -3 for traditional Robot is much better readable to +3 for RobotBT is much better readable.

4.2.2 Results. According to the interviewed developers there is a significant difference between the test suits. In summary, the use of behavior trees results in a more organized, compact and readable code. All five developers highlighted the possibility of reducing the test code using RobotBT while increasing the subjective readability.

Correspondingly, all developers voted for +3 (RobotBT is much better readable than plain Robot) except for one developer that voted for +2. Altogether, the average vote was 2.8 in favor of RobotBT.

As only drawback they identified the additional effort required to learn about behavior tree nodes, which is relatively low. They even suggested the integration of RobotBT into the Robot core features. For future improvements, they noted that a graphical support by the integration would make it easier to understand and debug the test cases and a drag-and-drop programming environment for BTs would be particularly helpful in developing new test cases.

Overall, the results of the interviews were positive and the developers found the integration to be easy to use. They approved that the readability increases and the will be more organized, which in turn increases the understandability. They were all sharing the same opinion that this feature is useful and delivers benefits.

5 CONCLUSION

The Robot framework is widely used for test automation, but practical experience shows that test case specifications quickly become unreadable. In this work, we have shown how behavior trees can be used to improve the specification of Robot test cases. We introduced RobotBT, a Robot library that allows the specification of Robot keywords using behavior trees. In a practical evaluation, we used a test suite of G DATA CyberDefense AG as an industrial case study to show that RobotBT is suitable for expressing real-world test cases. In addition, we conducted interviews with developers who confirmed the improved readability and practical applicability of RobotBT. In the future, the Robot Framework could be directly extended with the behavior tree functionality of RobotBT. We will also investigate graphical editing support for RobotBT.

REFERENCES

[1] 2023. Py-Trees. https://py-trees.readthedocs.io/en/devel/
[2] Noubar Akopian et al. 2023. GitHub Repository of RobotBT. https://github.com/noubar/RobotFramework-BehaviorTreeLibrary
[3] Thorsten Berger Andrzej Wasowski. 2023. *Domain-Specific Languages: Effective Modeling, Automation, and Reuse* (1 ed.). Springer Cham. XVI, 486 pages.
[4] Michele Colledanchise and Petter Ögren. 2017. How Behavior Trees Modularize Hybrid Control Systems and Generalize Sequential Behavior Compositions, the Subsumption Architecture, and Decision Trees. *IEEE Trans. Robot.* 33, 2 (2017).
[5] Michele Colledanchise and Petter Ögren. 2018. *Behavior Trees in Robotics and AI.* CRC Press.
[6] Davide Faconti. 2018-2023. BehaviorTree.CPP. https://behaviortree.github.io/BehaviorTree.CPP/BT_basics/
[7] M. Gärtner. 2012. *ATDD by Example: A Practical Guide to Acceptance Test-Driven Development.* Pearson Education.
[8] Razan Ghzouli, Thorsten Berger, Einar Broch Johnsen, Swaib Dragule, and Andrzej Wasowski. 2020. Behavior Trees in Action: A Study of Robotics Applications. In *International Conference on Software Language Engineering (SLE).* 196–209.
[9] Razan Ghzouli, Thorsten Berger, Einar Broch Johnsen, Andrzej Wasowski, and Swaib Dragule. 2023. Behavior Trees and State Machines in Robotics Applications. *IEEE Trans. Software Eng.* (2023).
[10] Pekka Klärck, Janne Härkönen, et al. 2023. RobotFramework. https://robotframework.org/
[11] Peter A. Lindsay, Sentot Kromodimoeljo, Paul A. Strooper, and Mohamed Almorsy. 2015. Automation of Test Case Generation from Behavior Tree Requirements Models. In *ASWEC.* 118–127.
[12] Malte Lochau, Sven Peldszus, Matthias Kowal, and Ina Schaefer. 2014. Model-Based Testing. In *SFM.* 310–342.
[13] Dan North. 2006. Introducing behaviour driven development. http://dannorth.net/introducing-bdd/
[14] Yunqiang Yan and Siyou Ma. 2018. Collaborative Simulation Testing with State Behavior Tree. In *QRS.* 456–462.