3rd IEEE International Conference on Robotic Computing
IEEE IRC '19

VizRob: Effective Visualizations to Debug Robotic Behaviors

Miguel Campusano and Alexandre Bergel
ISCLab, Department of Computer Science (DCC), University of Chile
{mcampusa,abergel}@dcc.uchile.cl

***Abstract*—Building and debugging robotic programs is known to be difficult. The robotic community has produced numerous tools, APIs and middlewares to help debug and trace the construction and execution of robotic behaviors. However, most of available debugging tools are text and log-oriented, leading to a tedious and ad-hoc debugging activity.**

In this paper we fully describe *VizRob*, a tool to debug robotic behaviors using logs and execution time. VizRob produces interactive visualizations built from log traces within a state machine model, that is, the visual representation of the behavior. VizRob is founded on deficiencies we empirically found from semi-structured interviews and a revision of tutorial materials. A small case study received an initial feedback of VizRob in a robotic software engineering team. Our case study shows: (i) VizRob helps engineers solve intricate debugging scenarios and (ii) engineers perceive VizRob as filling a relevant gap within the current tools for building robotic behavior.

I. Introduction

Robotic behaviors are commonly implemented using large and complex pieces of software. Although the robotic community has produced efficient APIs and middlewares to *build* robotic systems, there is a still a need for adequate tools to comfortably *debug* those systems.

One important aspect of developing a robot is building its behaviors. A behavior can be seen as a reactive problem and can be modeled using nested state machines. A survey taken by RoboCup participants [1] shows that state machines are one of the most affected algorithms when the robot has problems.

To understand the limitations of current debugging techniques, we reviewed prominent learning materials of ROS [2], the de-facto standard of robotic development. We then surveyed three robotic software engineers. This step is key to identifying patterns and tools commonly used in robotic software debugging. Subsequently, we employ these patterns to design a set of visualizations of ROS logs. We produced three visualizations, each showing a particular aspect of the logs, including logs' severity and frequency. These visualizations are packaged in a tool called *VizRob*. We also provided an implementation of VizRob for SMACH [3], a widely used framework to build robotic behaviors.

We receive initial feedback of VizRob from six robotic software engineers. Our results show that VizRob is key to solving complex debugging situations and it is positively perceived by the experiment participants. In particular, all of them acknowledge that our approach is relevant, highly promising and significantly reduces the effort to process logs compared to textual tools.

Contributions. The paper makes the following contributions:

- A description of VizRob, a set of high-level visualizations of logs and execution time in nested state machines;
- Description notes of the design decisions on VizRob;
- A first small case study to gather perception and usability of VizRob.

Outline. Our paper is structured as: Section II reviews learning material of ROS and SMACH and surveys three robotic software engineers. The section also formulates a set of research questions considered in this paper. Section III presents related work. Section IV describes the visualizations embedded into VizRob. Section V presents design notes of VizRob. Section VI details the case study we conducted to obtain an initial feedback of VizRob. Section VII answers our research questions. Section VIII presents the threats to validity of the case study. Section IX concludes and presents our future work.

II. Current Debugging Practices and Research Questions

We conducted an informal survey of debugging and logging techniques for robotic behavior. We reviewed learning and teaching material related to ROS as well as surveyed three professional programmers (Section II-A). We use results from this effort to formulate a set of research questions (Section II-B), which will be considered to design VizRob.

A. Looking at common logging and debugging problems

To understand how developers use logs, we:

1) analyzed several robotic programs. All of them are programs used in tutorial materials for ROS.
2) surveyed three robotic software engineers working in robotic behaviors.
3) contrasted engineers' answers to what we found in the tutorial materials. We found recurrent situations related to debugging robotic behaviors using logs.

The analysis of ROS learning material and programs indicates that the most important logging practices are: *marking critical events on the robot* (*e.g.,* low battery), *indicating value of variables* (*e.g.,* battery level, distances), *values of published variables*, *value of received variables* (*e.g.,* text heard, pose of the robot), *marking part of the program* (*e.g.,* starting to move), *marking the part where the program failed* (*e.g.,* the program did not receive a value, it did not plan a path). These situations are recurrently employed as "typical situations" for which a logging facility should be employed.

The results of our analysis is similar as to the results by Valdman [4] for generic log files. He stated that logs often show values of variables (*e.g.,* battery level) and that logs present information that developers consider important (*e.g.,* low battery, marking part of the behavior of the robot).

We contrasted what we found in learning materials with the survey of three software robotic engineers involved in robotic behavior from different robotic teams. We found the following:

- The three engineers use the ROS logs system.
- They only use `print` when they debug. After finishing the debugging process, those `print` instructions are removed.
- They use the ROS logging facility in a similar fashion as provided in the ROS learning materials.
- Logs are the most common way to understand what the robot is doing, but it is not the only way. We found other ways to introspect the robot behavior: using the lights of the robot or making it speak to indicate some aspect of its behavior. Another way is by using the `ipdb` debugging tool and analyzing saved executions a posteriori.
- Logs are complemented with dedicated tools (`rqt_console`, `swri_console`, ANSI colors on terminal).

Participants emphasized that logs are helpful but rarely sufficient alone, and some errors are very difficult to reproduce. The tools *rqt_console* [5] and *swri_console* [6] are sophisticated, general purpose GUI to filter and navigate ROS logs, which do not take account the domain of robotic behaviors.

The survey of Wienke and Wrede [7] shows developers also use special purpose visualization tools to monitor systems. They also state that developers mostly use `printf` or log file as debugging tools for robotic systems, while they sometimes use debuggers such as `gdb`.

All the information obtained in this section, including the data, the programs we analyzed and the survey is available online for reproducibility [8].

B. Research Questions

Based on the debugging practices we observed, we designed the VizRob debugger for robotic behavior. We formulated a set of research questions we wish to answer:

- **Q1:** Can developers locate the causes of faulty behaviors?
- **Q2:** Can developers locate the value of the variables when the program is running? Can developers locate the possible variables that produce faulty behaviors?
- **Q3:** Is the program more error-prone in their critical parts?
- **Q4:** Can developers understand the causes of faulty behaviors?
- **Q5:** Is VizRob faster at locating faulty behaviors than using conventional debugging methods?
- **Q6:** Is VizRob faster at locating the value of the variables that produce a faulty behavior than using conventional methods?
- **Q7:** Is VizRob faster at locating the critical parts of the program than using conventional methods?
- **Q8:** Is VizRob faster at understanding faulty behaviors than using conventional methods?

Our first four research questions focus on understanding errors of faulty behaviors, while the last four focus on the time of understanding these errors of faulty behaviors.

We define the *critical parts* of a program as the part of the program where, in its execution, produces the most logs and expends most of the execution time. Also, in these parts the program produces logs of a level of severity that developers should be concerned about, such as warning and error logs.

III. Related Work

In the previous section we mentioned *rqt_console* [5] and *swri_console* [6]. Both are sophisticated log listings for ROS systems. The *swri_console* has several log filters to focus on the important logs. However, both tools are for generic purposes, not being able to use the domain of robotic behaviors.

SMACH [3] is an API for nested state machine popular with ROS. SMACH provides a run-time visualization that shows the machine, the current executed state of that machine and several variables that are passed between states.

FlexBe [9] is a tool to create behaviors using visual programming, auto-generated code and already coded states. It also monitors and supports modification of the behavior at runtime. Because it uses visual programming, it also provides a visualization of the states and transitions of the built machines.

LRP [10] is a live programming language for behavior-based robots. In LRP, developers can build behaviors on the fly, *i.e.,* they can change the behavior of the robot while the robot behaves. LRP provides a visualization of the state machines that changes dynamically when the code is changed.

All these tools offer high level, run-time visualizations of state machines. However, these visualizations do not allow visualizing state machines produced by other tools. On the other hand, VizRob has an open API for developers to use other tools.

Moreover, the mentioned tools only visualize the machines without additional information. This reduces the feedback the tool gives to developers. VizRob integrates more than the machine visualization, *i.e.,* logs and execution time, providing more feedback than the previous tools.

There are visualizations for log files depending on the domain of the execution, for example, ASTRO [11] is a tool that visualizes log files produced by unit tests. This tool allows developers to see more than just passed or failed tests.

Augmented Reality has been used to debug the data captured by the robot, comparing this data against the real world [12], [13]. This helps developers to understand if the captured data is precise. While these tools analyze the data of the sensors, VizRob uses logs information on the domain of robotic behaviors to show developers where, in the program, the behavior may be failing. Both techniques are complementary, using both may give even more feedback to developers.

IV. VizRob

We use a graph metaphor to represent state machines and use polymetric views to represent metrics [14]. Section IV-A presents the common features of all VizRob visualizations.

Sections IV-B – IV-E describe our visualizations. Section IV-F discusses the navigations between visualizations. VizRob is an artifact available online[1].

As far as we know, there are no available visualizations that help developers to understand how are logs being produced in a nested state machine type of program. We used VizRob for that purpose.

The core of VizRob is independent of the robotic behavior API. VizRob is able to work with any API if a suitable bridge between VizRob and the API is built. Also, the API itself should follow the nested state machine paradigm. VizRob generates the visualizations automatically with the provided information passed by this bridge. VizRob is bridged with SMACH, a popular API to build robotic behaviors in ROS.

In this section we present the visualizations provided by VizRob. We use a routine to grab an object using an artificial robot arm as our running example throughout this section.

A. Common Features

VizRob presents three distinct visualizations with common threads. Each visualization represents a state machine with nodes that represent states and directed edges are their transitions. A green border around a state indicates the presence of a nested machine. The machine name is located at the top-left corner. The names of the state are displayed with a pop-up when the cursor is over one of the states. The polymetric information of each visualization is presented in a label.

In all visualizations, states are represented as a box to which metrics are vertically and horizontally mapped: the *height* of a box represents the execution time (*i.e.,* the time of execution inside the state), and the *width* represents the number of logs (*i.e.,* the number of logs produced while executing this state).

An example of one of these visualizations for grabbing an object can be seen in Figure 1 (more information on this particular visualization is shown in Section IV-B). We see that the state machine contains three nested machines and a seemingly important state that has 5 incoming transitions.

Every visualization has the option to remove and show the states that are not being executed during the robot execution. This is crucial for indicating the coverage of the robotic behavior during its execution. The transitions represent static relations: they are shown if they connect the executed states, even if the transition is not triggered.

In Figure 2 we see the same execution presented in Figure 1, but with the option to only show the coverage of the behavior execution. While in the whole visualization we identified a state that seems important because it has 5 incoming transitions, we can see that this state is not even executed.

B. Types of Logs

The *types of logs visualization* associates states to the severity of anomalies found in the logs. States are colored in yellow, red or dark gray. When there is at least one warning log in a state, this state is yellow. A state with at least one error log is red. When there are warning and error logs in the state, red has priority. Dark gray is the default state color, without any warning or error log produced by the state.

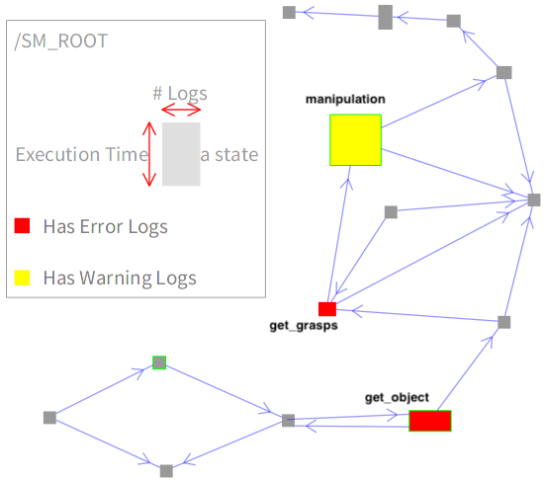

Fig. 1. Type of Logs Visualization. The big yellow state is called *MANIPULATION*, the biggest red square is called *GET_OBJECT* and the smallest red square is called *GET_GRASPS*.

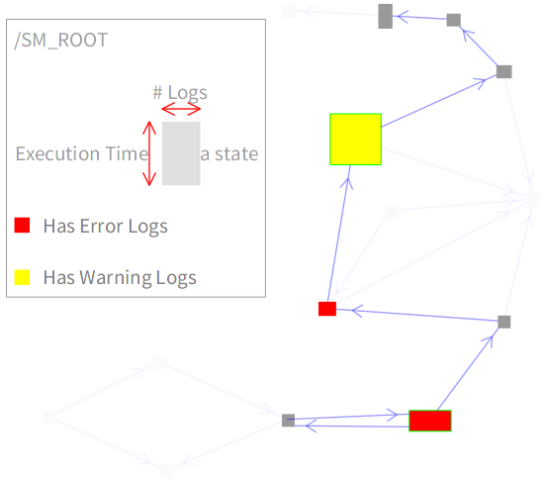

Fig. 2. Figure 1 while turning on the coverage option. All the states that do not execute in this example disappear.

Figure 1 shows the visualization of our running example. In the visualization there are two prominent and large states. The large yellow state, named *manipulation* in our example, triggers, at least, one warning log and takes a significant amount of time to execute. The wide red state, named *get_object*, triggers at least one error log while its execution do not take much time since it is short. This state has a green border, indicating it contains a nested machine. Both states have a considerable amount of logs of any severity. The state *get_grasps* also produces error logs since it is red, takes very little time to execute and does not produce a significant amount of logs, since it is very small.

[1] http://mcamp.github.io/VizRob

C. Error Logs

It is relevant to highlight logging data indicating erroneous behavior. The *error logs visualization* uses a dedicated color mapping. The number of error logs emitted from the execution of a state is linearly mapped to a gray-to-red fading. In this visualization, a red box indicates the state that emitted the largest number of error logs, while a gray state is the state with the least number of error logs.

Figure 3 illustrates the error logs visualization. We see that *get_object* is not the state that triggers the most error logs, even if it triggers the highest number of logs of any severity. The small red square is named *get_grasps* and deserves to be carefully considered by the robotic software developers.

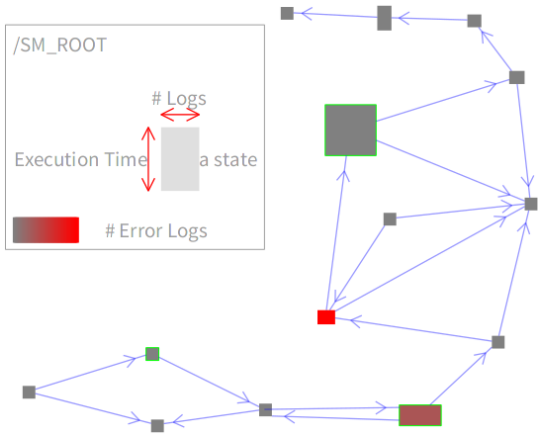

Fig. 3. Error Logs Visualization. In this example, the *GET_GRASPS* state generates more error logs than the *GET_OBJECT* state, even when the last generates more logs in general.

D. Frequency

Determining the number of times a state is executed is relevant when debugging. The *frequency visualization* represents the number of times a state is executed using a white-to-black fading.

In this visualization, a state is colored black if it has many executions and it is light-gray with very few executions. The number of executions of a state may vary from the time of the execution. For example, while a state can be executed for a long time, but only once, another state can be executed a short time, but many times. Figure 4 presents this visualization.

This visualization clearly distinguishes the states that are executed from the ones that are not executed. The white ones are not executed. The black ones are executed the same number of times. Note that in this execution, there are no gray states that are executed an intermediary number of times.

E. Logs Listing

Each of the previous visualizations are interactive. Clicking on a state lists all the logs associated with the state (Figure 5). Moreover, the exact location in the source file that triggered the log is also given. This list of logs offers the classical navigation and filtering options found in common debugging tools. The

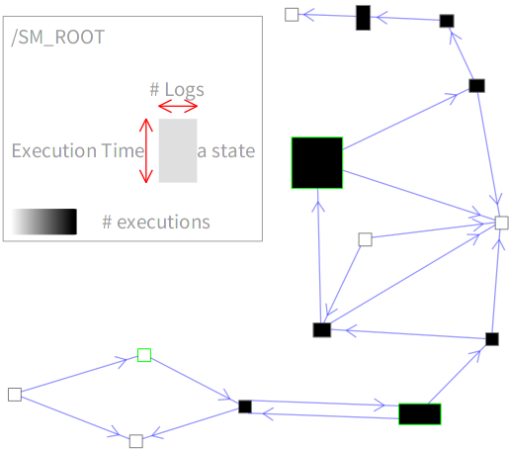

Fig. 4. Frequency Visualization. In this example, all the white states are not executed. When turning on the coverage visualization, we can see that all states not shown in Figure 2 are the white states of the frequency visualization.

listing provides two filters: one for the severity and another for the name of the program where the log is produced[2].

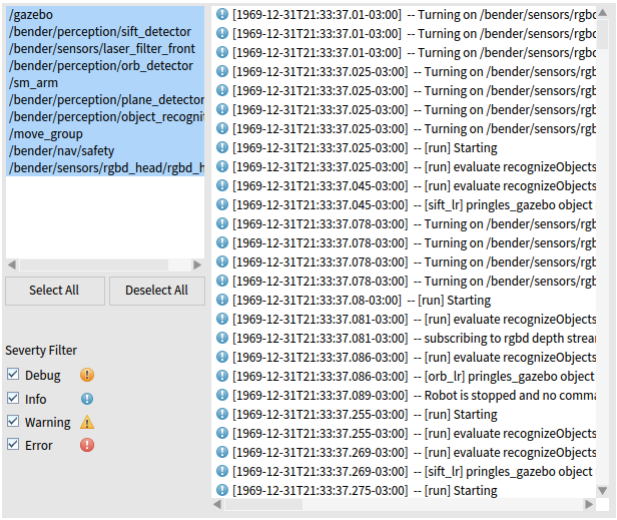

Fig. 5. Logs Listing. At the left we see the filter options: name of the program and severity. At the right we see the list of all the logs produced on the execution of this behavior. An icon appears on the left of the text of the log representing the severity of the log.

F. Navigation

Details of all visual elements may be obtained on demand. Clicking on an element opens new visualizations, leading to a chain of visualizations: the previous visualization remains accessible and interactive. Clicking on a state with a nested machine reveals the nested structure, with the option to use any of the previous visualizations. When a log is clicked, it is possible to see the attributes of the logs, such as the name of the program, severity and/or message. Figure 6 exemplifies the recursive behavior of VizRob.

[2]In the case of ROS, the name of the program is the name of the Node producing the log

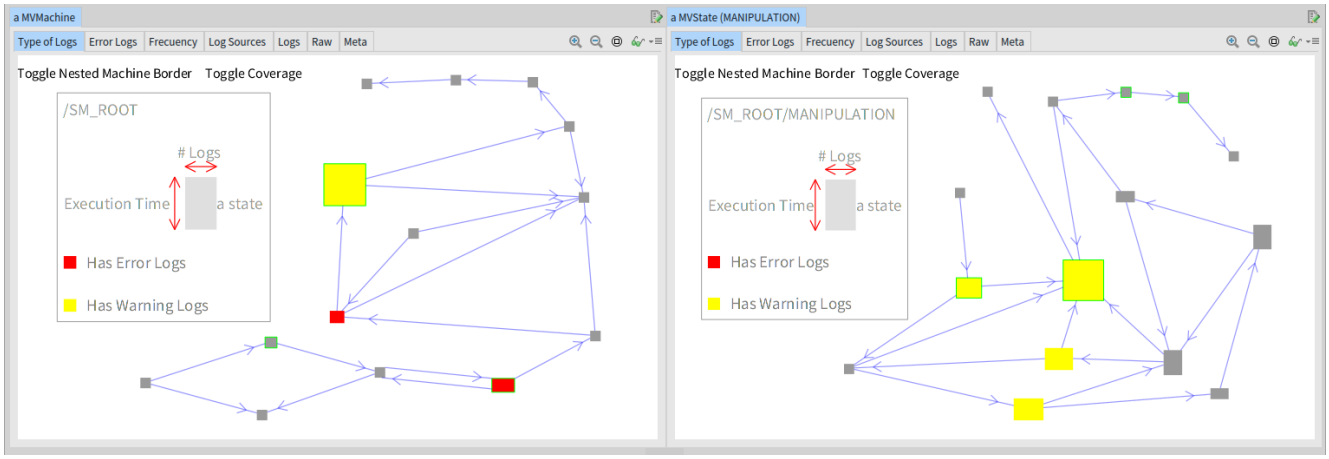

Fig. 6. Navigation of the Visualizations. In the left visualization we have the root machine. The right visualization represents the nested machine of the state represented with the big yellow square of the left visualization, the *MANIPULATION* state.

V. VizRob Design Notes

VizRob's core is designed around the concept of nested state machines, this allows VizRob to be independent of the robotic behavior APIs. It shows run-time information of a nested machine program: logs and execution time. It is important to simulate an execution of the behavior of the program in VizRob to get the right data to show in the visualizations.

VizRob generates its own machines, states and transitions by taking into account the information of the robotic behavior API. A *Machine* object contains all the *State* and *Transition* objects of that machine. For nested state machines, a state may have a unique machine instance.

Because of the run-time nature of VizRob, one of the most important aspects is how it saves the run-time information. VizRob saves the state executing of the machine, *i.e.,* the current state. Whenever a state is executing, VizRob creates a single *Status*. A Status is the representation of the run-time information of a state. Here, VizRob saves all the logs that are produced in that point of the execution. VizRob also saves time when the state stops being active. With this, VizRob gets the execution time of that status.

Moreover, a state can be active any number of times in the same execution of the program (*i.e.,* with a loop of states). For this reason, every state has a collection of statuses. The sum of all status times is the total execution time that is shown in the parametric views of VizRob (the parametric views explained in Section IV). The number of executions visualized in the frequency visualization (Section IV-D) is the amount of statuses that a state has.

VizRob allows for a machine to have more than one active state at the same time, which allows for concurrent states on the execution of a machine. In this case, the logs produced by the program are assigned to all the active states. We cannot be sure in which particular state the log is being produced.

For a machine, it groups the information already presented in states. For example, the logs of a machine are all the logs of all the states of that machine.

VizRob can be fed data automatically using a bridge that connects it with the desired robotic behavior API. The API should offer a way to expose relevant information about the static and dynamic configuration of the state machine. In particular, we designed a bridge that connects VizRob with SMACH [3], a popular ROS API to develop robotic behaviors. By using an extra interface shown in Figure 7, VizRob connects with the SMACH program, receiving the data while the program is running. The data that VizRob collects is: the static structure of the machine, the current states of the running machine and the logs of the program. VizRob is fed with the data of the SMACH program while the SMACH program is running, this allows assigning the program logs to the running states and setting the execution time and the number of executions of the states. This data is then processed by VizRob to automatically build the visualizations after the program stops. The program needs to stop for VizRob to collect the final information of the programs (*i.e.,* the execution time of the last active states).

VizRob only shows one execution of a behavior at a time. If VizRob is not stopped at the end of the behavior, it may clash with the information of a new behavior. For VizRob SMACH, to allow the recollection of data for a new behavior, a new connection with SMACH should be created, using a new instance of the VizRob SMACH UI.

VI. Initial Feedback: Case Study

We received an initial feedback of VizRob through a small scale case study involving six robotic engineers (who we refer to as *participants* in the remaining of the paper). The participants are members of the UChile Homebreakers team[3]. All the participants of this team have extensive experience in participating at the RoboCup competition (RoboCup@Home league) and all are experienced with nested state machines. The team built and designed the software aspects of two

[3] http://robotica-uchile.amtc.cl/about.html

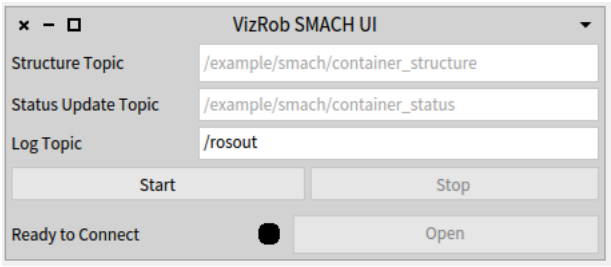

Fig. 7. SMACH Bridge UI for VizRob. Developers need to give the name of the three communication channels for: structure, status update and logs. After starting the recollection of data with the start button, the stop button becomes available. When stopping the recollection of data, the open button becomes available. Clicking on the open button will make the visualizations of VizRob available.

robots: *Bender*[4] and *Pepper*[5].

We also have as a participant a former member of the UChile Robotic Team[6]. This participant has extensive experience in participating at the RoboCup competition, RoboCup-Soccer league. The participant is an expert at building behavior and handing all software aspects of the NAO robot[7].

For the sake of reproducibility, the data produced in this case study is available online [8]. It includes videos, audios and the answers of the pre and post questionnaire.

A. Robotic Behaviors

We visited the UChile Homebreakers team on site and the participants agreed to use VizRob to debug robotic executions, on 2 different robotic behaviors. We asked the participants to use VizRob on faulty behaviors with difficult-to-debug anomalies, but also in seemingly correct behaviors. An apparently correct behavior may still have errors during its execution.

The first behavior is *Speech and People Recognition (SPR)*, which consists of 3 routines: categorizing people in a crowd (*e.g.,* determining gender and age); then, an operator asks questions about the crowd; finally, the robot is surrounded by people and they take turns asking the robot questions.

The second behavior considered in our case study is *Grasping*. In this task, Bender positions itself in front of a table. It recognizes an object located on the table and grabs it.

The participants employed VizRob on 3 different executions of these 2 different behaviors.

B. Problems found

In this case study all participants should find problems of a particular behavior in 3 executions of it. We label the participants according to the behavior they debug and the team they belongs to. For the behavior we use *S* for SPR and *G* for Grasping. For the team we use *H* for UChile Hombreakers team and *R* for UChile Robotic team.

For the SPR behavior we had 3 participants labeled SH1, SH2 and SH3. Two of them have a knowledgeable experience using SMACH and one of them has an advanced experience (within the scale: None, Beginner, Knowledgeable, Advanced and Expert). All are undergraduate students with an experience ranging from 1 to 3 years using SMACH.

For the Grasping behavior we had 3 participants labeled GH1, GH2 and GR3. GH1 and GH2 considered themselves to have a knowledgeable experience using SMACH (within the same scale as before), while GR3 do not have experience in SMACH. However, GR3 is an advanced robotic developer with more than 5 years of experience and 3 to 5 years of experience using ROS. GH2 is a Master student and GR3 is a professional holding a Master degree.

Depending on the execution of the behavior, participants found different types of problems in the executions thanks to VizRob. We list some of the techniques using VizRob that the participants employed to find these problems:

Error logs are important: The red states on the visualization are important, they show problems for all the five executions that they appeared.

Warning logs (in big states) may be important: There are several problems that became apparent to participants thanks to the yellow color on states for warning logs, especially when there were a lot of logs produced on that state. This is shown in the Type of Logs visualization with big yellow squares and they were relevant in the 2 executions where they appeared. The problems that became apparent at least need to be checked to see if they are important or not, as stated by GR3.

Similar states group errors: In one of the executions of the grasping behavior, GH2 notices a problem that was visible, thanks to a group of yellow states that were connected to each other. This group of yellow states shows several warning logs that are connected within the same problem.

Frequency visualization group several states: Connected with the previous problem, after finding all the yellow states, changing to the frequency visualization, GH2 notices that the errors not only were connected, but happened multiple times. The frequency visualization shows states with a lot of executions, indicating that the robot tried several times to grab an object, but finally give up.

Problems with variables: In 2 of 3 executions of the SPR behavior, there were problems of not filling or wrongly filling variables for the SMACH API. This is found by the three participants of the SPR behavior, thanks to a red square presented in the type of logs visualization. In one of the executions of SPR, SH3 believes this error is responsible for making the robot incorrectly answer questions, even when the execution of the routine to answer questions is several states ahead of the state where the error appeared.

Seemingly correct behaviors can have problems: One of grasping behavior executions performed correctly, resulting in the robot grabbing the object. However, the three participants notice several problems that may affect the robot in the future. These problems are apparent in the type of logs visualizations with yellow and red states.

[4] http://robotica-uchile.amtc.cl/bender-robocup.html
[5] http://robotica-uchile.amtc.cl/pepper-robocup.html
[6] http://robotica-uchile.amtc.cl/about.html
[7] https://www.softbankrobotics.com/emea/en/nao

C. VizRob vs participants' debugging tool

In robots like these, it is common to have several open terminals that execute different aspects of the robot. All participants stated they only use the terminal where they executed the behaviors to debug and not the other terminals that executed other aspects of the robot. They look at the logs in that terminal to see what is happening.

When the error of a behavior crashes the system, it is easy for the participants to find it because the last log of the usually terminal points to the crash. However, SH3 mentioned that VizRob helps to better localize the cause of the problem, because an error may be triggered way before it manifests in the execution. With VizRob, the participant can see that the program seemed fine until the error, reducing the debugging time for these kinds of errors. SH1's conclusion is similar: any other tool to debug is for generic purposes and they spend too much time looking for the right information.

The participants believe they can find almost all the errors they found with VizRob by using the terminal. However, all of them believe it is difficult and time consuming using the terminal compared with our tool. Within the terminal it is difficult to find the right log because of its information overload. These errors can be found by tools like *rqt_console*. However, with this tool they still may have a problem locating the root of the error if they do not see where the error is in the execution, as we previously mentioned.

Some errors can be found just by being present in the task at hand. If the robot is incorrectly answering a question, then it is easier to hear that the robot is performing poorly instead of looking at logs in any kind of sophisticated tool. However, with VizRob, SH3 found the cause of why the robot was answering incorrectly. The participant believes he could not reach the same conclusion by solely using the terminal.

In seemingly correct behaviors, participants told us they mostly ignore if there are problems because the robot behaves correctly. As we mentioned in the previous section, VizRob helps developers find problems even in correct behaviors, while they did not even try to find those errors in the terminal.

D. Use of VizRob

Five of the six participants first used the coverage option to find what the last state of the execution of the behavior was, believing they would find the error there.

After finding the last executed state, all the participants used the Type of Logs and the Error Logs visualizations to have a general view of the problem and click on the state that may have a problem. GH1 told us that these visualizations help him see if the machine is having other problems or if the current error stems from an earlier problem in the execution of the machine.

Three of the participants found VizRob extremely useful (on a scale of "not useful", "somewhat useful", "useful", "extremely useful"), while the other three found the tool useful. All of them wanted to adopt VizRob on the debugging of robotic behaviors. Two of them would use VizRob to solve issues in complex robotic behaviors.

GH1 told us that many events occurred while executing a robotic behavior. However, most of these events are not noticeable in the same terminal where the behavior of the robot runs. For example, it is not possible to see low-level hardware logs in the terminal of the behavior of the robot. VizRob is considered to be a significant change in that respect.

Moreover, robots are known to have problems of uncertainty and these problems may affect the robot in future executions of the same behavior, as stated by GR3. GH2 also stated that if there are problems that do not affect the execution of the machine in an isolated context, it may affect the execution of more complex behaviors when using this machine with others. Both participants agree that VizRob may reduce these issues because, with VizRob, they found problems even in a seemingly correct execution.

VII. Answering our Research Questions

Taking into account the previous experiment, we answered the eight research questions listed in Section II-B:

Q1: positive answer. The participants found several problems in all the executions, even with the execution that worked seemingly fine. VizRob helps them in finding faulty behaviors.

Q2: negative answer. VizRob does not help developers in finding the value of variables and how they change in time, nor in correct and faulty behaviors. Our best guess is that in the executions at hand, the developers do not use or only use a small number of logs to show the value of the variables.

However, participants still see the value of some variables that were useful. Nevertheless, these kinds of logs do not seem prominent in executions of the robot in the case study.

Q3: no answer. Even when the participants found more errors in the critical parts of the robot, we are not sure if in these parts there are more errors than other parts. Nevertheless, we show that critical parts are important places in the behavior of the robot and they need to be analyzed.

Q4: positive answer. The participants not only found the place where the faulty parts of the behaviors were, but also they gave several insights about why the robot may be failing. One participant stated this when he told us that the tool helped them to better pin down the possible causes of the faulty behavior.

Q5, Q6, Q7, Q8: positive answer, but only based on participant perception. In all the questions related to time, all participants considered this tool much faster to find bugs and therefore, to fix them rather than use their conventional methods. This is an important statement, however, it cannot be verified without a rigorous comparison.

VIII. Threats to Validity

In this section we present the most important threats to validity for our case study.

Ready to use VizRob. All participants found VizRob more comfortable to use rather than using their normal way of debugging. However, one of them explicitly told us that if VizRob is not as easy to install as if it is to use, he may not find it as comfortable as he stated.

Nevertheless, we do not want to measure that part of the development (even as important as it is), we only want to measure the effectiveness of VizRob in finding errors in the behavior. With that in mind, the participants found a significant number of errors using VizRob and they stated that VizRob helps them in locating the errors in an accurate and faster fashion.

Lack of comparable baseline. Because we conducted a small scale case study, we do not have a satisfactory baseline to compare to VizRob. The participants do not use sophisticated tools to debug programs, they only use the terminal. We therefore expect better results using VizRob.

Nevertheless, our case study is not about comparing VizRob against the terminal, or any other debugging tool for now. In this work we want to show how participants behave when using VizRob. Also, we want to show that the participants can find errors using VizRob without a long previous training.

IX. Conclusions and Future Work

In this paper we presented VizRob, a debugging tool for robotic behaviors. This tool incorporates metrics based on logs and execution time into several visualizations of nested state machines, a classical paradigm to represent robotic behaviors. Developers can navigate into the machines and states to see focused information of that particular element. Developers can look at the overall information or immerse into the behavior and focus their attention on a particular part of the system.

VizRob is fed on the fly by developers. In particular, it can be fed by running programs with a proper bridge between the program and VizRob. Using our bridge between SMACH and VizRob, developers can feed VizRob with behavior-based programs using SMACH.

We received initial feedback on VizRob using a small scale case study among six software engineers from the UChile Homebreakers team and the UChile Robotics team. Our preliminary findings show that participants find several errors in robotic behavior programs and they suggest solutions for these errors, even without looking at the source code of the programs. VizRob is positively perceived by the participants, and in particular, all of them want to adopt this tool in the future when they program robotic behaviors.

However, VizRob intensively uses the logs of the system. If the logs of the system are insufficient, VizRob may be insufficient for developers too. Nevertheless, VizRob still uses information of execution time of the program, showing developers where programs expend more of the time. This may be useful for developers, even for them to notice where the program needs more logs. This is an interesting approach and worthy of a future study for our tool.

VizRob has two main lines for future work. First, with our preliminary findings we need to conduct more experiments to measure the real value of the tool. In this work we have positive findings based on the qualitative information that participants gave us. It is important to measure the value of VizRob in an experiment with quantitative value.

Second, there are several ways to improve the usability of VizRob. VizRob is fed at run-time because we plan to make visualization of running nested state machine programs. This is, VizRob will become available at the same moment it is receiving data, not only a posteriori.

We also expect VizRob to simulate the execution of the program. In this simulation developers could navigate the visualizations of the state machines at any point on the execution of the program, not only at the end. This will allow them to have much more information of the behavior of their robots at any point of time.

Acknowledgment

We would like to thank the following colleagues for the semi-structured interview that helped us shape VizRob: David Conner, Loy Vanbeek and Cristopher Gómez. We thank the UChile Homebreakers and the UChile Robotic teams for helping us evaluate VizRob. We thank LAM Research for partially sponsoring the effort described in this paper. Finally, Miguel Campusano thanks Conicyt[8] for his funding via CONICYT-PCHA/Doctorado Nacional/2015-21151534.

References

[1] Gerald Steinbauer. A survey about faults of robots used in RoboCup. In *RoboCup 2012: robot soccer world cup XVI*, pages 344–355. Springer, 2013.

[2] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. ROS: an open-source Robot Operating System. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009.

[3] Jonathan Bohren and Steve Cousins. The SMACH high-level executive [ROS news]. *IEEE Robotics & Automation Magazine*, 17(4):18–20, 2010.

[4] Jan Valdman. Log file analysis. *Department of Computer Science and Engineering (FAV UWB)., Tech. Rep. DCSE/TR-2001-04*, page 51, 2001.

[5] ROS. The rqt_console. http://wiki.ros.org/rqt_console. Accessed: 13-02-2018.

[6] Southwest Research Institute Robotics. The swri_console. https://github.com/swri-robotics/swri_console. Accessed: 13-02-2018.

[7] Johannes Wienke and Sebastian Wrede. Results of the survey: failures in robotics and intelligent systems. *arXiv preprint arXiv:1708.07379*, 2017.

[8] Miguel Campusano and Alexandre Bergel. VizRob's Initial Feedback: Case Study, November 2018. DOI: 10.5281/zenodo.1476230.

[9] Philipp Schillinger, Stefan Kohlbrecher, and Oskar von Stryk. Human-Robot Collaborative High-Level Control with an Application to Rescue Robotics. In *IEEE International Conference on Robotics and Automation*, Stockholm, Sweden, May 2016.

[10] Miguel Campusano and Johan Fabry. Live Robot Programming: The language, its implementation, and robot API independence. *Science of Computer Programming*, 133:1–19, 2017.

[11] Diego Castro and Marcelo Schots. Analysis of test log information through interactive visualizations. In *Proceedings of the 26th Conference on Program Comprehension*, ICPC '18, pages 156–166, New York, NY, USA, 2018. ACM.

[12] Toby Hartnoll Joshua Collett and Bruce Alexander Macdonald. An augmented reality debugging system for mobile robot software engineers. *JOURNAL OF SOFTWARE ENGINEERING IN ROBOTICS*, 1(1):18–32, 2010.

[13] Patrick Renner, Florian Lier, Felix Friese, Thies Pfeiffer, and Sven Wachsmuth. Wysiwicd: What you see is what i can do. In *Companion of the 2018 ACM/IEEE International Conference on Human-Robot Interaction*, pages 382–382. ACM, 2018.

[14] Michele Lanza and Stéphane Ducasse. Polymetric views—a lightweight visual approach to reverse engineering. *Transactions on Software Engineering (TSE)*, 29(9):782–795, September 2003.

[8] http://www.conicyt.cl