Rate based Impact Analysis

Nishant Sharma
University of Nebraska-Lincoln, nishants1994@gmail.com

Follow this and additional works at: http://digitalcommons.unl.edu/computerscidiss

Part of the Robotics Commons

http://digitalcommons.unl.edu/computerscidiss/136

This Article is brought to you for free and open access by the Computer Science and Engineering, Department of at DigitalCommons@University of Nebraska - Lincoln. It has been accepted for inclusion in Computer Science and Engineering: Theses, Dissertations, and Student Research by an authorized administrator of DigitalCommons@University of Nebraska - Lincoln.
Impact Analysis (IA) identifies control and data dependencies to determine the system components that could be affected by a change. Changes to robotic systems as they are updated often alter the flow of control and sensor data. Changes to the rates at which data is published from sensors, controllers, and other parts of the system are particularly subtle and difficult to detect. These rate changes, even if minor (e.g. lowering the frame rate of a camera), can propagate throughout the system and have broad impacts. However, for robotic systems, these changes in flow rate cannot be precisely tracked by just looking at the control or data dependencies. Thus, existing impact analysis techniques report imprecise impact sets and do not help us understand how the rate changes can impact the system.

In this work, we developed rate based impact analysis techniques that take into account the rate relations between components along with their control and data dependencies. We developed both static and dynamic techniques that perform rate based impact analysis. The static technique uses code patterns to define rate dependencies among system components. It is then used to limit the propagation of a rate change only to the rate dependent components. Apart from detecting rate dependencies, the dynamic technique also extracts a mathematical model to define the rate relationships among system components. It is then used to report how a rate change will affect other components. We performed multiple studies on Robot Operating System (ROS) based systems where our static technique showed a reduction in impact set by 59%. However, our dynamic technique utilizes a richer representation of rate
dependencies and reduces the impact sets by 78%. We also illustrated through case studies, how the dynamic technique helps to characterize the impact of a rate change.
DEDICATION

To my parents Ashok Kumar Sharma and Pushpa Sharma.
ACKNOWLEDGMENTS

I would like to express my special appreciation and thanks to my advisors, Dr. Sebastian Elbaum and Dr. Carrick Detweiler, for their supervision and support towards the completion of this work. I would also like to thank Dr. Gregg Rothermel for agreeing to be a part of my committee. I would like to acknowledge the valued assistance from the members of NIMBUS Lab.

A special thanks to my family for their constant support and encouragement. Finally, I would like to thank my friends: Aditya Immaneni, Shivam Thukral, Najeeb Najeeb, Mayank Garg, Rachit Jain, and Shashank Sharma for helping me get through the last two years.

Thank You!
This work was partially supported by National Science Foundation under awards #1526652 and #1638099, and USDA-NIFA #2013-67021-20947. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of these agencies.
Contents

1 Introduction 1
  1.1 Motivation 3
    1.1.1 Using traditional impact analysis 3
    1.1.2 Using static rate analysis 4
    1.1.3 Using dynamic rate analysis 7
  1.2 Contributions 12
  1.3 Organization 14

2 Related Work and Background 16
  2.1 Impact Analysis 16
    2.1.1 Impact analysis for distributed systems 21
    2.1.2 Distributed robotic systems and their rate properties 23
  2.2 Robot Operating System (ROS) 24
3 Rate Based Static Impact Analysis (RSIA)

3.1 Definitions ................................................................. 29
3.2 Approach ................................................................. 31
  3.2.1 Dependency Analysis (DA) ........................................ 32
  3.2.2 Impact Analysis (IA) ............................................... 37
3.3 Implementation and Limitations ....................................... 39
3.4 Study ......................................................................... 40
  3.4.1 Artifacts ................................................................. 40
  3.4.2 Impact Analyses (IA) Techniques ................................. 41
  3.4.3 Study Setup ............................................................ 42
3.5 Results ........................................................................ 43
  3.5.1 RQ1: What are RSIA’s precision and recall at capturing topic and rate dependencies? 43
  3.5.2 RQ2: How effective is RSIA compared to existing IA techniques? 46
3.6 Overhead Analysis .......................................................... 49
3.7 Threats to Validity .......................................................... 50
3.8 Summary ...................................................................... 51

4 Dynamic Rate Impact Analysis (DRIA)

4.1 Approach ................................................................. 55
  4.1.1 Throttled Replay (TR) .............................................. 55
  4.1.2 Flow Rate Analysis (FR) ............................................ 58
  4.1.3 Impact Analysis (IA) ................................................. 60
  4.1.4 Flow Diff Analysis (FD) ............................................. 61
    4.1.4.1 Use cases for a Differential Rate Graph (DRG) .......... 62
4.2 Implementation and limitations ........................................ 64
4.3 Study ......................................................... 65
  4.3.1 Artifacts .................................................. 65
  4.3.2 IA Techniques .......................................... 66
  4.3.3 Study setup ............................................. 67
4.4 Results ...................................................... 68
  4.4.1 RQ1: What are DRIA’s precision and recall at capturing topic and rate dependencies? ............................................. 68
  4.4.2 RQ2: How effective is DRIA when compared to other IA techniques? ............................................. 71
  4.4.3 Case Study: Care-O-Bot ................................ 74
4.5 Overhead Analysis .......................................... 78
4.6 Threats to Validity .......................................... 79
4.7 Summary ..................................................... 80
5 Conclusions and Future Work .................................. 81

Bibliography ...................................................... 84
List of Figures

1.1 Care-O-Bot (COB), a ROS based mobile manipulation robot. ............................................. 2
1.2 Motivating example for rate based static impact analysis......................................................... 5
1.3 Impact set reduction across the nodes of COB for static IA technique.................................... 7
1.4 An aerial water sampling robot (H2OS) [30]................................................................. 8
1.5 Motivating example for rate based dynamic impact analysis.................................................. 9
1.6 Differential rate graph generated by rate based dynamic IA technique................................... 11
2.1 A simple ROS component snapshot............................................................... 24
3.1 High-Level Architecture of the proposed approach....................................................... 32
3.2 Summary for Source Code shown in Figure 3.2....................................................... 34
3.3 Impact set reduction ratio of A-GT and A-Tool over Trad.............................................. 47
3.4 Impact sets for COB components using RSIA..................................................... 48
3.5 Impact sets for PR2 components using RSIA..................................................... 48
3.6 Impact sets for H2OS components using RSIA..................................................... 49
3.7 Time overhead analysis for using RSIA............................................................... 50
4.1 Different Rate-Dependency scenario’s.......................................................... 54
4.2 Architecture of DRIA..................................................................................... 55
4.3 Impact set reduction comparison for various techniques.............................................. 71
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Summary of related work on impact analysis</td>
<td>17</td>
</tr>
<tr>
<td>2.1</td>
<td>Summary of related work on impact analysis</td>
<td>18</td>
</tr>
<tr>
<td>2.1</td>
<td>Summary of related work on impact analysis</td>
<td>19</td>
</tr>
<tr>
<td>3.1</td>
<td>List of symbols and their descriptions</td>
<td>29</td>
</tr>
<tr>
<td>3.2</td>
<td>Summary of source code statements/expressions considered from ROS API</td>
<td>33</td>
</tr>
<tr>
<td>3.3</td>
<td>RSIA: attributes for the analyzed systems</td>
<td>40</td>
</tr>
<tr>
<td>3.4</td>
<td>RSIA: techniques compared</td>
<td>41</td>
</tr>
<tr>
<td>3.5</td>
<td>RSIA-Tool Edge Detection for Subscribers</td>
<td>43</td>
</tr>
<tr>
<td>3.6</td>
<td>RSIA-Tool Edge Detection for Publishers</td>
<td>44</td>
</tr>
<tr>
<td>3.7</td>
<td>Edge classification comparison between RSIA-GT and RSIA-Tool</td>
<td>45</td>
</tr>
<tr>
<td>4.1</td>
<td>DRIA: attributes for the analyzed systems</td>
<td>65</td>
</tr>
<tr>
<td>4.2</td>
<td>DRIA: Technique comparison</td>
<td>66</td>
</tr>
<tr>
<td>4.3</td>
<td>DRIA: Subscriber detection</td>
<td>67</td>
</tr>
<tr>
<td>4.4</td>
<td>DRIA: Publisher detection</td>
<td>67</td>
</tr>
<tr>
<td>4.5</td>
<td>Edge classification comparison between DRIA-GT, DRIA-Tool, and RSIA-Tool</td>
<td>69</td>
</tr>
<tr>
<td>4.6</td>
<td>DRIA: case study details</td>
<td>75</td>
</tr>
</tbody>
</table>
List of Algorithms

1. Detecting and labeling rate dependency for each outgoing topic in a node n. 37
2. Algorithm for generating impact sets. 38
3. Throttled Replay Analysis. 57
4. throttle_record. Throttle the input at different rates while monitoring the rates of outgoing topics for a components. 58
5. Flow Rate Analysis Fits mathematical models on outgoing topics. 60
6. Updated rate dependency check from algorithm 2. 60
7. Updated rate dependency check for performing IA using a DRG. 62
8. Repeatability Test: To make sure that the component is properly setup before we try to throttle the incoming topics. 63
Chapter 1

Introduction

Impact Analysis (IA) techniques extract system component dependencies to generate a graph and traverse it to identify every reachable component affected by a change. Determining the set of affected components supports activities ranging from testing [31] to debugging [37]. Such component dependencies are usually extracted from source code [35], system traces [10], or a combination of the two [29]. IA techniques are usually assessed regarding their precision and recall [22]; that is, their effectiveness in recognizing all and just the entities affected by the change.

Impact analysis has also been explored for distributed systems using both static program analysis [21] [34] and dynamic system trace analysis [14] [19] [41]. These techniques differ in how they represent the system components when compared to non-distributed software. However, they still utilize the data and control dependencies of these systems to generate dependency graphs.

Many other systems, on the other hand, are also affected by rate dependencies in addition to data and control dependencies. For example, a robot’s performance and behavior depend in part on the rate at which data is produced and consumed by its components. Consider the Care-O-Bot (COB) robot [2] shown in Figure 1.1. In this robotic system, replacing an
Figure 1.1: Care-O-Bot (COB), a ROS based mobile manipulation robot.

arm position encoder with a higher resolution sensor, but with a lower data rate, may result in a position controller instability. Alternatively, updating a planning algorithm with one that renders faster data rates may overwrite a buffer potentially leading to skipping certain actions. Similarly, increasing the camera frame rate could result in better obstacle avoidance, but might worsening feedback to a remote operator if WiFi bandwidth is exceeded.

Clearly, in many systems such as robotic systems, the rate at which data is made available to some of these components is often as important as the data itself. In such cases, we are interested in understanding whether rate changes in the system could affect its performance and lead to incorrect behaviors. Therefore, we would like to know how rate changes propagate through a robotic system so that those areas that may be affected are examined and validated more carefully. Existing impact analysis techniques are imprecise in tracking such changes as they only utilize data and control dependencies.

In this thesis, we propose two IA techniques that target rate properties of distributed systems. We propose static and dynamic program analysis based techniques that utilize rate dependencies in distributed robotic systems to provide precise and smaller impact sets. A static rate based IA technique is essentially helpful for robotic systems as it analyzes
the source code of the system without executing it. Therefore, static IA is a safe option for robotic applications where executing a buggy system can be catastrophic (for example, robot crashing into the environment). However, as static techniques over-approximate impact, we also propose a dynamic rate based technique that captures rate dependencies at run time. Our dynamic technique also adds a how dimension to impact analysis results, reporting how a change will impact other components whereas existing impact analysis techniques only report what components are affected. Let’s have a look at some motivating examples emphasizing the potential benefits of adding rate dependencies to impact analysis techniques.

1.1 Motivation

In this section, we present examples from the Care-o-Bot robot (COB) [2] and an aerial water-sampling robot (H2OS) [30] after applying both traditional and the proposed impact analysis techniques. We use the term traditional impact analysis techniques for existing IA techniques that only utilize data and control dependencies to generate a system dependency graph.

We first explain how existing IA techniques analyze software systems to report impact sets. We present the benefits of a static rate impact analysis over traditional IA techniques. Then we present an example of the dynamic rate impact analysis and its benefits over traditional IA techniques. We also present an example of the above-mentioned how dimension to impact analysis results added by the proposed dynamic technique.

1.1.1 Using traditional impact analysis

Existing impact analysis employs static, dynamic, or a hybrid of the two program analysis techniques to extract software dependencies. The extracted dependencies usually are an abstraction of data or control flow between software entities. These dependencies altogether
represent a system dependency graph. IA techniques then traverse the system dependency graph starting from an updated component and report all reachable software entities as potentially affected by the change. This information is usually utilized for debugging (to trace which change impacted a failing test case), or to improve testing efficiency by carefully selecting what components to test and still guarantee system stability.

The software entities captured by different impact analysis techniques vary for different target system’s architectures. Many IA techniques utilize the extracted dependencies along with the domain knowledge from the targeted architecture to improve their precision and recall. In this thesis, we propose IA techniques that extract data and control dependencies for ROS based systems along with their rate dependencies, an aspect missing from existing IA techniques, to improve IA for robotic systems.

1.1.2 Using static rate analysis

The Care-O-Bot robot (Figure 1.1) is designed by Fraunhofer IPA to be a personal assistant in a variety of settings. COB sits on a movable base, utilizes lasers and cameras to scan the environment for obstacle detection and navigation, and employs two robotic arms for manipulation tasks. Like many other robotic systems, COB is implemented on top of the Robot Operating System (ROS) [36]. ROS based systems, like most distributed systems, utilize message passing for data sharing rather than the traditional object passing between functions. Such message passing systems usually employ publish-subscribe architectures. These systems rely heavily on publishers and subscribers to communicate using ‘topics’ (communication channels) between ‘nodes’ (software components) [27]. In architectures like ROS, message passing channels are often complex to identify and track. They have a large number of components and most dependencies are implied through further callbacks. For example, Care-O-Bot (shown in Figure 1.1) launches around 66 components to achieve its
perception and control capabilities. It has 451 communication channels often with more than one publisher or subscriber to each channel.

Figure 1.2 presents part of COB’s system dependency graph. The nodes are represented by circles and the topics are represented by squares. Out of COB’s 66 nodes and 451 topics, we only show 18 nodes and 9 topics. We assume that the node shown as a star has changed. Nodes and topics external to the abstracted system view are depicted in gray.

This graph encodes a conservative approximation of the node’s dependencies. In essence, if there is a path through the edges from one node to another, then the former can impact
the latter. In the case of COB, for example, if the change occurred in the process handling of the laser scanner (marked with a star in Figure 1.2), then a traditional impact analysis technique would traverse this dependency graph starting from the star shaped node and would propagate the effect along the edges to all reachable nodes. Using such an approach would render all the nodes in the graph in Figure 1.2 as potentially impacted by the change. The impact set contains 18 affected nodes that a developer will have to check.

Traditional IA techniques are likely to be overly conservative when changes are not data-driven but rather rate-driven, like in this case of updating the rate of the laser scanner. In such cases, we can do better by annotating the dependency graph with labels that reflect rate-dependencies among the edges. For example, let’s assume that we have a static analysis mechanism (like the one we propose in this thesis) to tell that the node marked as ‘A’ in Figure 1.2 publishes data to topic tf on a timer. Such a mechanism would recognize coding patterns that may be associated with rate changes. We can then label such outgoing edges as “Rate-Independent” (dashed in Figure 1.2). For other nodes like ‘B’, we can tell by examining its code that their publishing rate depends on the rate of the incoming topics because this node reacts to each inbound message by publishing a message of its own. With such information, we can prune the potential set of impacted nodes. The publishing edges from a node that are independent to the rate of incoming messages do not propagate the effect of changes in terms of rate, pruning the space of affected nodes. In fact, for the example in Figure 1.2, this process results in the reduction of the impact set of nodes by 61% (only 7 impacted nodes shown in the shaded area).

The benefits of the proposed static IA approach will vary based on the system coupling and the nodes being changed. Figure 1.3 illustrates the range of benefits for COB. The x-axis represents the 48 nodes in COB (leaf nodes from the system dependency graph are excluded since changing them does not impact any other node). The y-axis represents the impact set size assuming that the node on the x-axis changed. The pluses represent the
impact set size of traditional impact analysis; the circles represent the impact set size for the proposed rate-cognizant approach. The size of the impact set varies based on which node has changed, ranging from 19 to 1 with an average of 8.3 nodes impacted. The differences between the pluses and the circles show the potential of the proposed static approach, which is quite dramatic in some cases, especially for those nodes with longer dependency chains. Overall, the average reduction is 4.43 nodes, and for impact sets with an initial size of more than 10 nodes, the reduction is 9.82 nodes. This is highly beneficial for the developers both in terms of effort and time required to verify or test a system after an update.

1.1.3 Using dynamic rate analysis

We propose a dynamic program analysis based technique to complement the above mentioned rate based static impact analysis technique. It also helps in eliminating some of the limitations of static program analysis. We now present an example of a real robot illustrating the benefits of the dynamic rate based impact analysis technique.
Figure 1.4 shows an aerial water-sampling robot (H2OS) designed by the NIMBUS lab for assisting water scientists in efficiently taking water samples. It utilizes several onboard sensors for localizing outdoors and making sure it maintains a sufficient distance from the water bodies for safe flight. It uses a radio link to communicate with a ground station which executes its control and planning tasks.

Figure 1.5 presents a part of the system dependency graph for the aerial water sampling robot. The circles represent nodes, and the rectangles represent topics. An edge leaving a circle indicates a publishing connection from a node to a topic. Similarly, an edge leaving a rectangle indicates a subscription to that topic. The H2OS robot has 29 nodes and 50 topics. In Figure 1.5 we only show 20 nodes and 23 topics that are reachable from the changed component shown as a star (lower right). Hollow circles and rectangles represent links to the system components we ignored in this example.

The dynamic rate IA technique divides the rate dependencies into three categories. Figure 1.5 represents them as three different kinds of edges: solid, dotted, and dash-dotted. Solid
Figure 1.5: A part of the system dependency graph for the aerial water sampling (H2OS) robot [30]. Black circles represent the nodes. Blue rectangles represent topics connecting the nodes. Hollow circles and rectangles represent abstracted out subgraphs. Star represents the changed component. The graph contains 20 components which are reachable from the star. The lightly shaded region represents the impact set reported by our static IA technique whereas darkly shaded region is the impact set reported by our dynamic IA technique.

edges indicate that the rate of the outgoing topic is dependent on all incoming topics to that node. These edges propagate the impact of a system update on all the connected nodes. Traditional IA techniques that do not consider rate properties of distributed systems assume all edges to be solid. In the example shown in Figure 1.5, any traditional IA technique will propagate the impact of the change node (shown as a star) to every other component in
the graph. Therefore, a traditional technique in this example would report all 20 shown components as affected.

Another categorization of rate dependencies defines an outgoing topic to be rate-independent of all incoming topics on a node. These edges are shown as dotted lines in Figure 1.5. For example, node ‘A’ in Figure 1.5 which is a controller node in H2OS, is one such node which always maintains a fixed message output rate of 30hz irrespective of the rates of incoming topics to that node. As discussed in the previous section, our static analysis technique attempts to detect these rate-independent edges. Moreover, it then uses those edges for impact set reductions by not exploring subgraphs from the independent edges. In Figure 1.5 the static IA technique reduces the impact set from 20 to 18 nodes (light shaded region).

The third rate dependency categorization is defined when an outgoing topic is rate-dependent on one or more, but not all of the incoming topics. Our dynamic IA technique monitors the outgoing topic rates while manipulating the rates of incoming topics. Then the technique attempts to fit mathematical models on the rates of incoming and outgoing topics for each node. These models then define a set of incoming topics that affect the rate of the outgoing topic. Topics that have a rate dependency on one or more incoming topics are categorized separately by our dynamic IA technique. They are shown as a dash-dotted edge in Figure 1.5. For example, node ‘B’ (in Figure 1.5) is a multiplexer node that publishes messages from a selected incoming topic based on a system state. However, its rate is independent of the rate of the incoming topic of the multiplexing topic (the edge connected to the updated star component). Hence successors of the outgoing topic from node ‘B’ can be deemed unimpacted by the change. Our dynamic IA technique detects these edges together with the dotted edges to help further reduce the impact sets. In Figure 1.5 our approach reduces the impact set to just 8 nodes shown as a darkly shaded region (a 60% reduction from traditional IA techniques).

The dynamic IA technique also reports how a component might be affected by a change,
Figure 1.6: Abstracted flow rate graph for COB. The changed node is shown as a star. Circles represent nodes and rectangles represent topics. Grey colored components indicate links to the unaffected thus abstracted parts of the differential rate graph.

where traditional IA techniques only report what components are affected by a change. Our dynamic IA technique blends the information collected during system analysis with impact analysis results to generate a Differential Rate Graph (DRG) that captures the system’s current state and its state after an update. Knowing these states can help trigger further impact set reductions and the graph can also be utilized for system verification, optimizations, or update confirmations.

Figure 1.6 shows part of the DRG after an update (as mentioned in [I]) in the Care-O-Bot robot. A star represents the updated component. We annotate the edges of the DRG with a \(<t, \{RD_t\}, r_o \rightarrow r_n>\) label. Here, \(t\) represents the topic name, \(RD_t\) stands for its rate dependency set (incoming topics on which its rate is dependent), \(r_o \rightarrow r_n\) presents how the rate changed after a system update, \(r_o\) stands for the old rate value, and \(r_n\) stands for the new rate value. An asterisk in front of a topic label signifies that the topic’s rate has been affected by the update. In this example, we show only a part of the flow graph, abstracting components that are unaffected by the change. We also hide the labels of certain topics as they simply propagate the same rate change to the next component.

Figure 1.6 presents the DRG for the system update regarding a rate change in topic \(t_5\)
from 100Hz to 1Hz as mentioned in the COB’s issue report [1]. The DRG captures the new rate value of topic $t_5$ and updates the topic label to $<t_5, \phi, 100 \rightarrow 1>$. The label indicates that (a) the topic $t_5$ does not depend on any incoming topics, and (b) the topic rate has been changed from 100Hz to 1Hz. The updated label also provides a quick update confirmation to the developer suggesting whether the system update reflects the intended behavior or not.

The DRG can also be used to automatically propose the optimization mentioned in [1] which required source code inspection by a developer. The DRG can achieve this by first forming a data dependency chain followed by an analysis of the connected topics for rate patterns. For example, in Figure 1.6, the data dependency chain, focusing on topic $t_5$, is of the order $\{t_1, t_2\} \rightarrow \{t_5\} \rightarrow \{t_8\} \rightarrow \{t_9\}$. If we were to replace the arrows with rate inequalities the order would become $\{t_1, t_2\} \prec \{t_5\} \succ \{t_8\} = \{t_9\}$. It is clear that topic $t_5$ has a higher rate value than its neighbors and there might be a potential benefit in reducing the rate as it might free bandwidth or CPU processing time benefiting other system components. Automatically suggesting these optimizations can also help save crucial development time. We present more details on the DRG in a later chapter.

1.2 Contributions

In this thesis, we propose two different impact analysis techniques that add the rate dimension to impact analysis. This is particularly relevant to robotic systems that explicitly rely on timing properties or implicitly rely on rate assumptions.

We propose a static program analysis based impact analysis technique which builds on the insight that certain code patterns result in a fixed outgoing data rate making the topic independent of the node’s incoming data rates. We use that insight to reduce the number of components reported as affected by the change. This proposed static IA technique has been published in [10]. Our contributions are:
• A novel approach to impact analysis focused on the rate of incoming and outgoing data, an aspect overlooked by existing impact analysis techniques. The analysis incorporates components’ source code patterns that render data production rates independent of the incoming data rates (and hence independent of changes that may affect those incoming rates).

• A tool implementing the approach, targeting systems built in C++ using the Robot Operating System (ROS) middleware. The tool performs a static code and configuration analysis to identify what data flows between components and recognizes the patterns defined by the approach to infer rate independence.

• A study assessing the effectiveness and performance of the proposed approach on three systems (COB [2], PR2 [6], and H2OS [30]). The study shows that the approach has the potential to reduce the size of the impact set by up to half compared with existing IA approaches. The current automated implementation reduces the impact set size of three studied systems to 73%, 92%, and 41%. It does not require code execution and has an analysis overhead of about one third of compilation time.

We also propose a dynamic program analysis based IA technique which records data available at the incoming and outgoing topic as system traces. We replay system traces with throttled speeds to simulate various incoming data rates while simultaneously monitoring outgoing topics for rate changes. This helps us explore the mathematical relationship between the incoming and outgoing communication channels. Extracting the mathematical relation helps us (1) isolate outgoing channels from impact propagating from incoming channels (2) further optimize impact analysis, and (3) report how a rate change impacts other components in the system. The proposed dynamic IA technique attempts to reduce false positives while maintaining low false negatives when compared to the static rate based IA technique. As the dynamic analysis has access to run time information for a compo-
nent, it also improves on the detection of dynamically established communication links. Our contributions are:

- A novel approach to impact analysis that while reporting *what* components are affected by a change, also reports *how* those components are affected. The dynamic program analysis is focused on defining a rate based mathematical relation between the incoming and outgoing channels of a component.

- A tool implementation of the proposed approach, targeting systems built using Robot Operating System (ROS).

- A study assessing the effectiveness of the proposed approach on two systems (COB [2], and H2OS [30]). Our current implementation on average reduces the impact set size of the two systems by 80% providing a 40% optimization over existing rate based IA techniques.

- A case study on Care-O-Bot showing that rate flow graphs can be used for system update verification, verifying and proposing optimizations. We also present other ways to further utilize the rate flow graphs generated by the technique.

Finally, we present a roadmap on how a combination of the two complementary static and dynamic techniques can help improve the rate based impact analysis. We also present possible future directions for the work presented in this thesis.

### 1.3 Organization

The remainder of this thesis is organized as follows. We begin with a survey of related work. Since our work encompasses interdisciplinary work between robotics and software engineering, we present related work on impact analysis, distributed robotics systems, and
the intersection of the two domains focusing on the rate of information exchanged. We also provide a brief background on Robot Operating System (ROS) [36] a distributed software architecture that supports a publish-subscribe based communication model and is used by the systems we study.

After the related work, in Chapter 3 we present the new rate based static impact analysis technique. We cover the approach, its implementation, and the current limitations of the implementation. Furthermore, we present an analysis of the technique’s complexity and stopping conditions. We then explain the experimental setup which includes details on how the study was performed and what tools were used to analyze the outcomes. We present the obtained results in a categorized manner and also include the possible threats to validity of the study.

In Chapter 4 we introduce the rate based dynamic impact analysis technique that tries to eliminate the limitations of the static analysis technique. Following a pattern similar to that of the static impact analysis chapter, we present the approach along with its implementation details and limitations. Additionally, we present an analysis of the approach’s complexity and stopping conditions. We further present the artifacts of the systems analyzed as part of the study performed to evaluate the effectiveness of the dynamic analysis technique. This is followed by the obtained results and their analysis. Moreover, we present a case study that emphasizes on how robotic software developers can utilize the new information encoded in the rate flow graphs for various tasks like debugging or optimization.

Finally, in Chapter 5 we conclude this thesis with a summary of our contributions and possible future work.
Chapter 2

Related Work and Background

This chapter presents prior work in the field of impact analysis and its proposed applications on distributed systems. We also look at related work from distributed robotic systems and their rate properties. The related work highlights common difficulties in this area, what has been done, and what the limitations of prior work in the existing literature are. This motivates the contributions of this thesis and establishes the context in which they are developed. We follow related work with a brief background on Robot Operating System (ROS) which is the main target of the studies done in this work. Finally, we define and summarize the terminology used in the rest of the thesis.

2.1 Impact Analysis

Impact analysis (IA) was formally defined by Arnold and Bohner \[11\] as either identifying what to modify to accomplish a change \[37\] or identifying the potential consequences of a change \[25\] \[26\]. Impact analysis techniques use program analysis to report a set of entities that are either affecting or are affected by a change in the system. This set is called an impact set. Since \[11\], the impact analysis techniques have grown to be divided into three
major categories based on how the system is analyzed. These categories are static, dynamic, and hybrid.

Table 2.1: Summary of related work on impact analysis.

<table>
<thead>
<tr>
<th>Citation</th>
<th>Element of impact set</th>
<th>Analysis Type</th>
<th>Novelty</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arnold and Bohner [11]</td>
<td>Not applicable</td>
<td>Not applicable</td>
<td>Formally defines the term impact analysis and provided a framework for comparing different IA techniques.</td>
</tr>
<tr>
<td>Law and Rothermel [26]</td>
<td>Methods</td>
<td>Dynamic</td>
<td>Provides a dynamic impact analysis technique based on whole path profiling.</td>
</tr>
<tr>
<td>Orso et al. [31]</td>
<td>Block or methods (Developer’s choice)</td>
<td>Dynamic</td>
<td>Provided a technique for impact analysis and regression testing using field data with forward slicing. Also, presented case studies of real subjects on a real user population.</td>
</tr>
<tr>
<td>Eugster et al. [18]</td>
<td>N/A</td>
<td>N/A</td>
<td>Provides insights on full decoupling of the communicating entities in time, space, and synchronization for Publish-Subscribe architectures. Can be used to decouple components from impact propagating channels.</td>
</tr>
<tr>
<td>Ren et al. [37]</td>
<td>Tests</td>
<td>Static</td>
<td>Identifies affected test cases from an update by decomposing difference into a set atomic changes.</td>
</tr>
<tr>
<td>Apiwattanapong et al. [10]</td>
<td>Methods</td>
<td>Dynamic</td>
<td>Provides a dynamic IA technique that uses execute-after relations between entities.</td>
</tr>
<tr>
<td>Feng and Maletic [19]</td>
<td>Components, interfaces and methods</td>
<td>Dynamic</td>
<td>Identifies impact entities using impact rules on component interactions derived from component and sequence diagram.</td>
</tr>
</tbody>
</table>
Table 2.1: Summary of related work on impact analysis.

<table>
<thead>
<tr>
<th>Citation</th>
<th>Element of impact set</th>
<th>Analysis Type</th>
<th>Novelty</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jashki et al. [23]</td>
<td>Files</td>
<td>Static</td>
<td>Identifies a set of closely associated files also including non source code files which are usually ignored by program analysis based IA techniques.</td>
</tr>
<tr>
<td>Hattori et al. [22]</td>
<td>Classes (expandable to methods or fields)</td>
<td>Static</td>
<td>Performs a depth bounded IA search on a system representation of entities and relations.</td>
</tr>
<tr>
<td>Maia et al. [29]</td>
<td>Methods</td>
<td>Hybrid</td>
<td>Identifies impact sets combining both dynamically detected dependencies and statically obtained structural dependencies. Ranks the impact set results based on an entities relevance to a change.</td>
</tr>
<tr>
<td>Popescu et al. [34]</td>
<td>Components</td>
<td>Static</td>
<td>Detects inter- and intra-component dependencies to aid impact analysis in distributed systems.</td>
</tr>
<tr>
<td>Purandare et al. [35]</td>
<td>Statements</td>
<td>Static</td>
<td>Identifies conditional statements that affect the conditions under which a component produces a message in distributed systems.</td>
</tr>
<tr>
<td>Garcia et al. [21]</td>
<td>Statements, Message Types</td>
<td>Static</td>
<td>Improves on Popescu et al. with respect to detection of component dependencies in event-based distributed systems using message field detection.</td>
</tr>
<tr>
<td>Chen et al. [15]</td>
<td>Methods</td>
<td>Hybrid</td>
<td>Improves the precision of impact sets obtained by a static analyzer by generating another set of impact set using a aspect-based dynamic impact analysis technique.</td>
</tr>
</tbody>
</table>
Table 2.1: Summary of related work on impact analysis.

<table>
<thead>
<tr>
<th>Citation</th>
<th>Element of impact set</th>
<th>Analysis Type</th>
<th>Novelty</th>
</tr>
</thead>
<tbody>
<tr>
<td>Safi et al. [39]</td>
<td>Events</td>
<td>Static</td>
<td>Identifies event anomalies in event-based distributed systems by defining causation between consuming event types and accessed fields.</td>
</tr>
<tr>
<td>RSIA [40]</td>
<td>Components</td>
<td>Static</td>
<td>Introduces rate properties of distributed systems into impact analysis using static program analysis.</td>
</tr>
<tr>
<td>DRIA</td>
<td>Components</td>
<td>Dynamic</td>
<td>Proposes a dynamic program analysis based approach to rate based impact analysis. The approach also reports how a component is affected.</td>
</tr>
</tbody>
</table>

Static impact analysis techniques analyze the code to generate data or control flow representations. They mimic some of the parsing and analysis performed by a compiler, and traverse that representation based on the changes made to a code base. Because static techniques ignore system input, they tend to overestimate impact sets by considering every potential input. They can vary in the type and granularity of dependency captured. Chianti [37] captures atomic changes in source code and uses system call graphs to report impact sets. Chianti also labels the test cases affected by changes and helps reduce the testing effort required to revalidate a system. The Impala [22] tool works on call graphs of a system and represents the analyzed system as entity and relations. Impala introduces a dependency
graph depth bounded impact search. For a given change, Impala recursively searches for indirect dependencies until there are none left or the desired depth is reached. Impala shows high precision and recall even for small search depth levels. Jashki et al. [23] propose an impact analysis approach which mines clusters of closely associated files to generate impact sets by looking at a software change repository. The impact set generated by [23] also includes non-source code files, for example: configuration files, which are typically ignored by program analysis based impact analysis techniques. Acharya and Robinson [9] combine program slicing on a commercial static analyzer, CodeSurfer, to optimize performance and accuracy of impact analysis on large software systems. The resulting tool optimizes based on different slicing criterias, also making impact analysis customizable.

Dynamic impact analysis [10] [13] [25] [31] [32] on the other hand uses information collected through traces during the runtime execution of a system. Unlike static IA, dynamic IA underestimates impact as it only accesses information from the configuration on which dynamic IA was launched. Dynamic IA can be tuned to be computationally less expensive when compared to static IA which tries to consider all possible scenarios, whereas dynamic IA only deals with information collected at run time. PathImpact [25] [26] represents execution traces as a whole-path DAG to find impact sets of the given changes. PathImpact is incremental, and thus does not require recomputing the information after each update. CoverageImpact [31] gathers run time information to generate a covered set (all methods on a path where a changed function is seen) and a slice set (methods appearing in forward slices of the changed method) and then computes an intersection of the two to generate an impact set. An empirical study [32] on both CoverageImpact and PathImpact shows that the CoverageImpact technique is computationally inexpensive regarding time and space whereas the PathImpact technique is more precise at reporting impact sets. CollectEA [10] is a dynamic IA technique which extracts (a) when a method is called and (b) when a method returns, and defines a binary relation called ExecuteAfter. CollectEA then traverses the
ExecuteAfter relations to generate impacted sets. ExecuteAfter relations simply capture the entities that execute after an entity change in a system update. CollectEA is both time and space efficient as it only collects and analyzes a small amount of dynamic information. Diver [13] uses sequences of method level event execution traces with statically determined control and data dependency knowledge to track impacts. Although Diver utilizes more space and time than PathImpact, it reports more precise impact sets while reducing false positives.

Hybrid impact analysis techniques [29] [15] attempt to merge both static and dynamic techniques to reduce false positive and false negatives. Maia et al. [29] present SD-Impala, a hybrid of static and dynamic IA analysis techniques, and compared it against Impala (static IA) and CollectEA (dynamic IA). SD-Impala shows better recall than the other two algorithms but has lower precision than the static analysis technique. Given the overestimating nature of static impact analysis, Chen et al. [15] present a hybrid approach adding dynamic analysis that feeds on the output of their static analysis to improve the precision of the technique. They only remove a false positive from the impact set when it can be proven so using dynamic IA technique.

Since we focus on distributed systems in this work, we now consider prior impact analysis work that targets distributed system architectures.

2.1.1 Impact analysis for distributed systems

IA techniques rely on some form of system dependency graph to generate impact sets. Distributed systems decouple data-based dependencies between components, and dependencies are represented through communication channels. Extracting data and control dependencies in such systems is a non-trivial task. Rather than relying on traditional object passing between functions or invoking member functions of a class, distributed systems rely on message passing, implicit callbacks, and non-deterministic concurrency that reduces both the effi-
ciency and effectiveness of existing techniques and therefore requires specialized techniques for detecting dependencies. Both static and dynamic program analysis based techniques have been proposed to extract such dependencies. Purandare et al. [35] present a static program analysis approach which extracts a component dependency model from source code and also detects the conditions under which those dependencies occur. The extracted information is then used to alert developers about modifications to the message producing components that introduce or remove a conditional dependency. Popescu et al. [34] present Helios, which uses call graphs and state annotations to detect intra-component dependencies and source-sink detection for inter-component system dependencies in the source code. Garcia et al. [21] propose Eos, a static analysis based approach to extract component dependencies by detecting attribute based message field usage and message flow between system components. They define a message flow graph containing consumed message types and published message types by adding directed edges between the two, representing both intra- and inter-component flow dependencies. Safi et al. [39] present DEvA, a static analysis based technique that reports anomalies in event based distributed systems. To report event anomalies, DEvA extracts consumed event types and component field access locations. DEvA defines causation between the events and field access locations to generate two distinct sets, ConsumedToDef and ConsumedToUse, that represent the type of relation between a event and the corresponding accessed field. Finally, both these sets are joined to report impact sets. DISTIA [14] is a dynamic impact analysis technique that instruments a program to monitor the method execution and message passing events against a synchronized logical clock to compute partially-ordered sequences of method executions along with a message-receiving map for each process. Finally, the information is combined to generate event traces for all processes which defines dependencies to facilitate impact analysis.
2.1.2 Distributed robotic systems and their rate properties

Distributed robotics systems differ from other systems in their high dependency on (1) the environment and its configurations, and (2) the frequency or rate contracts of different components that guarantee system safety. Many studies have targeted the rate properties of such distributed systems [16] [18] [28] [33] [38]. Paikan et al. [33] propose a technique to favor time-critical components that require higher or fixed controlling rates by improving resource utilization using run-time prioritization certain of communication channels. Rusakov et al. [38] present six concurrency design patterns for coordination among components to solve common robotics task. These concurrency patterns are specifically interesting for robots as most tasks operate in parallel. Eugster et al. [18] discuss how the publishers or subscribers in a distributed architecture can be decoupled in terms of space, time, and synchronization. Identifying these commonalities allows one to better understand and implement scalability for publish-subscribe architectures. In this work, we also attempt to decouple publishers and subscribers in terms of their rate and time. However, our work is different in the way we utilize the decoupling for the purpose of impact analysis. We use decoupling as a tool to separate impact propagating entities from impact limiting entities. This separation allows us to label certain components safe from a change impact even in the presence of direct or indirect dependencies as long as it can be decoupled regarding its rate or time properties.

To the best of our knowledge, there is no work on impact analysis for distributed system that takes their rate properties into account. Therefore, in this thesis, we focus on rate properties of distributed systems for generating a system dependency graph. We now present a brief overview of ROS (Robot Operating System), a distributed software architecture.
2.2 Robot Operating System (ROS)

Robot Operating System (ROS) is an open source meta-operating system. ROS provides libraries and tools to help software developers create robotic applications. ROS is highly popular among both academic and industrial institutions for creating robotic systems. As of July 2016 [20], ROS had been cited 2683 times and there are more than 100 open-source robots available for both industrial and research use. ROS provides a topic-based publishing subscription abstraction. The ROS API provides a standard way to interconnect producers of information (like sensors) and consumers of information (like actuators). Processes in ROS are called ‘nodes’. A node can both consume and produce information. The produced information from one node can be shared with other nodes using ‘topics’ (communication channels). Topics are registered by nodes by clearly stating the topic name and whether the node is subscribing or publishing to that topic. A node can register any number of topics. However, for two nodes to communicate through a topic, an agreement on the data type is required. Topics in ROS can use only a single predefined data type for communication. These data structures are strongly typed and are called ‘ROS messages’. All the topic based communication in ROS is peer to peer and asynchronous. However, a central process called ‘ROS Core’ manages the registrations of different nodes and topics.

Figure 2.1 shows the 2 common ways of communication in ROS. Other than topics,
ROS also supports RPC calls called ‘ROS Services’. Service calls are synchronous and the initiating node gets blocked until the remote node responds. Topics, on the other hand, are asynchronous. A node can continue with its execution after publishing a message on a topic without waiting for any confirmation. Similarly, the subscribing node receives the messages directly in a buffer, and it is up to the node to check the buffer at its discretion. Though the topic based communication is asynchronous and requires no contract between two components, many components still expect the incoming messages to arrive at certain rates to perform optimally. The ROS API provides different ways to guarantee a timely execution of a function or a loop; a publish call made from such a code location is guaranteed to execute at fixed time intervals leading to fixed outgoing message rates. For example, ROS provides a Timer API to invoke a function at fixed time intervals. ROS also provides an adaptive sleep function through its Rate API which only lets the loop execute at fixed rates by sleeping after each loop iteration is executed until the next time interval is reached. Recently, ROS has introduced ROS Hz tests to check a topic’s rate value. These tests are defined using a topic’s name, a rate value, and a time interval. To check a ROS Hz test, system is executed and the given topic’s rate value is monitored for the given time interval and then compared against the given test value.

ROS also makes it easier to launch a system through the usage of XML based launch files. These files describe which components to launch and can also contain parameters that the launched nodes may read at runtime. The parameters from launch files are stored by the ROS core and a node can poll the core to set or get a value. Launch files are also used to define remappings for topic names to promote reusability of code. Some developers also used launch files to define topic names that are read polling the ROS Core at node startup. To further improve reusability, ROS provides tools that support runtime linking of shared libraries to any node. ROS also provides API to capture and playback the message stream for any topic as seen at runtime. These streams are called ‘Bag files’. Bag files are used in
ROS for both testing and debugging proposes.

ROS has no regulations on how fast a node should publish on a topic. However, most robots have components that make assumptions about data production and consumption to guarantee a smooth execution. When a node is updated, the rate of information production or consumption may change that might cause an assumption to become false for its connected components. We address this issue by proposing impact analysis techniques that take rate properties of robotic systems into consideration. We now present our rate based static impact analysis technique.
Chapter 3

Rate Based Static Impact Analysis (RSIA)

The primary objective of the proposed rate based static impact analysis technique is to incorporate the information on message exchange rate of distributed robotic systems into impact analysis. The aim is to improve the precision and recall of impact analysis techniques when applied to robotic systems where many changes are rate driven rather than data or control driven.

In this chapter, we present our Rate based Static Impact Analysis (RSIA) technique. The approach uses static program analysis to identify whether a node’s publishing edges are rate-independent, meaning the node publishes messages on a topic at a fixed rate under all conditions. In this technique, we identify a common set of code patterns that are highly probable to render a publisher rate-independent. For a node like ‘A’ (Figure 1.2), a sample code pattern is shown in Figure 3.1, where the publishing rate to ‘tf’ is fixed as the semantics of the \(<\text{rate\_var}>\).\text{sleep}()\) call enforces a timed wait, causing the publishing call to be invoked at fixed intervals. For a dependent node, like ‘B’ (Figure 1.2), a sample code pattern is shown in Figure 3.2. In this example, the call to \(<\text{publisher}>\).\text{publish}(<\text{msg}>)\)
publishTransform(){
  <publisher>.publish(<msg>);
}

foo(){
  ros::Rate <rate_var>(getPubRate());
  while (ros::ok()){
    publishTransform();
    <rate_var>.sleep();
  }
}

Figure 3.1: Component A - Rate-Independent publisher due to fixed rate caused by the adaptive sleep function provided by ROS::Rate API.

callback(...) {
  <publisher>.publish(<msg>);
}

foo(){
  <publisher> = <nh>.advertise("scan_out",...);
  <sub> = <nh>.subscribe("scan_in", 1, &callback...);
}

Figure 3.2: Component B - Rate-Dependent publisher as an outgoing message is published against every incoming message.

(line 2) occurs within a subscription callback function so the publishing rate of this node to topic ‘scan_out’ will depend on the rate of received messages from topic ‘scan_in’. The callback function is invoked every time a message is received on the incoming topic ‘scan_in’. Therefore, a change in the incoming message rate affects the rate of the outgoing message deeming the outgoing topic ‘scan_out’ rate-dependent on the topic ‘scan_in’.

The identification and development of these kinds of code patterns followed an iterative process. We started with a pool of candidate patterns based on our development experience and recommended practices, followed by several refinement steps (for example: group code reviews, consistency checks with runtime information, and comparison with information...
available through ROS tools) as we searched for those patterns in other code bases. We have also built a tool that automatically recognizes these patterns and labels the publish-subscribe graph accordingly.

### 3.1 Definitions

Our proposed technique calculates an impact set for a given set of changed components in a distributed robotic system. We now define key terminology that will be used throughout the rest of the thesis.

Table 3.1 presents a summary of the symbols with their descriptions. We assume a distributed system $S$, that utilizes message passing communication between a set of nodes
$N$, and a set of communication channels (called “topics”) $T$. A publishing edge in $S$, $n \rightarrow t$ is defined as a node $n$ sending messages on the topic $t$ where $n \in N$ and $t \in T$. Similarly, a subscribing edge in $S$, $t \rightarrow n$, is defined as a node $n$ receiving messages from the topic $t$. Function $r(t)$ denotes the rate of message published on a topic $t$. For every node $n$, we also have a set of incoming ($n.T_I$) and outgoing ($n.T_O$) topics. $B$ presents a ROS based trace file, called a “Bag file”. The technique finally reports an impact set ($I$) for a given set of changed components. Some of the terminology shown in Table 3.1 will be formally defined in the next chapter.

To perform impact analysis, the technique we propose first generates a Node Dependency Graph ($NDG$) that captures data dependencies between system components. A node (data) dependency ($ND$) defines whether a node $n_i$ has a data dependency on another node $n_j$.

**Node Dependency, ND**: A node $n_i$ is $ND$ on $n_j$ if $n_j \rightarrow t \rightarrow n_i$, $\{n_i, n_j\} \in N$

A node dependency graph also defines the flow of information which is used to propagate the impact of a change. However, the impact of a rate change can be blocked by a topic that always publishes at a fixed rate or does not depend on the incoming topic that is affected by the change. Such impact propagation decisions are made based on the rate dependencies of topics. A rate dependency ($RD$) defines whether an outgoing topic’s rate for a given node $n$ is dependent on the rates of the incoming topics.

**Rate Dependency, RD**: Given $t_i \rightarrow n \rightarrow t_j$, topic $t_o$ is $RD$ on $t_i$ if $r(t_o) = f(r(t_i))$.

In the foregoing, $f()$ is any arbitrary function other than a constant. When $f()$ becomes a constant, the given topic becomes rate-independent as its rate value does not depend on the rates of other incoming channels. An $RD$ captures whether the rate of data coming to a node affects the rate of an outgoing topic. It is possible that the rate of a topic depends on more than one incoming topic, in which case the outgoing topic will be rate-dependent on more than one incoming topic.
The techniques we propose in this and the next chapter annotate the node dependency graph with rate dependencies to then generate a Node Rate Dependency Graph (NRDG). In an NRDG, data dependencies are used to propagate the impact of a change while rate dependencies are utilized to limit the impact of that change.

**Node-Rate Dependency Graph, NRDG**: A set of vertex $V = N \cup T$, and edges $E = \{ND\}$ is NRDG if $\forall e \in E, RD(e)$ is defined.

Once the NRDG is generated, the developer is asked to input a set of change components, $C$. An impact set $I_i$ for every changed node $n_i$ is calculated. Finally, a global impact set $I$ is calculated by taking a union over all the individual impact sets.

**Impact set, $I_i$**: $I_i$ is a collection of nodes that are affected by a given changed node $n_i$, where $n_i \in C$.

**Global Impact Set, $I$**: $I = \bigcup_{\forall n_i \in C} I_i$.

We provide further details about the approach and the implemented tool in the next section.

### 3.2 Approach

Figure 3.1 shows the high-level architecture of the proposed static IA approach. It is divided into two phases: Dependency Analysis (DA) and Impact Analysis (IA). DA takes the ROS system code and its launch file as input. DA outputs a Node-Rate Dependency Graph (NRDG) where edges from a publisher to a topic are labeled as either ‘rate-dependent on’ or ‘rate-independent of’ the rate of incoming messages. IA takes this rate dependency graph and the list of changed component(s) as input. It then performs a depth-first traversal of the graph, starting from those changed components and stopping when a leaf node or a rate-independent publishing edge is found. IA reports the reachable set of nodes that constitute
the Impact Set for the changed component(s).

### 3.2.1 Dependency Analysis (DA)

The first step of DA is to generate the node dependency graph where the vertices are the nodes or topics, and the directed edges link the publishers and subscribers with their topics. DA analyzes every function in the system to produce a function summary containing a list of every publisher or subscriber used by the analyzed function. The summary is produced by generating and traversing the function control flow graph while searching for source-code expressions from a predefined set that depends on the API and middleware being used.

Table 3.2 presents a summary of the source code expressions used from the ROS API to detect and define publishers and subscribers and publish calls to the registered publishers, and the source code patterns that are used to check whether a publish call is executed at a fixed rate. To recognize and register subscribers, we check for statements that contain a function call to the expressions mentioned in Table 3.2. Once found, we check if the expression is a function call to the function named `subscribe` on an object from the ROS API Class `ros::NodeHandle`. If both object and function call match, we extract and record the first and third arguments of the function call as they correspond to the topic.
Table 3.2: Summary of source code statements/expressions considered from ROS API that helps detect (a) Node dependency (registering a publisher or subscriber), (b) a publish call (indicates the termination of a source-code path for RSIA), or (c) a rate-pattern.

<table>
<thead>
<tr>
<th>Expression</th>
<th>ROS API Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>advertise</td>
<td>Nodehandle</td>
<td>defines a publisher to a topic</td>
</tr>
<tr>
<td>advertiseCamera</td>
<td>image_transport</td>
<td>defines publishers on multiple topics all pertaining to a connected camera</td>
</tr>
<tr>
<td>subscribe</td>
<td>Nodehandle</td>
<td>defines a subscriber to a topic</td>
</tr>
<tr>
<td>subscribeCamera</td>
<td>image_transport</td>
<td>defines subscribers on multiple topics all pertaining to a connected camera</td>
</tr>
<tr>
<td>sendTransform</td>
<td>TransformBroadcaster</td>
<td>indicates a publishing edge to topic ‘tf’</td>
</tr>
<tr>
<td>lookupTransform</td>
<td>TransformListener</td>
<td>indicates a subscription to topic ‘tf’</td>
</tr>
<tr>
<td>waitForTransform</td>
<td>TransformListener</td>
<td>indicates a subscription to topic ‘tf’</td>
</tr>
<tr>
<td>RealtimePublisher</td>
<td>realtime_tools</td>
<td>defines a publishing edge to a topic</td>
</tr>
<tr>
<td>TimeSynchronizer</td>
<td>message_filters</td>
<td>defines subscriptions on one or more topics</td>
</tr>
<tr>
<td>Synchronizer</td>
<td>message_filters</td>
<td>defines subscriptions on one or more topics</td>
</tr>
<tr>
<td>TimeSequencer</td>
<td>message_filters</td>
<td>defines subscriptions on one or more topics</td>
</tr>
<tr>
<td>Chain</td>
<td>message_filters</td>
<td>defines subscriptions on one or more topics</td>
</tr>
<tr>
<td>publish</td>
<td>Publisher</td>
<td>a message published on a single registered topic</td>
</tr>
<tr>
<td>publish</td>
<td>CameraPublisher</td>
<td>different messages published on multiple registered topics</td>
</tr>
<tr>
<td>unlockAnd Publish</td>
<td>RealtimePublisher</td>
<td>a message published on a single topic</td>
</tr>
<tr>
<td>registerCallback</td>
<td>message_filter</td>
<td>defines a subscriber to a topic</td>
</tr>
<tr>
<td>sleep</td>
<td>Rate</td>
<td>indicates a fixed rate execution of the parent loop</td>
</tr>
<tr>
<td>createTimer</td>
<td>Timer</td>
<td>indicates fixed rate execution of the callback function</td>
</tr>
<tr>
<td>createWallTimer</td>
<td>WallTimer</td>
<td>indicates fixed rate execution of the callback function</td>
</tr>
</tbody>
</table>

name and callback function for the incoming messages. To detect publishers, we look for the function call named ‘‘advertise’’ from an object of type ‘‘ros::NodeHandle’’. From the ‘‘advertise’’ function call, we extract and record the name of the declared object
and the first function call argument. First function call arguments report the topic against which the publisher is registered. The object name is then used to locate publish calls made to the publishing topic. Similarly, other types of subscribers and publishers are detected by first matching the object type and the function name from the expressions mentioned in Table 3.2 and then extracting the arguments from those function calls that correspond to the topic’s name and the callback function or the object name depending on the object type.

Extracted information on publishers and subscribers is used to define a function summary. For subscribers, it contains the topic name and the callback function name. For publishers, it has the topic name and the variable name. If combined, the summary represents all the incoming and outgoing topics from a node. Figure 3.2 presents a sample function summary for the function ‘foo’ from the code of node ‘B’ shown in Figure 3.2. Node ‘B’ subscribes to topic ‘scan_in’ and registers the function named ‘callback’ as the function to be invoked when a message is received on the topic. The publishing topic for node ‘B’ is ‘scan_out’ and the registration is done using an object named ‘publisher’. The other function named ‘callback’ does not register any publisher or subscriber. Therefore, there is an empty list for both publisher and subscriber in Figure 3.2. However, both registered publisher and subscriber are captured in the function summary for function ‘foo’.

To generate the node dependency graph (NDG), that represents the connections between nodes and their corresponding topics, the approach first performs a union of all the summaries

<table>
<thead>
<tr>
<th>Summary: callback</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subscriber:</td>
</tr>
<tr>
<td>Publisher:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Summary: foo</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subscriber: scan_in: callback</td>
</tr>
<tr>
<td>Publisher: scan_out: &lt;publisher&gt;</td>
</tr>
</tbody>
</table>

Figure 3.2: Summary for Source Code shown in Figure 3.2
of the functions in a node. Then, the approach adds a vertex for each node, an edge from the node to a topic for each publisher, and an edge from a topic to the node for each subscriber. At this point, we have a graph to perform impact analysis on. We now discuss the additional analysis performed to label certain edges as rate-independent, which will help to reduce the size of the impact set.

Once the approach has generated the NDG, it further examines the source code, analyzing every path leading to a publish call to identify certain code patterns that render those edges rate-independent. A publish call is detected by locating the above mentioned publish call related expressions from Table 3.2. The approach looks at all the paths leading to a publish call (we name these publish paths) to check if there exists a predefined rate-pattern along these publish path ensuring fixed rate message publishing. We now introduce three initial patterns that render a publish path rate-independent (others are mentioned in Chapter 5) with their particular instantiation in ROS.

1. **No Subscribers**: If a node does not subscribe to any topics, then the outgoing edges of that node are labeled “Independent”. This pattern is common for sensing nodes that capture environmental data and publish it. In Figure 1.2 the node shown as a star belongs to this class.

2. **Timer**: Robotic middleware often provides support for a function to be invoked at fixed intervals. In ROS, such a function can be registered as a callback function against ros::Timer or ros::WallTimer. The registered callback function is invoked every time the given duration equivalent to the ros::Timer has passed, executing the callback function at fixed intervals. Figure 3.3 shows an example of such a pattern. Since the callback function will be invoked at fixed time intervals, the publisher’s publish call will also be invoked at fixed intervals making the path from the timer callback to the publisher’s publish call an independent publisher path. To detect this pattern,
3. **Adaptive Sleep**: Robotic middleware like ROS often provides adaptive sleep functions which take the execution time of a cycle into account and sleep for the unused time of the initialized duration, ensuring that the loop is executed at a fixed rate. In ROS, this can be done by initializing a `ros::Rate` object which specifies the rate at which the loop should be executed. Then, inside a loop, the `ros::Rate` object’s sleep function is called to sleep until the next execution should start. This sleep function is different from traditional sleep functions as it regulates the sleep time as required.

For example, in Figure 3.1, the function `publishTransform` is called at a fixed rate as the loop will be executed at a fixed rate because of the adaptive `ros::Rate` based sleep call. To detect this pattern, we locate a `ros::Rate` object followed by a loop and a `ros::Rate` based sleep call. Then we label any function call or publish call independent within the loop body.

Algorithm 1 describes the edge labeling process for a given node $n$; $n.T_O$ represents the set of outgoing topics for the given node $n$. The `callGraphSearch` function returns all loop-free paths from each publish call to each root node (either a callback function or the main function in the node). Function `findPatterns` then analyzes each path, searching for one of
Algorithm 1: Detecting and labeling rate dependency for each outgoing topic in a node n.

1 Function LabelPublishers(n.TO)
2   foreach t ∈ n.TO do
3       t.label ← Rate-Independent
4       foreach path2pub ∈ callGraphSearch(t) do
5          if not findPatterns(path2pub) then
6             t.label ← Rate-Dependent
7             break foreach
8       end
9   end
10 end

the three defined patterns. If a path conforms to a pattern, it is labeled as rate-independent. If it does not, then that path is deemed as rate-dependent, and consequently, the edge is labeled as Rate-Dependent. In the case where all paths to a publisher are labeled to have the rate-independent pattern, then that publisher is labeled as Rate-Independent.

3.2.2 Impact Analysis (IA)

As shown in Algorithm 2 IA takes a set of changed components and the node-rate dependency graph as input. Since changed components may impact the publishing rates, we re-label their outgoing topics as rate-dependent (lines 3-7). Next, we set all the changed nodes as not visited yet (lines 8-10). We then initiate a depth first graph traversal rooted at each changed node (lines 11-15). An impact set, $I_n$, for each changed node ($n$) is computed in the function DFS-VISIT. We generate a global impact set, $I$, by taking the union of all impact sets of every changed component (line 13).

Visiting the graph for a single changed component is shown in Function DFS-VISIT of Algorithm 2. For a component being explored, its neighbors are visited (a) if they are unvisited (line 21), or (b) if the edge dependency between the current component and its neighbor is Rate-Dependent (line 22). Otherwise, the neighbor component is either visited
Algorithm 2: Algorithm for generating impact sets.

Input: 
1. $C$: Set of changed components
2. $NRDG$: Node-rate dependency graph

Output: 
1. $I_n$: Impact set considering a single changed component $n$
2. $I$: Global impact set (a union of impact sets of all changed components)

1. **Function** ImpactAnalysis($C$, $NRDG$)

   1. $I ← \phi$
   2. foreach $Node$ $n ∈ C$ // Reset outgoing topics of every changed node as dependent
      3. do
         4. foreach $Topic$ $t ∈ n.TO$ do
         5. $t$ . label ← Rate-Dependent
         6. end
      7. end
   8. foreach $vertex$ $v ∈ NRDG$ // Mark every vertex unvisited
      9. do
      10. $v$ . visited ← False
      11. end
   12. foreach $Node$ $n ∈ C$ // calculate impact set of each changed node
      13. do
      14. $I_n ← DFSSelect(n, I_n)$
      15. $I ← I \cup I_n$
      16. $I_n ← \phi$
      17. end
   18. return $I$

2. **Function** DFSSelect($n$, $I_n$)

   1. $n$ . visited ← True
   2. foreach $vertex$ $v ∈ adjacent(n)$ do
   3. if $v$ . visited is False then
   4. if IsRateDependent($n, v$) // Expand impact set over Rate-Dependent edges
   5. then
   6. $I_n ← I_n \cup v$
   7. DFSVisit($v, I_n$)
   8. end
   9. end
   10. return $I_n$

3. **Function** IsRateDependent($n, v$)

   1. return edge($n, v$) . label = Rate-Dependent // returns true if the edge is rate-dependent

4. end
already or not impacted through the current component. We add all the components visited by the function DFS-VISIT to the impact set I (line 23). Function IsRateDependent checks whether an edge is labeled as Rate-Dependent.

### 3.3 Implementation and Limitations

Our approach implementation builds heavily on the source code analysis tool Clang (Version 3.9.0) [3], which works as a compiler front-end for C++. We use Clang to help us detect the subscribing and publishing channels and identify the patterns associated with independent publishing edges. More specifically, we use AnalysisDeclContext to generate the code’s Control Flow Graph (CFG), the CFG object for code traversal, CXXMemberCallExpr for detecting member function calls, getArg to retrieve the required argument values, CallExpr to identify regular function calls, CXXCtorInitializer to identify the base or member initializer for ROS objects, and DeclStmt to retrieve variable names. We utilize the YAML-CPP library [12] to store and parse the summaries as YAML files, and PUGI XML [24] for parsing and extracting partial information from ROS launch configuration files. Finally, we use Graphviz DOT [17] to generate visual depictions of the dependency graphs to facilitate their interpretation and debugging of the tool. Our tool RSIA (Rate based Static Impact Analysis) is available for use at [http://nimbus.unl.edu/tools/](http://nimbus.unl.edu/tools/).

In this work, we have attempted to reduce the inheritable limitations of the static program analysis in RSIA. The tool, however, is limited in resolving some C++ language features like templates. Targeting a highly dynamic robotic systems architecture like ROS also limits the implementation when it comes to detecting publishing and subscribing links. ROS provides runtime class loading options which is beyond the scope of static analysis. These dynamically loaded libraries are currently resolved by developer intervention which we assess in the next section. Finally, the precision of the tool can be further increased by extending the ROS
API implementation to include Time Synchronizer, Time Sequencer, Realtime Tools, and Chain.

3.4 Study

To assess the tool that implements the proposed approach, we performed a study on three robotic systems. The study evaluates the tool’s precision and recall when compared with a traditional impact analysis approach, and the ideal implementation of the proposed approach (obtained through a combination of manual and automated analysis). We assessed the proposed approach RSIA in terms of the following research questions:

RQ1: What are RSIA’s precision and recall at capturing topic and rate dependencies?

RQ2: How effective is RSIA compared to existing IA techniques?

3.4.1 Artifacts

Table 3.3 summarizes the artifacts of the three robotic systems, Care-O-Bot (COB) [2], PR2 [6], and aerial water sampler (H2OS) [30]. COB is a mobile manipulation platform. It is approximately 160 cm in height and consists of a moving base, a head, and a torso having arms and grippers attached to it. The arms have seven degrees of freedom. Among its sensors, it includes three laser scanners for obstacle detection and a camera for visual inputs. PR2 is a mobile manipulation platform developed by Clearpath Robotics [6]. Similar

<table>
<thead>
<tr>
<th>System</th>
<th># of nodes</th>
<th># of topics</th>
<th># of launch parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>COB</td>
<td>67</td>
<td>451</td>
<td>926</td>
</tr>
<tr>
<td>PR2</td>
<td>54</td>
<td>132</td>
<td>264</td>
</tr>
<tr>
<td>H2OS</td>
<td>29</td>
<td>50</td>
<td>71</td>
</tr>
</tbody>
</table>
Table 3.4: Different IA techniques compared against RSIA.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Abbreviation</th>
<th>Graph generation</th>
<th>Stopping rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traditional IA</td>
<td>Trad</td>
<td>NDG, Manually</td>
<td>None</td>
</tr>
<tr>
<td>RSIA - Ground Truth</td>
<td>RSIA-GT</td>
<td>NRDG, Manually</td>
<td>Independent Edges</td>
</tr>
<tr>
<td>RSIA - Implementation</td>
<td>RSIA-Tool</td>
<td>NRDG, Statically + Manually</td>
<td>Independent Edges</td>
</tr>
</tbody>
</table>

to COB, it has movable base, with two arms and multiple cameras attached to it. It also uses perception and vision to navigate with the help of several onboard sensors. H2OS (Figure 1.4) is an autonomous drone-based aerial water sampler. It has onboard sensors to navigate and detect its height above water bodies and a motor pump to then take 20ml water samples at programmed locations. H2OS is a product from NIMBUS Lab, UNL. Both PR2 and COB systems are open-sourced, and the three systems are written almost entirely in C++. All three systems are built on top of the ROS communication architecture and library stack, and are being used in industry and research labs.

### 3.4.2 Impact Analyses (IA) Techniques

Table 3.4 presents the techniques we compare against RSIA. We perform a three-way comparison of the generated impact sets. First, we use the traditional impact analysis (Trad) using a node dependency graph (NDG) generated manually which completely represents the node dependencies of the system. Second, we use the proposed approach RSIA using the node-rate dependency graph (NRDG) generated manually, which completely and correctly represents both node and rate dependencies of the system. We call this RSIA-GT, as it act as a ground truth for the automated tool. Finally, we use the automated version of the proposed approach RSIA implemented in a tool (RSIA-Tool) using the node-rate dependency graph automatically generated through static code analysis. RSIA-GT represents the maximum possible impact set reductions if there was an ideal implementation of the proposed technique. RSIA-Tool on the other hand represents the impact set reductions given the
current implementation of the approach.

3.4.3 Study Setup

To answer RQ1, we first evaluate the precision and recall of the tool at generating the node-rate dependency graph, \textit{NRDG}. To do this, we manually generated an \textit{NRDG}. Generating the graph entailed the inspection of each system (source code, launch files, and also runtime publish-subscribe graphs) through a mixed process of automated and manual analysis, intertwining with group code review sessions to analyze code samples and hard-to-determine dependencies. This process resulted in an \textit{NRDG}, with edges labeled as rate-dependent or rate-independent, that we deemed to be correct and treated as the ground truth for the study. We compare this \textit{NRDG} to the one constructed by the tool. We also break down the evaluation among publishers and subscribers that were detected and named by RSIA-Tool to make an impartial assessment.

To answer RQ2, we implemented the traditional IA approach by performing a DFS from a changed node on the manually generated node dependency graph of each system. And we used the manually generated node-rate dependency graph to assess the ideal implementation of our approach (RSIA-GT) since the tool may miss nodes/edges. To evaluate the IA portion of the RSIA-Tool, we used the tool’s generated node-rate dependency graph with additional user input to complete the names of those topics that the tool recognized but could not name unequivocally (because the names were defined in configuration files or used code constructs or API calls not yet supported by the tool implementation).

Finally, to assess runtime performance, we measured the duration of the tool implementing the approach and compared it against the time to compile the systems. We present the findings on runtime performance in Section 3.6.
Table 3.5: RSIA-Tool Edge Detection for Subscribers

<table>
<thead>
<tr>
<th>System</th>
<th>Total</th>
<th>Detected and Mapped</th>
<th>Detected</th>
<th>Undetected</th>
</tr>
</thead>
<tbody>
<tr>
<td>PR2</td>
<td>23</td>
<td>20</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>COB</td>
<td>40</td>
<td>26</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>H2OS</td>
<td>36</td>
<td>32</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

3.5 Results

We focus on the results for the research questions in this section.

3.5.1 RQ1: What are RSIA’s precision and recall at capturing topic and rate dependencies?

We break down these results for publishers and subscribers. For subscribers, we are interested in determining whether we can correctly detect the topics involved. For publishers, we care about topic detection as well as the dependency label assignment. Given this differentiation, we assess them separately.

RSIA-Tool, the automated approach, has 96% recall; that is, it identifies almost all publish and subscribe edges. RSIA-Tool has 100% precision, signifying that all identified publish and subscribe edges were correct. However, some edges are identified, but their names are not mapped to a topic because of certain limitations in the tool that we will discuss next. We examine this more closely by classifying edges into three groups: Detected and Mapped to the right topic, Detected but without a mapping, or Undetected. Unmapped topics are defined separately as the developer can fill the mapping for a detected edge with less efforts than detecting a edge by itself. To support this effort, our tool provides a dump on unmapped topics.

Subscribers. Table 3.5 presents the subscription edge detection information for the three systems. RSIA-Tool detected and mapped 87% of the subscribers with their right mappings.
Table 3.6: RSIA-Tool Edge Detection for Publishers

<table>
<thead>
<tr>
<th>System</th>
<th>Total</th>
<th>Detected and Mapped</th>
<th>Detected</th>
<th>Undetected</th>
</tr>
</thead>
<tbody>
<tr>
<td>PR2</td>
<td>53</td>
<td>50</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>COB</td>
<td>58</td>
<td>45</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>H2OS</td>
<td>35</td>
<td>30</td>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>

for PR2. Topic names for two detected subscribers remained unmapped because their names were provided through a launch file variable within a data structure which is not supported by the current tool implementation. RSIA-Tool missed a subscriber edge because the tool did not have the relevant API call information to retrieve it. For COB, RSIA-Tool detected and mapped 65% of the subscribers. 33% of the subscriber edges were not mapped as their names were defined in launch files. One edge went undetected. Topic mapping at startup is usually done by retrieving the launch file parameter from the ROS Core and then using that string to generate the subscription edge. However, since the launch parameter can be a list or a map, it requires some processing before a subscription call can be made. Even though we can detect when a launch parameter is accessed, such intense tracking of the processing done on the retrieved parameter is not part of the tool yet, and hence we are not able to map the names correctly. For H2OS, the tool detected all edges and mapped 89% of them. The rest had names defined in launch files.

Publishers. Table 3.6 presents the publisher edge detection performance. RSIA-Tool detected and mapped 94% of the publisher edges for PR2. RSIA-Tool missed two publishing edges (4%), again because of a missing API call implementation not registered with Clang. The remaining edge was detected, but the tool was not able to map the right edge name. For COB, the detection and mapping percentage was 69%, mainly because of the use of dynamic configuration options and the use of C++ constructs like pointers to functions that the current tool implementation cannot handle. 23% of COB publishers were detected but unmapped. The last 9% (5) of the undetected topics were caused by the linked libraries
Table 3.7: Edge Classification between the manually generated ground truth (RSIA-GT) and the implementation of the proposed approach (RSIA-Tool)

(a) PR2 - 51 Published Edges Detected

<table>
<thead>
<tr>
<th></th>
<th>A-Tool</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Independent</td>
<td>Dependent</td>
<td></td>
</tr>
<tr>
<td>A-GT</td>
<td>Independent</td>
<td>17</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>Dependent</td>
<td>0</td>
<td>23</td>
</tr>
</tbody>
</table>

(b) COB - 53 Published Edges Detected

<table>
<thead>
<tr>
<th></th>
<th>A-Tool</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Independent</td>
<td>Dependent</td>
<td></td>
</tr>
<tr>
<td>A-GT</td>
<td>Independent</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Dependent</td>
<td>1</td>
<td>38</td>
</tr>
</tbody>
</table>

(c) H2OS - 35 Published Edges Detected

<table>
<thead>
<tr>
<th></th>
<th>A-Tool</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Independent</td>
<td>Dependent</td>
<td></td>
</tr>
<tr>
<td>A-GT</td>
<td>Independent</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>Dependent</td>
<td>0</td>
<td>26</td>
</tr>
</tbody>
</table>

which were precompiled in ROS and the tool had no access to that part of the source code. For H2OS, RSIA-Tool detected all 35 publishing edges and 85.7% (30) were both detected and mapped. The remaining 14.3% of the edges were not mapped correctly as the topic names were not defined in the source code but were read from the launch file parameters.

Table 3.7 presents, for all the analyzed systems, comparison of the label assignments for the detected edges of the publishers\(^1\) between RSIA-Tool and the ones assigned by RSIA-GT as we want to determine how effective the tool is versus the ideal implementation of the approach. For PR2 (Table 3.7a), all 23 dependent labels are recognized as such by RSIA-Tool. However, RSIA-Tool is overly conservative and marks 11 independent edges as dependent. This will end up reducing the benefits of the approach, but it was the result of a conscious trade-off between the tool being more precise versus less complete in the implementation of the edge marking scheme. Table 3.7b presents the label matching results for the COB sys-

\(^1\)Recall that this is only done for publishers as they are the only ones to have rate-dependency labels.
tem. Out of 39 dependent edges, \textit{RSIA-Tool} mismarked one as independent, and out of 14 independent edges, it marked one as dependent. The mismarking of a dependent edge as an independent edge was caused by a dynamically loaded library that was beyond the scope of the tool’s analysis, and we further assess its impact in the next section. For one component, subscribers were defined in the dynamically loaded library that went undetected. Therefore, the publisher that was defined in the analyzed component got labeled as Independent since there were no detected subscribers for the node (conforming to the first pattern). For H2OS (Table \ref{table:3.7c}), all 26 dependent labels are recognized correctly. However, \textit{RSIA-Tool} conservatively marked six (6) independent edges as dependent.

\subsection*{3.5.2 RQ2: How effective is RSIA compared to existing IA techniques?}

Figure \ref{fig:3.3} summarizes the impact set reduction for all three systems. In this figure, the size of the impact set returned by \textit{Trad} is the baseline (100\%). To compute the data in this summary, we executed each approach as many times as there were nodes in a system, assuming that one distinct node changed each time. We compute the ratio between the accumulated size of the impact sets of \textit{RSIA-GT} and \textit{RSIA-Tool} over \textit{Trad}.

For COB, the impact set was reduced to 45\% by \textit{RSIA-GT}, and to 75\% by \textit{RSIA-Tool}. The single false-positive in COB (an edge was declared as independent when it was not) did not have an impact on recall because there were no subscribers to that topic. For PR2, the impact set is barely reduced by either version of our approach as most independent edges belong to components that are not coupled to many other components. H2OS shows the highest impact set reduction. This is in part because \textit{Trad} struggles to provide any reduction as the system data-flow is highly coupled. \textit{RSIA-GT} and \textit{RSIA-Tool} can de-couple some central communication components by defining some edges as independent, reducing the
Figure 3.3: Impact set reduction ratio of \textit{A-GT} and A-Tool over \textit{Trad}.

impact to approximately 40\% of the \textit{Trad} set. \textit{RSIA-Tool} lags behind \textit{RSIA-GT} due to the presence of false negatives. Depending on the location of a false negative, many nodes may get added to the impact set which can highly impact the overall reduction.

We now look at our results in more detail, checking the range of impact set sizes as different components in a system change. Figure 3.4 presents the results for COB, with the components on the x-axis, and the impact set size on the y-axis. Components are arranged in decreasing order of their impacted depth in the Node-rate dependency graph. Impact depth is the size of the longest impact propagating path in the graph. Leaf nodes are not shown as they have no impact when changed. Pluses represent the impact set produced by \textit{Trad}, circles represent the impact set produced by \textit{RSIA-GT}, and crosses represent the size of the impact set generated by \textit{RSIA-Tool}. Five (5) of COB’s 46 nodes, all with large impact sets under \textit{Trad}, are noticeably reduced by the proposed approach. \textit{RSIA-Tool}, however, could not improve on \textit{Trad} when the reported sets had a handful of components. In Figure 3.5 for PR2, we note that the reduction achieved by \textit{RSIA-Tool over Trad} is limited. \textit{RSIA-GT}
Figure 3.4: Impact set size reduction for Care-O-Bot when applying an approach assuming the component in the x-axis is changed.

Figure 3.5: Impact set size reduction for PR2 when applying an approach assuming the component in the x-axis is changed.
Figure 3.6: Impact set size reduction for aerial water sampler when applying an approach assuming the component in the x-axis is changed.

does not provide any reductions either as the code patterns are not observed as frequently. The exceptions (i.e., node 1 presents a reduction from 14 to 10 nodes) occur mostly when sensing nodes in the periphery of the system are changed. Figure 3.6 for H2OS, illustrates yet a different scenario with major gains in reduction independent of what component was changed. The architecture of H2OS is such that most nodes are highly data-coupled but not always rate-coupled, which means that \textit{RSIA-Tool} can provide on average impact sets of less than 8 nodes while \textit{Trad} delivers sets of 19 nodes on average.

3.6 Overhead Analysis

In this section, we look at the efficiency of RSIA-Tool in reporting the reduced impact sets. We measure the tool’s runtime performance regarding the analysis overhead when compared against that of compiling the system (without our analysis). Figure 3.7 shows the time
to compile and analyze each system. The time to analyze a system was computed as the average of the analysis times where each component was assumed to be changed. Our tool took approximately 30% longer than compilation alone, ranging from three to eight extra minutes when it ran on a laptop with Intel i7-2670QM processor (8 cores, 2.20 GHz) running Ubuntu 16.04 with 12 GB RAM. This shows that the tool is highly practical to use given the higher impact set reductions than existing impact analysis techniques.

3.7 Threats to Validity

Threats to external validity concern the generalization to systems other than the ones we studied. In this work, we study three robots that use ROS as a framework. The selected systems and ROS, however, are quite popular and large, covering a range of similar systems. Furthermore, we note that the cost of studying more systems and middleware is non-trivial. It requires extensive and careful manual analysis to determine the ground truth that took
months for the studied systems.

**Threats to internal validity** have to do with how the study was performed. We recognize that analyses involving a manual process are susceptible to bias. We attempted to control that bias by having multiple participants examine sample code. For cases that were hard to interpret, we compared the manual and automated results to address any potential incompleteness in the manually computed node-rate dependency graph. Similarly, the code may exhibit other dependencies that we failed to identify either manually or with the provided tool.

**Threats to conclusion validity** concern our ability to draw accurate statistical conclusions. We want to note that the analysis is deterministic in nature. To statistically compare the impact set reductions for different techniques, we take an average of all impact sets generated assuming each and every node in the system has changed. In the future, we plan to perform a study with a larger set of robotic systems.

**Threats to construct validity** have to do with how we measured the performance of our presented approach. In this work, we focus more towards getting the dependencies out with correct rate-dependency labels. However, since the performance of any impact analysis is judged based on its time consumption, we also performed a timing analysis of the implemented tool for completion.

### 3.8 Summary

We presented a static program analysis based impact analysis technique which is highly useful for distributed software systems that use publish-subscribe message passing models. Robotics systems can specifically utilize this technique as the rate requirements are more safety critical in those systems. We have also implemented a tool for the proposed approach using the compiler frontend *CLang*, targeting robotic systems written using the Robot Op-
erating System (ROS). We performed a study on three robotic systems and analyzed the implemented tool for its precision and recall. The tool shows a one-third increase in compilation time while impact sets can see a reduction of 60%. We present the possible future directions of this work in Chapter 5.
Chapter 4

Dynamic Rate Impact Analysis
(DRIA)

In Chapter 3, we presented a rate based static impact analysis technique, RSIA, that analyzes C++ based source code to extract data and rate dependencies. It is, however, limited in its capabilities to detect: (a) components programmed in other languages, and (2) dynamically established dependencies. Therefore, in this chapter, we present a dynamic program analysis based rate impact analysis (DRIA) technique to complement RSIA. The goal of DRIA is still to report a set of components impacted by a rate change, but we attempt to reduce the impact set size by dynamically isolating impacted outgoing topics based on their rate dependencies on incoming topics. DRIA also reports how rates relate throughout the system using a mathematical model.

Figure 4.1 presents several different rate dependencies scenarios. A connecting line between the incoming and outgoing topic indicates a rate dependency. A broken sign signifies that the outgoing topic is rate-independent of the incoming topic. Traditional impact analysis techniques tend to just consider node (data) dependencies and assume all outgoing topics to be rate-dependent on all incoming topics. This would work well for cases 4.1a and 4.1c.
4.1 because the outgoing topics from those nodes already depend on all incoming topics. However, traditional IA techniques overestimate rate dependencies. For example, they treat rate dependencies of 4.1g as if its 4.1f labeling $t_3$ and $t_4$ dependent on both incoming topics where only one incoming topic is affecting their rates. The static program analysis we introduced in Chapter 3 for rate impact analysis considers rate dependencies where an outgoing topic is rate-independent of all the incoming channels to handle cases like 4.1b and 4.1e. We improve the rate impact analysis by determining which topic affects the rate of an outgoing topic. Our dynamic rate impact analysis approach can distinguish all rate dependencies shown in Figure 4.1.
4.1 Approach

In this section, we present the key aspects of our approach: Throttled Replay (TR), Flow Rate Analysis (FR), Impact Analysis (IA), and Flow Diff Analysis (FD). Figure 4.2 presents the high-level architecture of the proposed approach. TR, FR, and IA together make up the first stage of the approach that reports whether or not a component is affected by a given rate change. FD is the second stage of the approach that reports how a rate change is impacting other components in the system. We now present details of the analysis steps.

4.1.1 Throttled Replay (TR)

The task of TR is to analyze system components and bag files to produce a node (data) dependency graph along with component level bag files. TR analysis takes as input (1) system component executables, (2) system trace files, and (3) a configuration file. It generates (1) a node dependency graph (NDG), and (2) a set of bag files per outgoing topic for each node in the system. FR analysis then utilizes the bag files to extract rate dependencies and update the NDG with extracted dependencies to generate a Node-Rate dependency graph which can be used for impact analysis.
TR analysis does not require instrumentation of system executables; it just records the data available at the incoming and outgoing topic interfaces using ROS based tools. The Bags files are system traces encoded as a stream of <time/value> message pairs for each published topic in the system. Such timed message streams are common in message oriented middleware distributed systems. TR replays these bag files to throttle the incoming message rates on a node. TR then reads the outgoing message rates while throttling the input to a node. Finally, a configuration file is provided by the user to fine tune the analysis. The configuration file may contain, for example, parameters to decide on a throttling sampling strategy or to set the tolerance level for rate changes.

Algorithm 3 presents the pseudocode for TR. TR initializes the node dependency graph, NDG, (line 2) and then populates it as the analysis progresses. TR launches a node, n, and adds its publishing and subscribing edges to the graph (line 5-14). Once the NDG for a node is constructed, TR then determines whether the node is repeatable (line 15).

Repeatability: A node n is repeatable if \( r(t_i) - r(t'_i) < \delta \), \( t_i \in B \), \( t'_i \in n.T_O \).

In this equation, \( \delta \) is the repeatability tolerance limit provided in the input configuration, \( B \) is the input set of bag files, and \( n.T_O \) is the set of outgoing topics of node n. A node is accepted as repeatable when its outgoing topics are within tolerance limits given no throttling of its incoming topics. For any node that fails the repeatability test, TR conservatively assigns its outgoing channels to be rate-dependent on all its incoming channels (lines 17-21). In the worst case scenario, where all nodes fail the repeatability test, the performance of our approach will be equal to that of a traditional IA technique.

For nodes that pass the repeatability test, TR throttles their incoming topics as presented in Algorithm 4. TR uses the set of throttling values from the input configuration to generate a permutation of all the possible throttling scenarios (line 3). As iterating over all permutations can be time-consuming, TR samples a subset of the permutations based on the sampling parameter provided by the user (line 4). TR iterates over the sampled throttling scenarios.
Algorithm 3: Throttled Replay Analysis.

**Input:**
- \( S \): System
- \( K \): Configuration for TR analysis
- \( B \): System level bag files

**Output:**
- \( NDG \): Node dependency graph
- \( \{ b_n | \forall n \in N \} \): Set of all component level bag files

```plaintext
Function ThrottledReplay(S, K, B)
  NDG ← B ← φ                              // for every node n in System S
  foreach n ∈ N                               // add node n to NDG
    do
      launch(n)
      NDG ← NDG ∪ n
    foreach t ∈ n.T_I                          // add node n’s incoming topics to NDG
      do
        NDG ← NDG ∪ t ∪ edge(t, n)
      end
    end
    foreach t ∈ n.T_O                          // add node n’s outgoing topics to NDG
      do
        NDG ← NDG ∪ edge(n, t)
      end
    end
    if is_repeatable(n, B, K) then
      b_n ← b_n ∪ throttle_record(n, B, K)
    else
      foreach t ∈ n.T_O                          // make outgoing topics rate dependent on all incoming topics
        do
          NDG ← NDG ∪ update_edge(n, t, n.T_I)
        end
      end
    end
  end
  return NDG, \{ b_n | \forall n \in N \}
end
```

and records the outgoing topic rates while throttling the incoming topic rates (line 4-9). TR finally returns the set of recorded bag files along with the generated \( NDG \) (Algorithm 3, line 24).
**Algorithm 4: throttle_record.** Throttle the input at different rates while monitoring the rates of outgoing topics for a components.

```plaintext
1 Function throttle_record(n, B, K)
2    \( b_n \leftarrow \phi \)
3    \( P \leftarrow \text{permute}(K, n.T_I) \)
4  foreach \( s \in \text{sample}(P, K) \) do
5      relaunch(n)  // relaunch to avoid contamination
6      replay_bag(B, n.T_I, s)
7      \( b \leftarrow \text{record_bag}(n.T_O) \)
8    \( b_n \leftarrow b_n \cup b \)
9  end
10  return \( b_n \)
11 end
```

### 4.1.2 Flow Rate Analysis (FR)

The flow rate analysis expects as input a node dependency graph with a set of bag files that capture the outgoing topics given the throttled incoming message rates for each node. FR analysis attempts to fit a predefined mathematical model to represent the rate relations between incoming and outgoing topics. The following models define the rate relations between system components and are the most commonly found models in robotic systems.

Our proposed approach utilizes these models to represent the relation between the rate of incoming and outgoing topics.

**Linear Rate Dependency, LRD** Topic \( t_o \) fits an LRD model if \( r(t_o) = \sum_{\exists t \in n.T_I} f(r(t)) \), where \( f(x) = ax + b \).

Here, the function \( f() \) is a linear function of the input rate, \( a \) and \( b \) are constants, and the rate of the outgoing topic \( t_o \) is a linear combination of rates on the incoming topics. The LRD model represents rate dependencies for an outgoing topic when one or more incoming topics contribute linearly to its rate. The outgoing topic \( t_3 \) from Figure 4.11 fits the LRD model as it is rate dependent on both incoming topics \( t_1 \) and \( t_2 \).

Some outgoing topics always publish messages at a constant rate under all conditions.
Rate dependencies for such topics are defined using a constant rate dependency (CRD) model.

**Constant Rate Dependency, CRD**: Topic $t_o$ fits the CRD model if $r(t_o) = c$.

In the foregoing, $c$ is a constant. Topic $t_2$ from Figure 4.1b fits the CRD model. An outgoing topic fitting the CRD model is guaranteed never to propagate the impact of a system rate update. The rate independent edges from our static approach, RSIA, all fit under the CRD model as RSIA only detects edges that are guaranteed to have a fixed/constant publish rate under all conditions.

Distributed system architectures like ROS also contain nodes that perform a synchronizing operation. Such node usually publishes either at the minimum or the maximum of all the synchronized incoming topic rates. We define such rate relations using a min-max rate dependency (MMRD) model.

**Min-Max Rate Dependency, MMRD**: Topic $t_o$ fits the model if $r(t_o) = \text{min}/\text{max}(r(t))$, $\exists t \in n.T_I$.

The MMRD model defines which incoming topics can affect the rate of an outgoing channel. However, only the slowest (if minimum) or the fastest (if maximum) incoming rate topic affects the rate at any given instant. Therefore, it is not guaranteed that an incoming topic will always affect outgoing topics rate values even if the outgoing topic rate depends on it. Figure 4.1c presents topic $t_3$ which fits MMRD model.

Algorithm 5 presents the FR analysis. It expects a system $S$, with a NDG, and a set of bag files for each node, $\{b_n | n \in N\}$. FR analysis then tries to fit each predefined model to every outgoing topic (lines 5-7). A model $M$ can be represented as a data structure that consists of three components: a mathematical equation ($M.\text{eq}$), a set of rate dependencies ($M.\text{RD}$), and an error representing the quality of the fit ($M.\text{er}$). The best fit is calculated based on the minimum error of the models (line 8) as an error value of zero signifies that the mathematical model perfectly captures the rate relation between the outgoing and incoming
Algorithm 5: Flow Rate Analysis
Fits mathematical models on outgoing topics.

\begin{algorithm}
\textbf{Input} : $S$: System
$\{b_n|n \in N\}$: Bag files reported by TR
$NDG$: Node dependency graph

\textbf{Output}: $NRDG$: Node-rate dependency graph

\begin{algorithmic}
\Function{FlowRateAnalysis}{$S$, $\{b_n|n \in N\}$, $G$}
\For{\textbf{each} $n \in N$}
\For{\textbf{each} $t \in n.T_O$}
\State $b \leftarrow b_n[t]$
\State $M_{LRD} \leftarrow \text{LRD}(b)$
\State $M_{CRD} \leftarrow \text{CRD}(b)$
\State $M_{MMRD} \leftarrow \text{MMRD}(b)$
\State $M \leftarrow \min(M_{LRD}, M_{CRD}, M_{MMRD})$ \Comment{select best fitting model}
\State $NDG \leftarrow NDG \cup \text{update\_edge}(n, t, M.RD)$
\State $NDG \leftarrow NDG \cup \text{add\_label}(n, t, M.eq)$
\EndFor
\EndFor
\State \Return $NDG$ \Comment{its updated with rate dependencies therefore becoming a $NRDG$}
\EndFunction
\end{algorithmic}
\end{algorithm}

Algorithm 6: Updated rate dependency check from algorithm 2

\begin{algorithm}
\Function{IsRateDependent}{$n, v, u$}
\State \Return $\text{edge}(u, n) = e \mid \exists e \in RD_{\text{edge}(n, v)}$
\EndFunction
\end{algorithm}

topics. The selected model represents the rate dependencies and the mathematical rate relation of the outgoing topic. Based on the selection, FR updates the $NDG$ with the best mathematical rate relation and that edge’s rate dependency set (line 9-10). \texttt{update\_edge} function defines an edge by its source, destination, and rate dependency set values.

4.1.3 Impact Analysis (IA)

Similar to RSIA, DRIA attempts to generate an impact set using depth first search by traversing the $NRDG$. However, unlike RSIA, which only stores a ‘rate-dependent’ or ‘rate-independent label’ for each topic, DRIA uses a rate-dependency set for every outgoing topic.
This provides a better resolution of rate dependencies than a trivial “independent of all or not” label. Therefore, we modify the IsRateDependent function from Algorithm 2 to accommodate DRIA’s rate-dependency representation. Algorithm 6 presents the new rate-dependency check that returns True when the incoming edge is an element of the outgoing edge’s rate-dependency set. DRIA’s representation of rate-dependency can represent RSIA’s rate-dependency labels. For example: a rate-dependent label is equivalent to an outgoing topic having rate dependencies on all incoming topics and a rate-independent label is similar to not a having a rate-dependency at all. Moreover, DRIA can also distinguish the scenario where an outgoing topic is dependent on some but not all of the incoming topics. This extra level of resolution enables DRIA to provide greater impact set reductions.

4.1.4 Flow Diff Analysis (FD)

Flow-Diff analysis aims to change the impact analysis by reporting how a component that is part of an impact set is affected by a change. Current impact analysis results only include the name of a component that might be affected. Therefore, every component from the impact set needs to be tested to verify the impact. Also, just reporting what components are affected by a change does not help in improving system understanding; for example, what dependencies are propagating the impact, what made a component safe or unsafe, etc.. FD attempts to improve both the expressiveness and precision of impact set results by first generating a Differential Rate Graph (DRG) and then utilizing the DRG to further prune the impact sets.

A Differential Rate Graph is a Node-Rate dependency graph annotated with new predicted rate values and rate dependencies. For any given system update, FD takes the NRDG of the original system $S^i$ and the updated system $S^{i+1}$ and generates an DRG by first annotating the changed components with the new rate values of the components updated
between systems $S^i$ and $S^{i+1}$, and then it propagates these new values using the extracted mathematical models for each topic. This information is annotated on the graph using $< t, \{RD_t\}, r_o \rightarrow r_n >$ labels, where $r_o \rightarrow r_n$ captures the change in rate value of a topic $t$ from $r_o$ to $r_n$, and $RD_t$ represents the set of rate dependencies for topic $t$.

Adding both $RD_t$ and $r_o \rightarrow r_n$ helps improve the expressiveness of impact analysis results. We utilize this expressiveness of the $DRG$ to improve impact set reductions by also pruning impact space when the new rate value has not changed despite having a rate dependency on an affected incoming topic. One example of such a scenario is a synchronizing node. If the node’s outgoing rate is limited by the slowest incoming topic, changing any other incoming topic’s rate will not affect the outgoing rate as long as it’s rate value is still higher than the rate of slowest incoming topic. Therefore, further impact propagation can be limited at such components providing us with more precise impact sets than $NRDG$ alone. We update the Rate-Dependency check for IA to incorporate this as shown in Algorithm 7.

4.1.4.1 Use cases for a Differential Rate Graph ($DRG$)

The $DRG$ captures the system’s topic rate state before and after an update. We use these states to propose rate optimizations in the system. For example, if we have a component chain with data dependencies in $n_1 \rightarrow t_1 \rightarrow n_2 \rightarrow t_2 \rightarrow n_3 \rightarrow t_3 \rightarrow n_4$ such that $r(t_1) < r(t_2) > r(t_3)$, then we propose reducing the rate of topic $t_2$ as it will free up resources for other tasks in the system. Such rate patterns can be defined to examine data dependency paths and autonomously propose rate optimizations.

The $DRG$ offers quick feedback describing why or why not the impact was propagated...
Algorithm 8: Repeatability Test: To make sure that the component is properly setup before we try to throttle the incoming topics.

```plaintext
1 Function is_repeatable(n, B, K)
2    set_params(n)
3    launched ← launch(n)
4    while launched do
5        if launched.T₀ = n.T₀ then
6            replay_bag(n.Tᵢ, throttle ← 1)
7            b ← record_bag(n.T₀)
8            if r(b.T₀) - r(n.T₀) < K.δ then
9                return true
10           else
11                return false
12        end
13    else
14        launched ← launch(n.neighbor)
15        if launched = False then
16            return false
17    end
18 end
```

through an outgoing topic by presenting both rate differences and the rate dependency set in the results. The DRG can also assist in: (1) performing a pseudo rate update, and (2) verifying an ROS Hz test without running a test suite. A pseudo update is possible when a developer knows what rate changes are being planned and wants to check whether they will leave the system in a stable state. ROS Hz tests, on the other hand, require system execution. FD analysis, however, only executes the changed components and propagates the new rate values for each topic in the system. These rate values can be used against the ROS Hz test, greatly reducing the need for system execution. The DRG can save development time if an update is going to break other rate properties and can also be a safer alternative than the dynamic ROS Hz tests. We further discuss these benefits through a case study of Care-O-Bot in Section 4.4.3.
4.2 Implementation and limitations

We have implemented DRIA using ROS Bag tools and Bash shell scripts combined with the updated IA tool from RSIA targeting distributed systems developed with ROS. DRIA leverages a number of ROS tools for recording and analyzing traces. It utilizes rosbag’s record feature to record system traces, the play feature for replaying the system traces, the filter feature to separate different topics from a trace containing multiple topics, and finally the info feature to extract the rate information for a given topic. We used roslaunch to launch every component in isolation. We use yaml-cpp to read and write NRDG into files and use the Graphviz DOT to generate a visual depiction of the dependency and flow graph. We use MATLAB for the model fitting flow rate analysis.

During the implementation, we observed that the repeatability test for a node sometimes fails due to reasons other than the rate of the outgoing topic exceeding tolerance limits. One such example is a node waiting for the environment to be set up. This environment setup might mean that a node waits for either a launch parameter to be available or another node/service to exist. Our approach towards the repeatability test is shown in Algorithm. We start by setting all the parameters read from the system launch file (line 2). Then, we try to analyze system components in isolation. However, when a component fails the repeatability test, we launch its neighboring nodes to help complete the environment (line 14-19). We stop when: (1) the neighboring nodes cannot be launched anymore, or (2) the node being tested registers and starts publishing on all its outgoing topics.

One limitation of this proposed technique comes from the system components that either rely on user input or external hardware to be available. In such cases, the technique fails to analyze the components. By default, our technique labels each outgoing topic of a node to be rate dependent on every incoming topic of that node. This conservative approach towards labeling outgoing topics in non-repeatable nodes reduces its precision, but it is more
important to be conservative in this setting. We note that deeming a component dependent when it is not will lead to larger impact sets, but no impacted components will be shipped.

4.3 Study

We performed a study to evaluate the proposed approach through an implementation that targets distributed systems written using the Robot Operating System (ROS). We assessed DRIA in terms of the following research questions:

**RQ1:** What are DRIA’s precision and recall at capturing node and rate dependencies?

**RQ2:** How effective is DRIA when compared to other IA techniques?

For the additional differential rate graph generated by DRIA, we performed a case study to analyze and emphasize the usability of the graph. Finally, we present an overhead analysis for the proposed technique in Section 4.5.

4.3.1 Artifacts

Table 4.1 summarizes the artifacts of the two robotic systems, Care-O-Bot (COB, simulated) [2] and water sampler (H2OS) [30]. Dynamic program analysis requires execution of a system for analysis. However, with DRIA targeting robotics system, access to the hardware is required to generate system trace files for the analysis. As we do not have access to the hardware of the COB robot’s hardware, we decided to analyze its simulation stack which is equally as complex. We execute the robot in simulation and collect trace files as the robot

<table>
<thead>
<tr>
<th>System</th>
<th># of nodes</th>
<th># of topics</th>
<th># of launch parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>COB</td>
<td>67</td>
<td>451</td>
<td>926</td>
</tr>
<tr>
<td>H2OS</td>
<td>29</td>
<td>50</td>
<td>71</td>
</tr>
</tbody>
</table>

Table 4.1: Attributes for the analyzed systems.
Table 4.2: Different IA techniques compared against DRIA. $t_o$ represents an outgoing topic, $t_i$ represents some incoming topic, and $R_D$ that provides rate dependency set of a topic

<table>
<thead>
<tr>
<th>Approach</th>
<th>Abbreviation</th>
<th>Graph generation</th>
<th>IA stopping rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traditional IA</td>
<td>Trad</td>
<td>NDG, Manually</td>
<td>None</td>
</tr>
<tr>
<td>DRIA Ground Truth</td>
<td>DRIA-GT</td>
<td>NRDG, Manually</td>
<td>if $t_i \notin R_D(t_o)$</td>
</tr>
<tr>
<td>RSIA Implementation</td>
<td>RSIA-Tool</td>
<td>NRDG, Statically + Manually</td>
<td>if $r(t_o) = \text{constant}$</td>
</tr>
<tr>
<td>DRIA Implementation</td>
<td>DRIA-Tool</td>
<td>NRDG, Dynamically</td>
<td>if $t_i \notin R_D(t_o)$</td>
</tr>
</tbody>
</table>

performs various predefined tasks. H2OS, on the other hand, is developed at the NIMBUS Lab, which gives us access to the real water sampling robot and its trace files from various experiments and demos. We selected trace files of the most recent successfully executed water sampling mission for this analysis.

4.3.2 IA Techniques

Table 4.2 presents the techniques we compare against DRIA. We perform a four-way comparison of the generated dependencies and impact sets. First, we manually generated an NDG to simulate a traditional impact analysis technique (Trad) that only looks at system node dependencies for impact analysis. Then, we compared our results against RSIA’s implemented (RSIA-Tool), the static rate impact analysis approach described in Chapter 3. RSIA-Tool extracts a node-rate dependency graph (NRDG) using static program analysis of components written in C++. The gaps in RSIA-Tool’s generated NRDG, components and dependencies it fails to detect, were filled in manually. For DRIA, we manually generated an NRDG through a rigorous semi-automated study of the source code, launch file, runtime publish-subscribe graph, and execution of system nodes in isolation. These node and rate dependencies here are deemed ground truth values, and therefore are named DRIA-GT. Finally, DRIA’s implementation (DRIA-Tool) also generates a NRDG using dynamic program analysis. We compared DRIA-Tool with DRIA-GT to calculate the tool’s precision and recall, and its ability to achieve possible impact set reductions.
Table 4.3: DRIA: Subscriber detection

<table>
<thead>
<tr>
<th>System</th>
<th>Total</th>
<th>Detected</th>
<th>Undetected</th>
</tr>
</thead>
<tbody>
<tr>
<td>COB</td>
<td>286</td>
<td>284</td>
<td>2</td>
</tr>
<tr>
<td>H2OS</td>
<td>66</td>
<td>63</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 4.4: DRIA: Publisher detection

<table>
<thead>
<tr>
<th>System</th>
<th>Total</th>
<th>Detected</th>
<th>Undetected</th>
</tr>
</thead>
<tbody>
<tr>
<td>COB</td>
<td>376</td>
<td>376</td>
<td>0</td>
</tr>
<tr>
<td>H2OS</td>
<td>54</td>
<td>48</td>
<td>6</td>
</tr>
</tbody>
</table>

4.3.3 Study setup

To answer RQ1, we present an analysis of the implementations’ capability to detect both node and rate dependencies with respect to the ground truth. We also present our analysis of the tool’s ability to detect node dependencies. To answer RQ2, we compare the precision and recall of the techniques at reporting impact sets for a given change. Since RSIA-Tool requires a developer to fill in the gaps in the dependency graph, we present RSIA-Tool with (1) gaps filled in and allocating a conservative rate dependency label (dependent on all incoming topics) for the missed outgoing channels (minimal developer effort), and (2) gaps filled in with the ground truth values (maximum developer effort). Finally, we present a case study on Care-O-Bot to demonstrate the use of the differential rate graphs on real-world examples.
4.4 Results

4.4.1 RQ1: What are DRIA’s precision and recall at capturing topic and rate dependencies?

One of the first steps of DRIA is to generate a complete graphical representation of the node dependencies by detecting subscriptions (incoming topics) and publishers (outgoing topics) registered by a node. Table 4.3 presents the results for detected subscribers in each system. For H2OS, DRIA detected 63 out of 66 subscribers, missing only three subscribers. It was unable to detect three subscribers as their parent nodes crashed when replayed, expecting a hardware resource to be available. In COB, DRIA detected 284 out of 286 subscribers. A node in COB waits for a service request before registering some of its subscribers. Since no such service calls were mimicked by DRIA-Tool, the approach failed to detect the two subscribers linked to that node. DRIA requires a system to be executed: (a) to collect input trace file(s), and (b) to replay and record a component’s behavior after input rate throttling. As we do not have access to COB’s Hardware, DRIA used a different configuration of Care-O-Bot. Therefore, the total number of subscribers and publishers are different from the data presented in Section 3.5.1.

Table 4.4 presents the publisher detection results for DRIA. The tool detected all 376 publishers of COB with 100% precision. Unlike RSIA, we categorize publisher detection only into Detected and Undetected as DRIA always maps the correct topic name to a detected subscriber. For H2OS, the six undetected publishers came from the same set of nodes that crashed before waiting for a hardware resource to become available. As stated in Chapter 3, RSIA, on the other hand, only detected 102 out of 376 publishers in COB and failed to detect publishers that were dynamically established or originated from components written in languages other than C++. DRIA’s high precision comes from its capability to both (1)
Table 4.5: Outgoing topic classification. Topics that are rate dependent on at least one incoming topic are shown as $D_t$. Topics that are rate independent of all incoming topics are shown as $I_t$.

(a) Water Sampler. RSIA only detected 34 out of 54 outgoing topics.

<table>
<thead>
<tr>
<th></th>
<th>DRIA-Tool</th>
<th>RSIA-Tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>$I_t$</td>
<td>29</td>
<td>10</td>
</tr>
<tr>
<td>$D_t$</td>
<td>2</td>
<td>13</td>
</tr>
</tbody>
</table>

(b) Care-O-Bot. RSIA only detected 102 out of 374 outgoing topics.

<table>
<thead>
<tr>
<th></th>
<th>DRIA-Tool</th>
<th>RSIA-Tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>$I_t$</td>
<td>310</td>
<td>0</td>
</tr>
<tr>
<td>$D_t$</td>
<td>36</td>
<td>30</td>
</tr>
</tbody>
</table>

Detect runtime established links, and (2) understand components irrespective of the language of their origin.

Once the approach generates the node dependency graph, it attempts to extract rate dependencies to produce a node-rate dependency graph ($NRDG$). Table 4.5 presents the comparison of DRIA-GT with DRIA-Tool and RSIA-Tool on rate dependency detection for both systems. Label $D_t$ represents cases in which the technique has detected that topic $t$ has at least one rate dependency on the incoming topics i.e. $|RD_t| > 0$. $I_t$ represents cases in which topic $t$ is rate independent of all incoming topics i.e. $|RD_t| = 0$. Table 4.5a presents the comparison of rate dependencies detected by both DRIA-Tool and RSIA-Tool when compared to DRIA-GT for the H2OS robot. DRIA-Tool misassigned rate dependencies to ten topics, all of them conservatively labeled as dependent when they were independent. These misassignments happened due to two reasons. First, a node (with four outgoing topics) only publishes after receiving a start message on one of the incoming topics. Our arbitrary selection of bag files did not include the start message scenario. Therefore, the node failed...
the repeatability test and DRIA conservatively made all outgoing topics rate dependent on every incoming topic. Second, some nodes crashed due to missing hardware and failed the repeatability test. Therefore, DRIA conservatively labeled the corresponding six outgoing topics as rate dependent. DRIA-Tool mislabeled two dependent edges as independent. For one node in H2OS, the selected bag files executed only a part of the source code, which lead to no messages being published on two of the outgoing topics. However, the node still passed the repeatability test as the input bag files also show zero messages published on the topics. For topics with no published messages, DRIA-Tool assigned CRD model assuming a constant publishing rate of zero hertz. RSIA-Tool, on the other hand, only detected 34 out of 54 outgoing topics, however, it assigned correct rate-dependencies among the detected topics. RSIA is also conservative in nature and only labels a topic rate-independent when it can be guaranteed. RSIA-Tool conservatively mislabeled ten topics as rate dependent that were rate independent.

Table 4.5b presents the rate dependency confusion metric for the Care-O-Bot system. DRIA-Tool correctly identified all rate-independent topics for COB and approximatively mislabeled 36 outgoing topics rate-independent that were rate-dependent (90% precision). DRIA-Tool’s limitation here is associated with the bag files. In COB, for some nodes, a part of their source code does not get executed and therefore, they do not publish on some of their registered topics even when there is a rate dependency. Since both the provided bag files and the repeatability test display no messages published on the corresponding topics, DRIA labeled them as independent of incoming topics assigning a CRD model with constant $c = 0$.

In comparison with DRIA-GT, DRIA-Tool had a precision of 0.9 and a recall of 0.97. Whereas, the RSIA technique showed a higher precision of 1, its recall was lower (0.23). Also, note that DRIA required less developer input to fill up the gaps in the automatically generated $NRDG$ and had higher recall values than RSIA. DRIA had an inherent capability
of detecting runtime established links as well as the ability to analyze components irrespective of their programming language.

4.4.2 RQ2: How effective is DRIA when compared to other IA techniques?

Figure 4.3 presents the impact set reduction for both analyzed systems. In the figure, the baseline is represented by the Traditional analysis (Trad). Just as in Chapter 3, we performed impact analysis assuming one node has changed and repeated the process for all nodes in the system for each approach. To obtain the data presented in Figure 4.3, we computed the average size of the impact sets reported by each approach. The dashed line represents the impact sets reported by DRIA-GT. The impact sets for RSIA-Tool are reported for two cases: (1) when a human manually fills in the gaps in the dependency graph with the true
values of the rate dependencies (solid line), and (2) when the user conservatively sets those missing topics to be rate dependent on all incoming topics (overlapped dotted bar on the solid line bar).

In Figure 4.3, data for DRIA-GT shows that impact sets were reduced to 23% for COB, while DRIA-Tool reduces the impact sets to 22%. DRIA-Tool overestimated the impact set reduction due to falsely labeling some rate dependent topics as rate-independent, leading to more impact set reductions. DRIA-Tool’s results can be removed by making more bag files available to the analysis. However, given that no messages were published on certain topics in the provided bag files, we can argue that such topics are in fact rate-independent as long as the system’s launch conditions remain the same. RSIA-Tool, without a developer filling in the gaps with true values, showed no impact set reduction. This is due to COB’s simulated system having a cycle in its NRDG causing all components to be connected. However, when a human fills in the gap, RSIA-Tool reduced the impact set to 37%. Impact set reductions reported by DRIA-Tool were 40% better than that of the RSIA-Tool with the developer’s help filling in gaps.

For H2OS, DRIA-Tool reported reduction in impact set to 23%, DRIA-GT on the other hand reduced impact sets to 25%. We found that for two topics, DRIA-Tool had fewer rate dependencies when compared to DRIA-GT. In H2OS, two nodes performed a multiplexer operation, and as the bag files did not execute all scenarios, DRIA-Tool was unable to detect the complete set rate dependencies.

Figures 4.4 and 4.5 present a component level comparison of impact sets for each system. The X-axis represents the nodes ordered in the decreasing order of their impacted depth in the dependency graph. The Y-axis represents the size of their corresponding impact set. The figure shows the impact sets for Trad (pulses), DRIA-GT(circles), RSIA-Tool (crosses), and DRIA-Tool (triangles). We only present the impact sets for RSIA-Tool when the gaps in

---

1Depth is the length of the longest path reported by DFS while calculating impact set.
Figure 4.4: Impact set size reduction for COB when applying an approach assuming the component in the x-axis is changed.

Figure 4.5: Impact set size reduction for H2OS when applying an approach assuming the component in the x-axis is changed.
its dependency graph are filled with true rate dependencies. Figure 4.4 presents the impact sets for the COB system. We can see that the impact set reduction was significant when the original impact set was large in Trad, and less important when there were a few nodes in Trad’s impact set. The node-wise impact set reduction for H2OS (Figure 4.5) shows that Trad’s impact set for each component was as large as the number of nodes in the system, as they are highly data-coupled. For H2OS, DRIA-Tool isolated the system providing a notable impact set reduction reporting only 3.8 nodes as impacted on average whereas Trad reported 16.5 nodes and RSIA-Tool reported 6.8 nodes on average as impacted.

4.4.3 Case Study: Care-O-Bot

The objective of this study is to showcase the usability of the Differential Rate Graph (DRG), generated by the FD analysis, using real world use cases. For this study, we selected the use cases by polling COB’s Bug/Issue tracker on GitHub with synonyms of rate/frequency and related terminology. We selected the two cases that were only targeting a rate change in the system. Using the selected cases, we showcase how DRG can be used for obtaining quick update feedback, automatically proposing topic rate optimizations, and pseudo system rate update verifications.

Table 4.6 presents the two cases of COB’s driver stack. The two studied cases arise from different packages of the same cob_driver stack. The last column of the table presents a description of the issues originally reported on the COB’s GitHub issue tracker calling for action to perform rate changes in the system.

In the first case [1], a developer reported that all the outgoing topics were published at the same rate, which was not required, and proposed to reduce the rate of a topic $t_5$ down from 100 hz to 0.5 or 1 hz. In the second case [4], a developer, while bug fixing, noticed a parameter loop_rate set to 100. Based on the developer’s knowledge of the system, he
Table 4.6: Case study details. Bug reports are taken from COB’s issue tracker on GitHub

<table>
<thead>
<tr>
<th>Stack</th>
<th>Package</th>
<th>Issue #</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cob_driver</td>
<td>cob_base_drive_chain</td>
<td>207</td>
<td>The base_drive_chain diagnostics message is published with 100Hz. This is</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>not necessary and should be reduced. I think every half second or second is</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>absolutely enough and could be easily done with a simple ros timer...</td>
</tr>
<tr>
<td>cob_driver</td>
<td>cob_scan_unifier</td>
<td>248</td>
<td>... +Yes, the rate of 100Hz is (considerably) faster than the scanner publish</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>rate (which is 12Hz for the S300, afaik). However, this is on purpose to send</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>out the combined scan asap. ...</td>
</tr>
</tbody>
</table>

requested a throttling down of the loop_rate as all the incoming topics were maxed out at around 12 hz making it suboptimal to run the loop at 100 hz. We now look at both cases in depth and show how a DRG can be helpful in such rate related system updates.

Figure 4.6 presents a differential rate graph for COB after the above-mentioned system update for the first case. The star represents the changed component. Circles and rectangles represent nodes and topics, respectively. Links to the unaffected parts of the system are shown in gray. DRIA’s FD analysis (1) generated this DRG using the precomputed NRDG before the system update, and (2) analyzed the changed components for updating the rate dependency information. It then propagated the information using both the pre-extracted and newly extracted mathematical rate models to predict the new rate value for each topic.

The DRG in Figure 4.6 shows the change in the outgoing topic $t_5$ of the updated component (star). Topic $t_5$ is labeled “$< t_5, \{\phi\}, 100 \rightarrow 1 >$”, indicating a rate change from 100hz to 1 hz which was proposed in the issue tracker. Also note that the immediate next data-dependent topics of $t_5$ are topics $t_8$ and $t_9$. New rates for these topics have not changed since both of them are rate-independent given that $RD_{t_8} = RD_{t_9} = \phi$. This informs us that: (1) the rate update for topic $t_5$ was successfully updated, and (2) there is no unintentional
impact of the update as the impact was stopped by both \( t_8 \) and \( t_9 \) from propagating any further. This verifies that the developer’s intuition about the rate optimization was correct.

The DRG shown in Figure 4.6 can also be used to automatically propose the rate optimization mentioned in [1] which was made by a COB system developer. It is achievable by analyzing the DRG to look for rate patterns that suggest rate optimization. To calculate the optimization for topic \( t_5 \), we first list its data dependent topic chain, which is of the order \{\( t_1, t_2 \)\} → \{\( t_5 \)\} → \{\( t_8 \)\} → \{\( t_9 \)\}. Then, if we replace the data-dependencies with rate inequalities, the order becomes \{\( t_1, t_2 \)\} < \{\( t_5 \)\} > \{\( t_8 \)\} = \{\( t_9 \)\}. The inequality clearly shows that topic \( t_5 \) has higher rates than its neighbors and there might be a potential benefit in reducing the rate as it might free bandwidth or CPU processing time. Such rate patterns are not always limited to just neighbors and can be found using long dependency chains. We plan to explore this in future and develop rate patterns that can identify sites for optimization.

Figure 4.7 presents the rate flow graph for the second case. Here, a developer had requested to turn down the loop rate for publishing to topic \( t_8 \). However, topic \( t_8 \) does not show any rate change after the update. This happened because the changes made updated only the node’s internal behavior. Note that the original rate of topic \( t_8 \) was also 12 hz,
different from what was mentioned in the issue tracker. Upon taking a deeper look at the source code and the development of the issue over time, we found that the mentioned parameter *loop_rate* was used by the node only to send out a message as soon as at least one message has arrived on all three incoming scanner topics $t_4$, $t_5$, and $t_6$. For the most part, the loop will just reiterate after checking the status of the incoming channels without doing any processing. As there is no easy way to figure out when the scan messages from each topic will arrive, the developers had chosen to run the synchronizer loop at the much faster rate (100 hz) than the scanning rate of the hardware (12 hz) to ensure that the synchronized message’s data will be updated with the latest information as soon as the last messages arrives. Also, COB contains over 926 parameters that are set at launch time. This level of complexity can be overwhelming for a human to follow. We noticed that the issue was resolved by changing the node’s internal implementation to drop the loop_rate parameter, but the proposed rate change never came in existence. Therefore, we conclude that the developer’s rate assumption was wrong.

A *DGR* could have been used here by performing a pseudo system rate update by passing the new rate value of the intended topic to FD analysis. FD analysis would then propagate the rate change and update rates for other topics. In the case of [4], performing a pseudo
rate update would have reported a DRG exactly the same as shown in Figure 4.7, suggesting that as far as the rate of topic $t_8$ is concerned, no rate updating system change was needed.

### 4.5 Overhead Analysis

DRIA requires intensive execution and is highly dependent on the input configuration. In this section, we present an upper bound complexity analysis for the time taken by DRIA to analyze a system.

The most expensive part of DRIA is its throttling inputs at different rates. This cost is affected by several factors including the number of incoming topics ($n.T_I$), the set of provided BagFiles ($B$), the set of throttling rates ($R$) and the number of nodes in the system $N$. DRIA generates a power set $\mathcal{P}$ of incoming topics creating different throttling settings. For each setting, DRIA throttles every topic in a setting for every throttling rate creating multiple throttling configurations i.e. $|e|^{|R|}$ where $e \in \mathcal{P}(n.T_I)$. All the input bag files are replayed for each throttling configuration. Note that DRIA records all outgoing topics for all throttling configurations, thus the number of outgoing topics does not increase the state space. The upper bound on input state space size $S_n$ for a node $n$ can be represented as

$$O(S_n) = \left\{ \sum_{e \in \mathcal{P}(n.T_I)} \left\{ \sum_{b \in B} (|e|^{|R|} \ast b) \right\} \right\}$$

For a system, $S$ with $|N|$ nodes, the overall upper bound becomes $|N| \ast O(S_n), \forall n \in N$.

To control DRIA from exploding at runtime, we control the complexity of the analysis by sampling throttling configuration instead of performing exhaustive throttling of all throttling configurations. Note that the average number of incoming topics is 3.67, which can be easily throttled in significantly less time without any sampling. For both H2OS and COB, we saw a maximum of 40 incoming topics, but only for one node. Also, incoming topics on which no
messages were published do not affect the confrontational state space as they get dropped from the Bag Files.

4.6 Threats to Validity

Threats to external validity concern the generalization of the proposed approach to other systems and contexts. We only analyze two robotic systems with DRIA. This is susceptible to generalization bias as one of the analyzed systems, COB, used the compilation stack rather than the hardware stack. This was done due to lack of access to COB hardware to perform dynamic analysis. However, note that a majority of packages are commonly deployed between a hardware execution and software simulation. We do consider extending our analysis to more hardware based executions.

Threats to internal validity have to do with how the study was performed. To avoid input bias, we arbitrarily collected system traces. For COB, we collected bag files for the default system launch configuration files provided by the system developers. For the H2OS system, we selected the bag file from the most recent successful water sampling experiment as it best represents the system in a stable state. To mimic a majority of possible rate changes on the incoming topics, we uniformly selected throttling rates that both increase and decrease the rate of a topic, and we also avoided selecting values that are symmetric in nature; for example 2 times, 4 times, 8 times, etc.. Finally, we attempted to maintain diversity in throttling by uniformly sampling the generated throttling configurations.

Threats to construct validity have to do with the metrics used to measure the performance of our approach. While analyzing data dependency detection for DRIA, we categorize the publisher detection into only two categories: ‘Detected’ and ‘Undetected’. We dropped the category ‘Detected and unmapped’ because DRIA is inherently capable of always mapping the publisher once it is detected. To avoid biasing results towards those for RSIA in
impact set comparisons, we filled in the gaps in the RSIA’s automatically generated Node-rate dependency graph, as mentioned in Chapter 3.

4.7 Summary

We presented a dynamic program analysis based impact analysis technique which works without any source code instrumentation and uses only pre-recorded bag files for analysis. The technique throttles the incoming message rates for a node while monitoring its outgoing topic for rate changes. The changes in the outgoing topic's rate are then used to define a mathematical relationship between the incoming and outgoing topics of a node and are also used to assign rate dependencies which are later used for impact analysis. We have implemented a tool for the proposed approach that targets robotic systems developed using ROS. We performed a study on two robotic systems and analyzed the implemented tool for precision and recall. The tool showed impact set reductions reaching approximately 80%. We also presented the impact analysis based differential rate graph that captures the system states before and after a system update precisely reporting how a change can affect other components of the system. We present potential future directions of this work in Chapter 5.
Chapter 5

Conclusions and Future Work

In this thesis, we presented two approaches to support developers of robotic systems in understanding the subtle impacts of code changes that affect the rate at which data is produced or consumed. Both approaches are more precise than existing impact analysis techniques. We have shown their potential through manual examination and with automated tools for ROS.

We first presented a rate based static impact analysis technique, RSIA. RSIA uses static program analysis to detect both data and rate dependencies among system components. Once the rate dependencies are detected, RSIA performs impact analysis while optimizing impact sets. This is done by limiting exploration of the dependency graph to the rate-dependent edges. In the three studied systems, the tool reduced the impact sets by up to 41% compared to existing IA techniques, thus greatly reducing the developer’s effort required to revalidate a system.

The approach and tool, however, are still at an early development stage. The approach could incorporate a richer set of patterns, including those that attempt to synchronize different communication channels, concurrency publishing patterns, and special real-time publishing patterns. The tool could also be improved by adding support for dynamic library
detection and by performing a more precise code analysis. We are also interested in extending the approach to analyze the effect of changes on system performance, as well as exploring the potential of incorporating dynamic analysis to improve the effectiveness of the approach. We will be exploring such improvements and further applying the tool to a larger number of systems.

Second, we also presented another rate based impact analysis technique that uses dynamic program analysis, DRIA. DRIA extracts both data (topic) and rate dependencies to generate a node-rate dependency graph (NRDG). An NRDG represents rate dependencies using a predefined set of mathematical models. DRIA uses the dependency information to improve impact set reductions. Studies from two robotic systems show that DRIA can provide reductions in impact set size by up to 80% when compared with traditional IA techniques and is approximately 40% optimal compared to RSIA. DRIA also reports how the impact of a change propagates throughout the system using a differential rate graph. We also performed a case study of COB bugs/optimizations which emphasizes the usability of differential rate graphs for scenarios like system update verification, rate optimization proposals, pseudo rate updates, and rate test case validation.

The current implementation of DRIA is fully automated and requires minimal interaction with the developer. However, the approach can be extended to improve handling of nodes that are not fully repeatable and have external dependencies. We plan to study more robotic systems and discover other possible mathematical models that may improve the precision of rate relationships. We are interested in exploring these potential research directions and applying the tool to a larger number of systems. We further believe that a tool for automatically reading rate flow graphs can be utilized to provide optimization suggestions, rate test validation, system integrity checks, etc. We wish to explore this further by performing additional case studies on robotic systems.

Finally, one of the more crucial future directions is to fuse the two complementary tech-
niques. For example, DRIA can help improve detection of runtime established links and RSIA has the capability of analyzing components that have external dependencies. We strongly believe that a hybrid approach to rate based impact analysis can further improve both the efficiency and effectiveness of the current techniques.
Bibliography

[1] base_drive_chain’s diagnostics message is published with 100 hz # 207.  


[4] [cob_scan_unifier] update rate # 248.  
https://github.com/ipa320/cob_driver/issues/248

[5] Matlab and statistics toolbox release 2016b, the mathworks, inc, natick, massachusetts, 
united states.


[9] Mithun Acharya and Brian Robinson. Practical change impact analysis based on static 
program slicing for industrial software systems. In Proceedings of the 33rd International 
Conference on Software Engineering, ICSE ’11, pages 746–755, New York, NY, USA, 
2011. ACM.


