ROSDashboard: a visual debugging tool for robotics

Felix Kaser[1], Bruce A. MacDonald[2], Andreas Angerer[3]

Abstract— **Data collected during debugging is traditionally rendered as text. The special application field of robotics faces problems with this approach, since a typical robotic system constantly gathers and processes data from the surrounding environment through sensors. Robotic applications are also hard to interrupt during debugging, since robots generally don't run in a deterministic and suspendable environment. Developers of robotic applications are confronted with high amounts of data during debugging, which becomes hard to interpret if the data is represented as text and high amounts of data need to be interpreted at once. This paper introduces a new tool to support debugging of robotic applications: It takes into account the special requirements for a debugging tool in a robotic development environment, especially the uninterrupted rendering of debugging data and the need for better visualization of data to support a faster interpretation of data. The tool's goal is to help developers understand the data during debugging more quickly and improve overall productivity during robot development. A graphical user interface was developed where developers can choose how they want to visualize data collected during debugging.**

I. INTRODUCTION

Current debugging tools for robotics are mostly based on debugging techniques and interfaces developed for traditional non-robotic applications. Those techniques and interfaces were developed specifically to suit the requirements of traditional systems: First they assume that a program can be interrupted in its execution, which in a robotic environment is not possible most of the time. Second the data handled in traditional applications is usually discrete and based on user input, as opposed to the sensory data a robot handles. The data a robot handles is in general closely related to the real world environment of the robot and comes from sensors that provide a continuous data stream of readings. Although the computing world has changed in recent years, most tools have not. Robot developers often still use tools which were developed under different circumstances and based on different requirements. The application field of robotics has special requirements for debugging tools which are often not met by current debugging tools.

Traditional debugging tools usually render data collected during debugging as text and it's the developer's task to interpret the data. In a suspendable environment the developer has as much time as they need to interpret and analyse the data. When debugging robotic applications the developer usually has a lot less time, because of the nature of the application: Robotic applications usually can not be interrupted in their execution, because robots generally don't run in a deterministic and suspendable environment [1]. Interrupting a robotic application would destroy the continuity in which the sensors collect data, the environment of the robot would change substantially and thus the robot's behaviour would change, which is an example of the "probe effect" and makes it hard to reproduce a fault unless it has a single cause [1]. It is necessary to collect data during debugging without interrupting the execution of the program. Although some technical solutions have been presented in recent years [1], many developers still rely on printf-style debugging or other logging mechanisms. This approach is much simpler and does not require external tools, but the source code must be modified. Source code modification can be a problem with so called "Heisenbugs", software faults that disappear because the observation affected the bug [2]. The data collected with print or logging statements is usually text-only, which requires the developer to constantly parse and interpret logging messages. Due to the large amount of data that is processed this often means developer consoles are filled with logging messages that often contain more complex content such as numeric data.

ROSDashboard, the tool presented in this work, aims to support the developer during debugging by visualizing data in a graphic way and thus eliminating the cognitive effort needed to parse and interpret text based logging messages. While most of the currently available visualization tools in robotics focus on spatial data to help understand the robot and the environment in which it runs [3], [4], rendering of abstract data is still uncommon. ROSDashboard provides a dashboard interface to robot developers, which they can populate with graphical widgets to visualize all kinds of data from the robot. The dashboard can be customized to display widgets according to the current robot hardware and development stage. It can be used to visualize data during debugging as well as monitor data during the normal execution of the robot. This means ROSDashboard is a tool that a) can be adapted to many different use cases and b) allows the developer to choose the widgets he or she thinks represent the data best, according to their mental model and the meaning of the data. The tool is based on ROS (Robot Operating System) which abstracts from specific robot hardware and takes care of inter process communication [4].

[1]F. Kaser is a student in the Software Engineering Elite Graduate Program at University of Augsburg, Technische Universität München and LMU Munich. kaserf at in.tum.de

[2]B. MacDonald is an Associate Professor at the Department of Electrical and Computer Engineering, University of Auckland, New Zealand. b.macdonald at auckland.ac.nz

[3]A. Angerer is with with the Institute for Software and Systems Engineering, University of Augsburg, D-86135 Augsburg, Germany. angerer at informatik.uni-augsburg.de

II. ROS - Robot Operating System

ROS is an Open Source framework for complex robotic systems which has grown significantly in the last years, has an active community backing the project and supports many of the currently available robots [5]. It was developed to abstract from the hardware of the robot and make it easier to create modular robot software, which can run on different robots and on different machines. The modular approach makes development easier, because the work can be divided amongst different developers or development teams. This also allows the developer to change only small parts of a complex system, without the need to build and re-deploy the whole system. The modules in ROS are called *nodes* and several nodes executed together are called a *stack*. ROS *packages* bundle nodes and stacks and are used to make software modules available to other developers. Everyone can create their own package which can be indexed by ROS so that their software modules can be found, downloaded and used by other developers. There exist many packages, nodes and stacks with implementations of algorithms for some of the most common problems in robotics (e.g. navigation, localization, joint movement, etc.) and they can easily be (re-)used.

The communication between ROS nodes is either asynchronous through a publish/subscribe mechanism or synchronous through services. Nodes can send messages by publishing a message on a topic and receive messages by subscribing to that topic. This mechanism is flexible and decouples the sender from the receiver. A publisher node does not need to know if there are other nodes listening and vice versa. For synchronous communication and guaranteed delivery of messages, services can be invoked. The routing is established during runtime through the ROS core. The communication between nodes is one of the main sources for debugging data when debugging a ROS application. The same communication framework is also used for the logging mechanism in ROS, which publishes messages to the special purpose topic */rosout*.

ROS comes with several tools to assist the developers during the development and debugging of a robot. The tools most relevant to this work are:

- rostopic — a command line tool to monitor topics and publish messages on topics.
- rxplot — a graphical tool which plots data from one or more topic fields on a cartesian coordinate system.
- rxconsole — displays logging messages that have been published on the special purpose topic */rosout*.
- rviz — renders models of the robot in 3D and visualizes spatial data like point clouds, robot poses, trajectories, etc. [4].

III. Related Work

This section first gives an overview of existing visualization tools which are used in robot development frameworks. LabVIEW's visualization tools are presented as an example of a flexible visualization interface which can be used for many different application areas, not only robotics. Finally a novel way to capture data during debugging is presented.

A. Visualization Tools in Robotics

Most of the currently available robot frameworks have their own tools for data visualization. Player/stage [6] contains a tool playerv, ORCA [7] has the tools OrcaView2d and OrcaView3d, CARMEN [8] features a graphical interface called robotgui that enables control of the robot and visualization of data collected through sensors. All of these tools focus on visualization of pre-defined data collected from known interfaces such as laser sensors, mapping algorithms and computer vision modules. They were not developed to visualize arbitrary and abstract data which is often used for debugging.

ROS has two tools to visualize data. The simpler one is rxplot, which plots values as a time series on a cartesian coordinate system. rviz is a more complex tool to visualize spatial data in 3D. It was designed to render 3D models of robots together with data from sensors and algorithms: point cloud data, laser data, marker positions, arm postures, planned paths, etc. rviz has a plugin system which can be used to add further visualizations and modules.

ARDev is an augmented reality (AR) debugging system based on Player/Stage [3]. The AR approach to debugging helps the developer to understand the global state of the robot by augmenting a video feed of the robot with information of the robots' perception. Like the tools presented above, it focuses on pre-defined data such as laser and sonar sensors. The visualization of abstract data has been identified as possible future work [3]. Initial evaluation studies were promising that the visualization can help developers significantly during debugging [3].

3D visualizations such as rviz and ARDev are used during debugging to understand the robot's view of the world. Both of them require a substantial amount of set up work. For rviz a 3D model of the robot must be created, which has the exact proportions of the real robot. ARDev requires an even more complex setup for AR. During the development of the tool an intelligent debugging space (IDS) was permanently installed in a laboratory, where the robot under development can be constantly tracked with markers and cameras [3]. This setup is often unavailable, especially in outdoor situations, and is also often not applicable for debugging tasks where only abstract data is available or required. A general problem with 3D visualization tools is that abstract data often does not fit into the 3-dimensional rendering and in the case of AR can hide important information of the real world [3].

B. LabVIEW

LabVIEW (Laboratory Virtual Instrumentation Engineering Workbench) is a graphical programming environment developed by National Instruments[1]. It features *Front Panels* which allow a developer to add widgets to a graphical interface. These widgets can display the status of the graphical

[1] http://www.ni.com

program or act as control interface (button, knobs, etc.) for the program. For each widget on a panel an element is created in the block diagram of the program which can be connected to other parts in the block diagram. LabVIEW delivers a high number of pre-defined visualization widgets such as text displays, sliders and progress bars, which can be used to create a dedicated user interface for an application. This user interface can be used to control the application as well as monitor and debug the application.

The front panel feature was described as "LabVIEW's biggest advantage," its "best feature," or its "main power" in the open-format answers of a survey of LabVIEW programmers [9]. It would be possible to design concrete front panels also for the purpose of debugging a specific application. This however requires integrating each front panel element directly into the visual program and is thus highly intrusive. Though a more transparent debugging possibility would be desirable, LabVIEW shows the potential of a flexible tool for visualization of abstract data in form of graphical widgets.

C. Tracepoints

The non-interruptible nature of robotic applications is a challenge when it comes to data collection for debugging. Real time constraints make it even more difficult to collect data. Tracepoints have been investigated as a technology to collect data for debugging without interrupting the execution of a program [1]. The tracepoints approach does not require source code modification and can also be used in real time critical systems.

IV. ROSDASHBOARD

The purpose of the tool presented in this paper is to support developers of robot applications during development and debugging. Current visualization tools specialize on visualizations for pre-defined data with a fixed structure, such as laser data, mapping information and path planning. ROSDashboard bridges the gap between existing visualization tools and low level text representations of data such as logging frameworks or print statements. The tool provides a graphical user interface for easy visualization of data streams. It uses the inter process communication platform of ROS to gather data for visualizations. This can be either existing data from module to module communication or new data specifically published for debugging. Although tracepoints could be a data provider for a visualization tool like ROSDashboard, for the scope of the first iteration of the tool we decided to use logging statements to collect data. Since the tool is based on ROS, which is not a real time operating system, the real time constraints were considered out of scope for this work.

The main interface of ROSDashboard is a central dashboard where developers can add visualization widgets through drag and drop from a toolbox. Currently a number of basic widgets have been implemented to visualize data, the tool's design though allows easy integration of more complex widgets. Fig. 1 shows ROSDashboard's main interface with an example configuration of widgets on the dashboard.

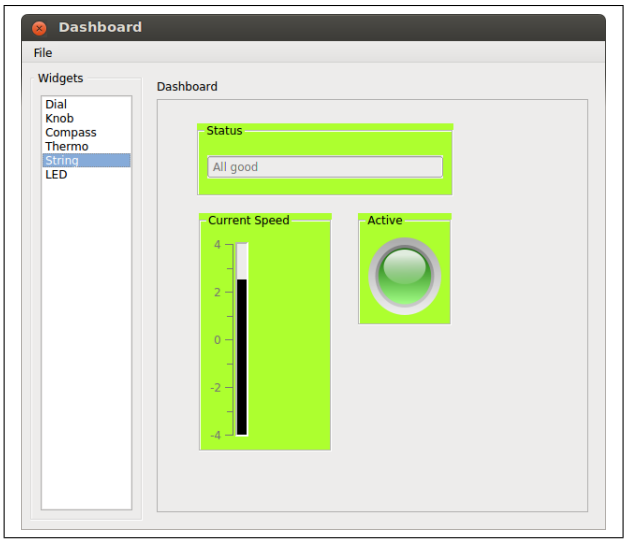

Fig. 1. Screenshot of ROSDashboard's main interface.

A. Requirements

The requirements for ROSDashboard are derived from the special robotics needs for a debugging tool. When debugging robotic applications it is usually not possible to interrupt the execution of the application and follow a step-through approach. All data must be captured, processed and visualized live. The tool is designed for mobile robots, which means the communication is usually distributed across a network.

Due to many different application scenarios in robotics and the diverse environment of available frameworks for robot development, many researchers and developers have built their own tools to support them during debugging [3]. Developing your own debugging tool is extremely time consuming and the developed tools are often one-time-only tools, because they don't fit the use case of other applications and are too hard to adapt to a new project. In order to allow the use of this tool in many different scenarios and use cases, flexibility has a high priority. Flexibility not only means the tool can be adapted easily to fit different problems, it also means the tool can be adapted to suit different developer's preferences. Each developer might prefer a different set of widgets to visualize the collected data, thus loose coupling of the collected data and the visualization widgets is necessary.

Debugging robotic applications is a highly iterative process, where small changes are made and immediately deployed to the robot. The debugging tool should keep the configuration overhead as minimal as possible so that the developer is not distracted from the problem analysis task. Adding and removing new widgets to ROSDashboard should not require many steps, otherwise the developer will be distracted by the configuration task.

B. Design

ROS is used to abstract from the robot's hardware and to decouple the collection of debugging data from the visualization. As explained in section II, the ROS publish/subscribe mechanism is able to decouple the sender from the receiver

of messages. ROSDashboard can access any data published on ROS topics: data that is used to communicate between different nodes and data that is specifically published for debugging purposes. This means ROSDashboard transparently accesses and visualizes data in existing projects without the need to modify the code, if the data is already published.

The current ROS logging framework publishes the log messages on the special purpose topic */rosout*, but converts everything to a String before the transmission. Since those logging messages are text based, valuable information about the type of the data is lost. To solve this problem temporarily, an API was exposed in ROSDashboard to provide a set of convenience methods which can be used to simply publish messages on a topic without loosing the type information. A more permanent solution that allows transparent data collection is discussed as future work (see section VI). If the developer chooses not to use the ROSDashboard API due to the new dependency to ROSDashboard, they can always publish the message manually.

Although the current software does not support third party widgets yet, the internal structure of the widgets is aligned to allow easy integration of a plugin engine in the near future. For now if a third party wants to have specialized widgets, they need to modify the source code and add the code for widgets directly to the source tree. This is possible because ROSDashboard is developed as an Open Source project, the source code is hosted on Github[2].

An early prototype was announced to the ROS community and valuable feedback has been collected and has been taken into consideration: small changes have already been implemented and bigger changes are documented as future work.

C. Implementation details

This section presents important implementation details. First the object model is presented: it has been designed to allow easy integration of a plugin framework in the future (see section VI). Python was chosen as the target language for ROSDashboard, because it is a good match for fast prototyping and iterative development. Python's duck typing ("If it walks like a duck and quacks like a duck, it must be a duck" [10]) was used to minimize the configuration overhead when new widgets are added to the dashboard: Usually a datatype must be chosen, but through topic introspection the type was chosen dynamically (see IV-C.2). Minimizing the configuration overhead has been identified as a core requirement. Qt is used as graphical toolkit for ROSDashboard, which is currently the recommended toolkit for graphical tools in ROS.

1) Object model: The object model (see Fig. 2) was designed to make it easy to extend ROSDashboard with more widgets. The abstract `DashboardWidget` class covers the general tasks that are the same for every widget. Its internal method structure allows subclasses to overwrite only some parts of functionality, without the need to rewrite most

[2]http://github.com/kaserf/rosdashboard

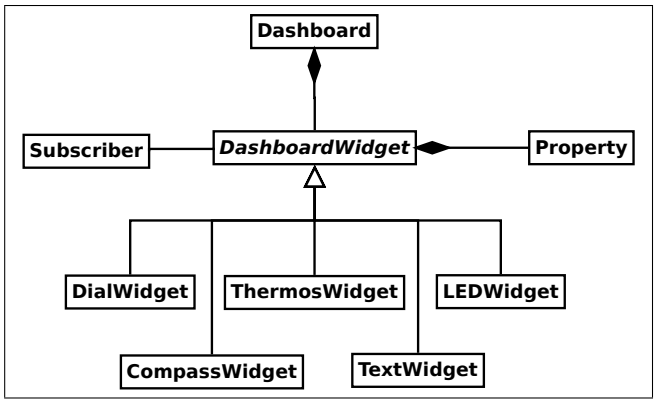

Fig. 2. Overview of the core object model structure.

of the other methods. This should allow easy integration of a plugin framework at a later stage, where third party widget developers only need to implement the specific parts of the new widget they want to provide and the common tasks are still handled by the default implementation in `DashboardWidget`.

The subscription setup and the properties management are implemented in the `DashboardWidget` class, because they will be the the same for most widgets. The main reason is to hide the technical details from a third party plugin developer to make his life easier. The plugin developer only needs to specify which properties his widget has and must implement the callback that gets called when the properties have been changed. The `DashboardWidget` supports numeric, text and float properties. If more specific properties are needed the plugin can overwrite the properties-specific methods to provide an implementation tailored to this specific widget.

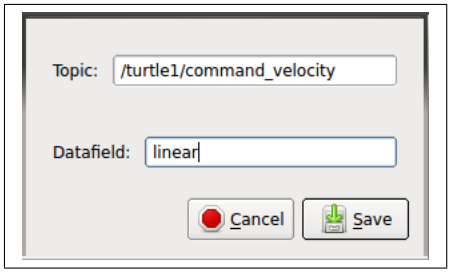

Fig. 3. Screenshot of the topic setup dialog.

2) Topic introspection: ROS topics were originally not designed and developed as something the user or developer chooses graphically: They are usually created, configured and used in the source code. ROSDashboard exposes the topic setup in a graphical user interface every time a new widget is added to the dashboard. To make this as easy as possible and without much overhead, a technical solution was chosen to reduce the number of fields to be set during the topic subscription setup. Normally you have to select a topic name and a message data type. The data type can be one of the standard message types like Float, Integer, String and Boolean or a more complex message type which

contains more information in a structured message. To access one data element of a message the "datafield" field was introduced in the graphical interface. Fig. 3 shows an exemplary topic setup configuration to access the linear velocity of the */turtlesim/Velocity* message published to the topic */turtle1/command_velocity*. Using Python's duck typing and the *rostopic* module it was possible to avoid the complexity of dynamically binding message type classes during runtime and detect the message type automatically. If a topic is not yet published and thus the message type of this topic is not defined yet, the method call to *rostopic* will block until the message type becomes available. To avoid blocking of the user interface a listener thread was implemented to wait until the message type for a topic becomes available (see Fig. 4). Avoiding to manually ask the user for a message type makes the configuration of widgets easier and faster for the user, it also keeps the implementation significantly simpler, because no dynamic binding of message type classes during runtime is needed.

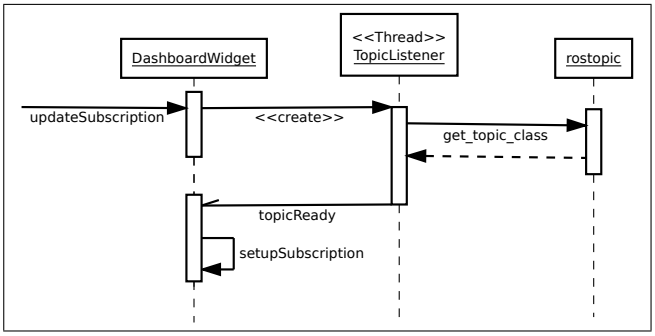

Fig. 4. Exemplary flow of events for asynchronous topic subscription setup.

V. Showcase

Although the current implementation is a prototype, it has all the features that were initially planned. The implementation is running stably and first attempts to use it as a debugging tool have been made. ROSDashboard is ready to be used for the development of robot applications and can be used for further evaluations (see Section VI).

Fig. 5 shows a simple showcase scenario: ROSDashboard is running alongside the *turtlesim_node* node which is used in many examples in the ROS tutorials[3]. It monitors the values for linear and angular speed which are published by the *turtle_teleop_key* node to control the turtle simulation. The String widget is configured to display messages from */rosout*, which in this example shows a warning when the turtle hits a wall. For the purpose of this small showcase, there was no need to modify the turtlesim source code. The only topics used by the showcase scenario are topics that are already used to control the turtle in the simulation and to display warnings.

Fig. 6 shows the ROS computation graph during the execution of the showcase scenario. It shows how ROSDashboard is connected to the nodes which are debugged. The topics

[3]http://www.ros.org/wiki/ROS/Tutorials

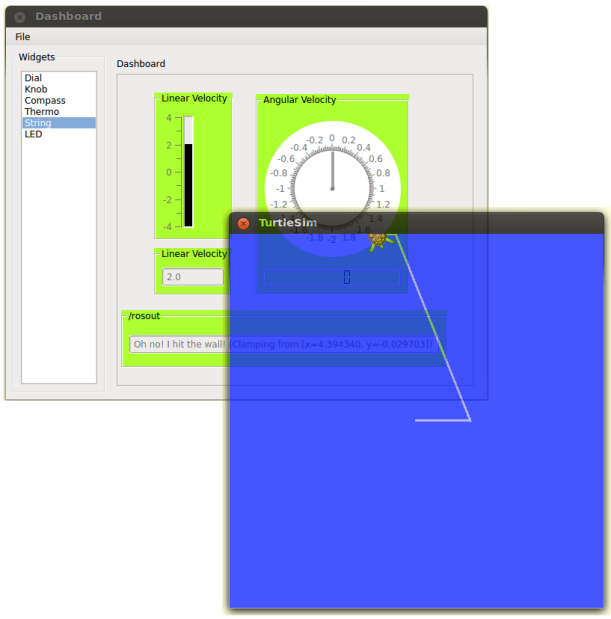

Fig. 5. ROSDashboard running alongside turtlesim_node.

needed for the showcase are */turtle1/command_velocity* for the linear and angular velocity and */rosout* for warnings when the turtle hit the wall.

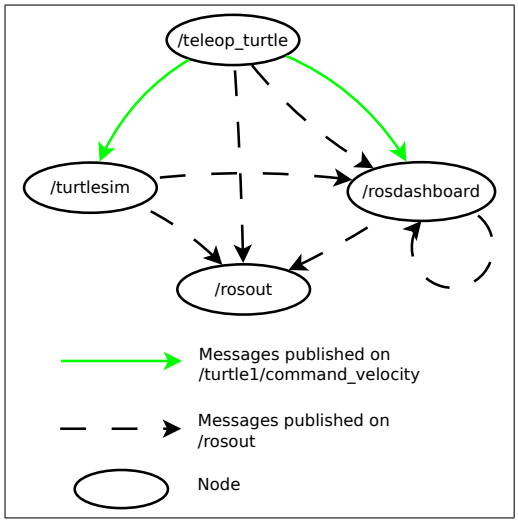

Fig. 6. Simplified ROS computation graph with ROSDashboard.

VI. Conclusion and Future Work

ROSDashboard, the tool presented in this paper, is capable of visualizing simple abstract data that is being published on a topic in the ROS communication middleware. Although the tool presented in this work has not been fully evaluated yet, it shows the possibilities and potential of simple visualizations during debugging of robotic applications. Further evaluations need to be conducted in order to quantify the improvements the tool has on a reduced cognitive effort and thus debugging performance.

The modular design and Open Source approach followed during the development of the tool makes it easy to extend and adapt the tool in the future. It also allows developers to choose a representation of data they find most helpful, with the only restriction being the number of available widgets and the data type compatibility. The simple drag and drop principle to add and remove widgets on the dashboard makes the tool flexible towards changes during the debugging process.

The presented tool is a first prototype to evaluate the use of a simple visualization tool during debugging. Further work can be done to increase the number of available widgets. Ultimately there should be a plugin engine where third party developers can easily add widgets for new and more complex kinds of data. The extensibility of the tool was one of the initial requirements and the structure of the program allows easy integration of such a plugin engine.

While the initial prototype's implementation of the topic setup dialog (Fig. 3) features simple text fields where the developer can enter arbitrary Strings, a more sophisticated solution can be implemented to improve the user interface. Using the existing ROS tools a list of available topics can be accessed, which makes it possible to create smarter interfaces. For example type-ahead completion for the topic name and a drop down list to choose the datafield parameter of a message are possible options.

Since the current logging mechanism in ROS transmits data as text, existing logging statements cannot be used as data source for the visualization. This can be resolved either by extending the logging mechanism to keep the type information of data or by implementing a parsing mechanism that can extract data which is embedded in text-based messages through regular expressions. This has been mentioned by the ROS community after the first prototype was announced. Another suggestion from the ROS community was to change the widgets to be more general and thus make it possible to have both visualization widgets and control widgets. This would give developers the ability not only to monitor values during execution but also manipulate configuration values and give commands to the robot during a debugging session.

RQT[4] is a graphical interface that groups together many different graphical tools for ROS. It was under active development during the time of this project and it was chosen not to integrate ROSDashboard into RQT yet, but to keep it in mind for future work. The plugin interfaces to integrate a tool into RQT have been finalized recently and ROSDashboard can easily be integrated. This would allow developers to use ROSDashboard amongst other graphical tools in one unified graphical interface that saves the state of all tools when exited and work can be resumed where left of.

ROSDashboard provides a good base to conduct research on the impact of visualizations during debugging of robotic applications. Future research can evaluate how visualizations affect the developer during debugging, especially if it decreases the cognitive load and thus makes debugging with visualizations faster and more productive.

ACKNOWLEDGMENT

This work is part of the final project in the Software Engineering Elite Graduate Program at University of Augsburg, Technische Universität München and LMU Munich[5]. It was conducted at the Department of Electrical and Computer Engineering, University of Auckland, New Zealand.

REFERENCES

[1] L. Gumbley and B. MacDonald, "Realtime Debugging for Robotics Software," in *Australasian Conference on Robotics and Automation (ACRA)*, Sydney, Australia, 2009. [Online]. Available: http://www.araa.asn.au/acra/acra2009/papers/pap167s1.pdf

[2] M. Grottke and K. Trivedi, "A classification of software faults," in *Supplemental Proc. Sixteenth International Symposium on Software Reliability Engineering*, 2005, pp. 4.19–4.20. [Online]. Available: http://www.grottke.de/documents/AClassificationOfSWFaults.pdf

[3] T. Collett and B. MacDonald, "An Augmented Reality Debugging System for Mobile Robot Software Engineers," *Journal of Software Engineering for Robotics*, vol. 1, no. 1, pp. 18–32, 2010. [Online]. Available: http://joser.unibg.it/index.php?journal=joser&page=article&op=view&path%5B%5D=7

[4] M. Quigley, K. Conley, and B. Gerkey, "ROS: an open-source Robot Operating System," in *IEEE International Conference on Robotics and Automation (ICRA)*, vol. 32, Kobe, Japan, 2009, pp. 151–170. [Online]. Available: http://pub1.willowgarage.com/~konolige/cs225B/docs/quigley-icra2009-ros.pdf

[5] T. Foote. (2012, July) ROS community metrics. [Online]. Available: http://pr.willowgarage.com/downloads/metrics/metrics-report-2012-07.pdf

[6] B. Gerkey, R. Vaughan, and A. Howard, "The player/stage project: Tools for multi-robot and distributed sensor systems," in *Proceedings of the 11th International Conference on Advanced Robotics*, Coimbra, Portugal, 2003, pp. 317–323. [Online]. Available: http://robotics.usc.edu/~gerkey/research/final_papers/icar03-player.pdf

[7] A. Makarenko, A. Brooks, and T. Kaupp, "Orca: Components for robotics," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Beijing, China, 2006. [Online]. Available: http://gias720.dis.ulpgc.es/Gias/asignaturas/master-siani-isspe/Bibliografia/ORCA/makarenko06iros_orca.pdf

[8] M. Montemerlo, N. Roy, and S. Thrun, "Perspectives on standardization in mobile robot programming: The Carnegie Mellon navigation (CARMEN) toolkit," in *Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, 2003, pp. 2436–2441. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1249235

[9] K. WHITLEY and A. F. BLACKWELL, "Visual Programming in the Wild: A Survey of LabVIEW Programmers," *Journal of Visual Languages & Computing*, vol. 12, no. 4, pp. 435–472, Aug. 2001. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S1045926X00901988

[10] S. Rozsnyai, J. Schiefer, and A. Schatten, "Concepts and models for typing events for event-based systems," in *Proceedings of the 2007 inaugural international conference on Distributed event-based systems - DEBS '07*. New York, New York, USA: ACM Press, 2007, p. 62. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1266894.1266904http://dl.acm.org/citation.cfm?id=1266904

[4] http://www.ros.org/wiki/rqt

[5] http://www.studieren.se