Rate Impact Analysis in Robotic Systems

Nishant Sharma, Sebastian Elbaum and Carrick Detweiler

Abstract— **Changes to robotic systems as they are updated or upgraded often affect the flow of control and sensor data. Developers and users spend a significant amount of time tracing the impact of these changes that could otherwise have negative impacts on the robot's performance and behavior. 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. In this work, we develop and implement an approach to help identify the set of components whose rate may be impacted by a system change. The approach builds on the insight that certain code patterns render component's outgoing data rate independent of the component's incoming data rate. We use that insight to reduce the number of components reported as affected by the change to minimize the number of components that must be reevaluated by the developer. A study of an implementation of the approach on three ROS systems shows that it can reduce the size of the impact set by up to 41% in cases when the changes have broad data impacts. The analysis is performed at compile time and only adds a third more to the compilation time.**

I. Introduction

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 [1] in Figure 1. In this system, replacing an arm position encoder with a higher resolution sensor, but with a lower data rate, may result in a position controller instability. Or 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 worsen feedback to a remote operator if WiFi bandwidth is exceeded. Various approaches for handling this have been proposed including Paikan et al. [18] who uses run-time channel prioritization for components that are time-critical and require higher controlling rate. However, few researchers have addressed the general problem of analyzing the impact of rate changes.

Clearly, in many robot 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 the rate changes in the system could affect its performance and lead to incorrect behaviors. More specifically, we would like to know how rate changes propagate through the system so that those areas that may be affected are examined and validated more carefully.

Fig. 1: Care-O-Bot (COB) [1], a ROS based mobile manipulation robot.

Techniques to understand the impact of a change fall under the umbrella of *Impact Analysis (IA)* [5]. These techniques generally identify dependencies among code entities and traverse those dependencies starting from a changed location to determine the set of impacted code entities. Existing IA techniques have focused exclusively on data and control dependencies in code.

In this work, we add the rate dimension, which is particularly relevant to robotic systems that explicitly rely on timing properties or implicitly rely on rate assumptions. We build on the insight that certain code patterns render component's outgoing data rate independent of its incoming data. We use that insight to reduce the number of components reported as affected by the change. 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 approaches. The analysis incorporates component's 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 [1], PR2 [3], and H2OS [17]). The study shows that the approach has the potential to reduce the size of the impact set 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.

II. Motivating Example and Approach Intuition

Figure 1 shows COB [1], a robot designed by Fraunhofer IPA to be a human assistant in a variety of settings. COB sits on a movable base, utilizes lasers and cameras to scan

The authors are with Department of Computer Science and Engineering, University of Nebraska - Lincoln, Lincoln, NE 68588, USA. Email:{nsharma, elbaum, carrick}@cse.unl.edu. This work was partially supported by National Science Foundation under awards #1526652 and #1638099, and USDA-NIFA #2013-67021-20947.

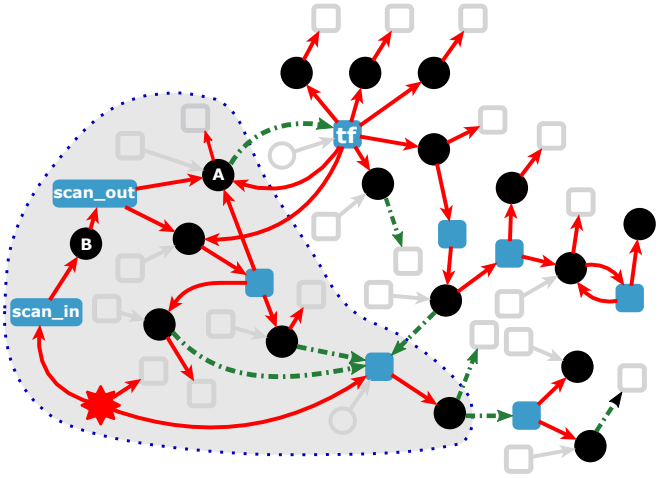


Fig. 2: Part of COB's publish-subscribe dependency graph. Circles represent nodes (software components) and squares represent topics. Edges are labeled Dependent (solid) or Independent (dashed). We assume the node with a star has changed. The traditional IA approach deems solid circles as being affected by the changed. Solid circles within the dotted line area show the impact set produced by the proposed approach, 50% smaller.

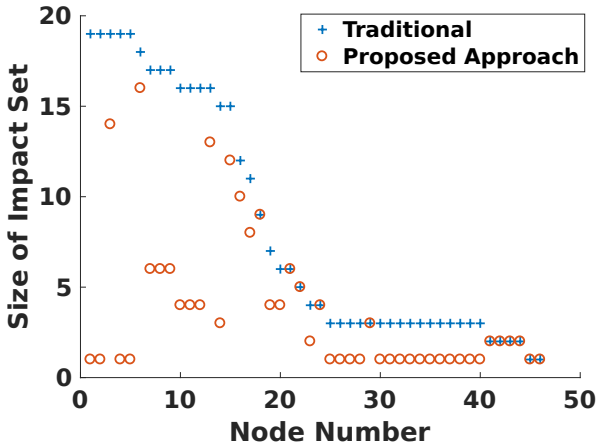


Fig. 3: Impact set reduction across the nodes of COB.

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) [20] and relies heavily on publishers and subscribers to communicate using '*topics*' between nodes [14]. A node can publish or subscribe to multiple topics, and multiple nodes can subscribe or publish to a single topic, providing flexibility in how the nodes of the system are connected, but still enabling modularity as nodes are separated through topics. Many studies have analyzed the publish-subscribe architecture [8] [10] [15] [18] [22]. Eugster et al. [10] presented that the publish subscribe architecture can be decoupled in terms of space, time, and synchronization. Recently, Rusakov et al. [22] presented multiple concurrency patterns for common robotics coordination tasks operating in parallel.

```
1  publishTransform(){
2    <publisher>.publish(<msg>);
3  }
4  foo(){
5    ros::Rate <rate_var>(getPubRate());
6    while (ros::ok()){
7      publishTransform();
8      <rate_var>.sleep();
9    }
10 }
```

Code 1: Component A - Independent Publisher due to fixed rate caused by adaptive sleep.

```
1  callback(...) {
2    <publisher>.publish(<msg>);
3  }
4  foo() {
5    <publisher> = <nh>.advertise("scan_out",...
6    <sub> = <nh>.subscribe("scan_in", 1, &callback...
7  }
```

Code 2: Component B - Dependent Publisher.

Figure 2 presents part of the graph representation of COB's publish-subscribe architecture. The ROS nodes are represented by circles and the topics by squares. COB has 66 nodes, and 155 topics (we only show 18 nodes and 9 topics in the figure). We will assume that the node with a *star* has changed. Nodes and topics external to the abstracted system view are depicted in gray.

This graph representing the publish-subscribed architecture of COB also encodes a conservative approximation of the nodes dependencies. In essence, if there is a path through the edges from one node to another, then the former can impact the later. 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 2), then a traditional impact analysis would traverse this dependency graph starting from the star node and propagating the effect along the edges to all reachable nodes. Using such approach would render all the nodes in the graph in Figure 2 as potentially impacted by the change. The impact set contains *19* affected nodes that a developer will have to check.

When changes are not data-driven but rather rate-driven, like in the case of updating the rate of the laser scanner, such an approach is likely overly conservative. 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 mechanism (like the one we proposed in this paper) to tell that the node marked as *A* in Figure 2 publishes data to topic *tf* on a timer. We can then label such outgoing edge as "rate-Independent" (or just "Independent" – dashed in Figure 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 2, this process results in the reduction of

the impact set of nodes by 50%. In this work we develop this impact analysis process focused on rate.

The benefits of the approach will vary based on the system coupling and the nodes being changed. Figure 3 illustrates the range of benefits for COB. The x-axis represents the 48 nodes in COB (leaf nodes are excluded since changing them does not impact any other node), the y-axis represents the impact set size assuming that the node in 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 will vary based on which node 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 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 a size of more than 10, the reduction is 9.82 nodes.

Crucial to the cost-effectiveness of this approach is the analysis of nodes to identify whether their publishing edges are independent. Through this work we identify a common set of code patterns that are highly likely to render a publisher independent. For a node like *A*, a sample code pattern is shown in Code 1, where the publishing rate is fixed as the semantics of the `<rate_var>.sleep()` (line 8) call enforce a wait period before the next iteration through the loop to publish again. For a dependent node, like *B*, a sample code pattern is shown in Code 2. In this example, the call to `<publisher>.publish(<msg>)` (line 2) occurs within a subscription `callback` function so the publishing rate of this node will depend on the rate of received messages. The identification of these code patterns followed an iterative process, starting with a pool of candidate patterns based on our development experience and recommended practices, followed by several refinement steps 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. Further details about the approach and the implemented tool are provided in the next section.

III. Approach and Implementation

Figure 4 shows the high-level architecture of the proposed approach. It is divided into two phases: Dependency Analysis (DA) and Impact Analysis (IA). DA takes as input the system code and its launch file (a file to configure the system deployment through parameters and node and topic remappings). DA outputs a dependency graph where edges from a publisher to a topic are labeled as either 'dependent on' or '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 publisher is found. IA reports the reachable set of nodes that constitutes the Impact Set for the changed component(s).

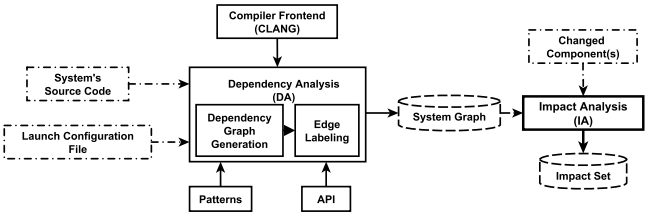


Fig. 4: High-Level Architecture of the proposed approach.

```
1  callback(...){
2    <publisher>.publish(<msg>);
3  }
4  foo(...){
5    <timer_var> = <nh>.createTimer(...,&callback,...
6  }
```

Code 3: Pseudocode from *cob_obstacle_distance_moveit* package exhibiting the Timer based pattern.

A. Dependency Analysis (DA)

The first step of DA is to generate the system 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 subscription used by the analyzed function. The summary is produced by generating and traversing the function control flow graph while searching for function calls in a predefined set that depends on the API and middleware being used. For ROS, this set includes calls like `advertise`, `advertiseCamera`, `subscribe`, `subscribeCamera`, `sendTransform`, `lookupTransform`, `RealTimePublisher` and the object of the predefined type (e.g., `ros::NodeHandle`, `image_transport`, `tf::TransformerListener`, `TransformerBroadcaster`, `message_filter`. Figure 5 presents a sample summary for the code of Node B in Code 2. For subscribers, it contains the *topic name* and the *callback function name*. For publishers, it has the *topic name* and the *variable name*.

To generate the dependency graph of published and subscribed messages, the approach first performs a union of all the summaries 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 on which to run impact analysis. 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 graph, it further examines the source code, analyzing every path leading to a publisher's publish call to identify certain code patterns that render those edges as rate-independent. We now introduce three initial patterns (others are mentioned in Section VII)

```
Summary: Node B
Subscriber: scan_in: callback
Publisher: scan_out: <publisher>
```

Fig. 5: Summary for Source Code 2.

Algorithm 1: Labeling publishers in a node.

```
1  Function LabelPublishers(publisher[])
3      foreach pub ∈ publisher[] do
5          pub.label = Independent
7          foreach path2pub ∈ callGraphSearch(pub) do
9              if not findPatterns(path2pub) then
11                 pub.label = Dependent
13                 break foreach
```

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 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. Code 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, we locate a call to function `createTimer` and then we extract the argument which gives the callback function name which will be invoked at a fixed rate.

3. Adaptive Sleep. Robotic middleware like ROS often provides adaptive sleep functions which take execution time of a cycle into account and sleep for the leftover time of the initialized duration, ensuring that the loop is executed at a fixed rate. In ROS, this can be done by initializing an `ros::Rate` object which specifies the rate at which the loop should be executed. Then, inside a loop, `ros::Rate` object's sleep function is called to sleep until the next execution should start. For example, in Code 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.

In the second step of DA, for each node, for each publisher to a topic, the approach gathers all loop-free paths from each publishing location to each root node (either a callback function or the main function in the node). It then analyzes each path, searching for one of the three defined patterns. If a path conforms to a pattern, then it is labeled as such. If it does not, then that path is deemed as dependent, and consequently the edge is labeled as *Dependent*. In the case that all paths to a publisher are labeled to have the Independent pattern, then the publisher is labeled as *Independent*. This process is succinctly described in Algorithm 1.

Algorithm 2: Impact Analysis. IS_G represents the impact set considering all changes, IS represents the impact set of a changed node.

```
1  Function ImpactAnalysis(changed_components[], G)
3      IS_G = IS = φ
       // Reset outgoing edges of changed component as dependent
4      foreach c ∈ changed_components[] do
6          foreach edge e ∈ getOutgoingEdges(G, c) do
8              e.dependent = True
10     foreach vertex v ∈ G do
12         v.visited = False
14     foreach c ∈ changed_components[] do
16         IS = DFSVisit(c, IS)
18         IS_G = IS_G ∪ IS
20     return IS_G

21 Function DFSVisit(c, IS)
23     c.visited = True
25     foreach v ∈ adjacent[c] do
27         if v.visited is False then
               // Exand impact set only over Dependent edges
28             if edge(c, v).dependent is True then
30                 IS = IS ∪ v
32                 DFSVisit(v, IS)
34     return IS
```

B. Impact Analysis (IA)

As shown in Algorithm 2, IA takes a list of changed components and the system dependency graph as input. Since changed components may impact the publishing rates, we re-label their outgoing edges as dependent (lines 4-8). Next we set all the nodes as not visited yet (lines 10-12) to then initiate a depth first graph traversal rooted at each changed component (lines 14-18). A global impact set, IS_G, contains the union of all impact sets IS of every changed component. IS is computed for each changed node in the function *DFS-VISIT* (line 21-34). Nodes adjacent to a changed node are visited, extending the traversal and impact set over dependent edges (lines 28-32).

C. Implementation

Our approach implementation builds heavily on the source code analysis tool Clang [2], 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 [6] to store and parse the summaries as YAML files, and PUGI XML [12] for parsing and extracting partial information from ROS launch configuration files. Finally, we use Graphviz DOT [9] to generate visual depictions of the dependency graphs to facilite their interpretation and debugging of the tool. Our tool *RSIA* (Rate based Static

Impact Analysis) is available for download from `http://nimbus.unl.edu/tools/`

IV. Study Setup

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).

The analyzed systems are Care-O-Bot [1], PR2 [3], and an autonomous aerial water sampler (H2OS) [17]. PR2 is a mobile manipulation platform developed by Clearpath Robotics. H2OS is a drone-based water sampling solution [17] from our own NIMBUS Lab[1]. Both PR2 and COB systems are open-sourced, and the three systems are written almost entirely in C++ using ROS extensively.

We assess the proposed approach in three phases. First, we evaluate the precision and recall of the tool at generating the dependency graph. To do this, we manually generated a ground truth dependency graph. 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, intermingled with sessions where all authors reviewed code samples and hard-to-determine dependencies. This process resulted in a dependency graph, with edges labeled as dependent or independent, that we deemed to be correct and treated as the ground truth for the study. First part of the study compared this ground truth graph versus the one constructed by the tool. We also break down the evaluation among publishers and subscribers that were detected and named.

Second, to compare the impact sets, we implemented the traditional IA approach by performing a DFS from a changed node on the ground truth dependency graph of each system. We used the same ground truth graph to assess the ideal implementation of our approach. To evaluate the IA portion of the tool, we used the tool's generated graph with 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).

Third, to assess the runtime performance, we measured the duration of the tool implementing the approach and compared it against the time to compile the systems.

We recognize that the study presents several threats that will limit the validity of the results. From an external validity perspective, we only studied three systems using ROS. 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. From an internal perspective, we recognize that analyses involving a manual process are susceptible to bias. We attempted to control that bias by having multiple participants examining 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 graph. Similarly, the code may exhibit other dependencies that we failed to identify either manually or with the provided tool. We provide a link to the analyzed code to enable the reproduction and assessment of the results. We acknowledge that the performance of the approach may not be indicative of what happens in practice as the engineer's familiarity with the code may introduce more variability into the IA process. With these limitations in mind, we proceed to share and analyze the study results.

[1] `nimbus.unl.edu`, but programmed by another researcher

TABLE I: A-Tool Edge Detection for Subscribers

System	Total	Detected and Mapped	Detected	Undetected
PR2	23	20	2	1
COB	40	26	13	1
H2OS	36	32	4	0

TABLE II: A-Tool Edge Detection for Publishers

System	Total	Detected and Mapped	Detected	Undetected
PR2	53	50	1	2
COB	58	45	8	5
H2OS	35	30	5	0

V. Study Results

We present the results in three stages. First, we present the tool's capability to detect system component dependencies. Second, we perform a three-way comparison of the generated impact sets by traditional impact analysis using the ground truth dependency graph (*Trad*) generated manually, the proposed approach using the ground truth dependency graph (*A-GT*) generated manually, and the automated version of the proposed approach as implemented in the tool (*A-Tool*) which generates the dependency graph through code analysis. Third, we compute the overhead of the automated approach.

A. Dependency Graph

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 label assignment. Given this differentiation, we assess them separately. *A-Tool*, the automated approach, has 96% recall, that is, it identifies almost all publish and subscribe edges. However, some edges are identified, but their names are not mapped 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*.

Subscribers. Table I presents the subscription edge detection information. *A-Tool* detected and mapped 87% of the subscribers with their right mappings for PR2. Topic names for two detected subscribers remained unmapped because their names were provided through a launch file variable

TABLE III: Edge Classification for the ground truth instance of the approach (A-GT) by the instance of the approach implemented tool (A-Tool)

(a) PR2 - 51 Published Edges Detected

		A-Tool	
		Independent	Dependent
A-GT	Independent	17	11
	Dependent	0	23

(b) COB - 53 Published Edges Detected

		A-Tool	
		Independent	Dependent
A-GT	Independent	13	1
	Dependent	1	38

(c) H2OS - 35 Published Edges Detected

		A-Tool	
		Independent	Dependent
A-GT	Independent	3	6
	Dependent	0	26

within a data structure not supported by the current tool implementation. *A-Tool* missed a subscriber edge because the tool did not have the relevant API call information to retrieve it. For COB, *A-Tool* detected and mapped 65% of the subscribers. 33% subscriber edges were not mapped as their names were defined in launch files. One edge went undetected. For H2OS, the tool detected all edges and mapped 89% of them. The rest had names defined in launch files.

Publishers. Table II presents the publisher edge detection performance. *A-Tool* detected and mapped 94% publisher edges for PR2. *A-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) undetected topics were caused by included files that the tool failed to reach. For H2OS, *A-Tool* detected all edges and 86% (30) were detected and mapped. The remaining 14.3% of the edges had names defined as part of launch file parameters.

Table III presents confusion matrices for all the analyzed systems, comparing the label assignments for the detected edges of the publishers[2] between *A-Tool* and the ones assigned by *A-GT*. For PR2 (Table IIIa), all 23 dependent labels are recognized as such by *A-Tool*. However, *A-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 accurate in the implementation of the edge marking scheme. Table IIIb presents the label matching results for the COB system. Out of 39 dependent edges, *A-Tool* mismarked one as independent, and out of 14 dependent edges, it marked one as independent. The mismarked 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 IIIc), all 26 dependent labels are recognized correctly. However, *A-Tool* conservatively marked six (6) independent edges as dependent.

[2]Recall that this is only done for publishers as they are the only ones to rate-dependency labels.

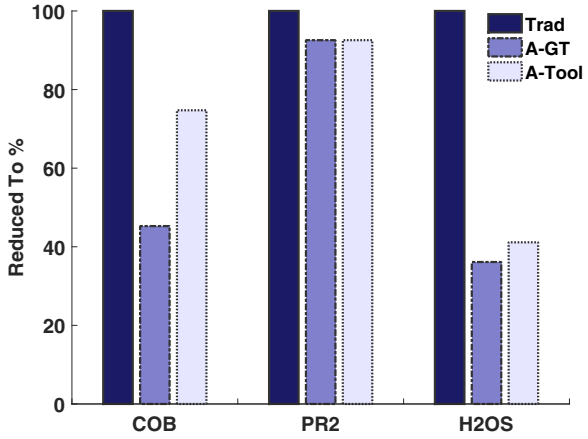


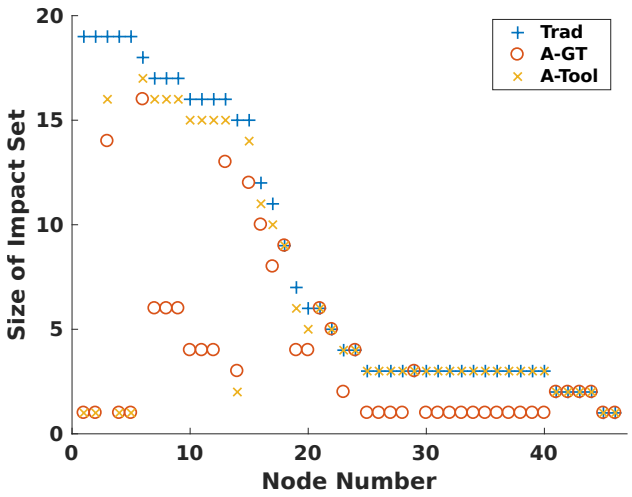
Fig. 6: Impact set reduction ratio of *A-GT* and A-Tool over *Trad*.

B. Impact Analysis Sets

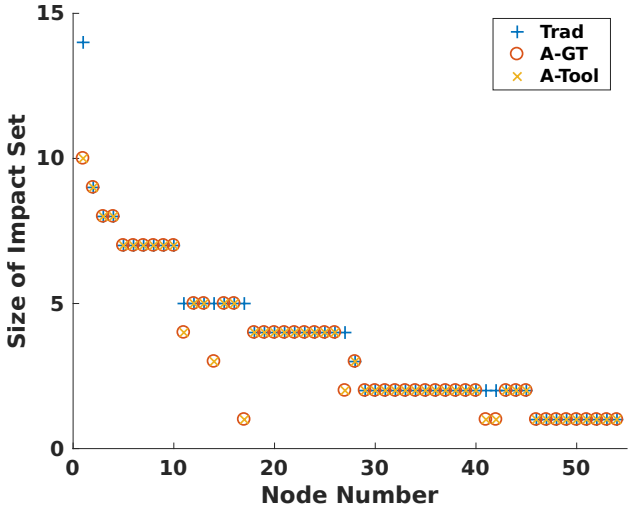
Figure 6 summarizes the impact set reduction for all three systems. In this figure, the size of the impact set returned by *Trad* is the baseline (100%). To compute the data in this graph summary, we executed each approach as many times as nodes in a system, assuming that one distinct node changed each time. We compute the ratio between the accumulated size of the impact sets of *A-GT* and *A-Tool* over *Trad*.

For COB, the impact set was reduced to 45% by *A-GT*, and to 75% by *A-Tool*. The single false-positive in COB (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 *Trad* struggles to provide any reduction as the system data-flow is highly coupled. *A-GT* and *A-Tool* can de-couple some central communication components by defining some edges as independent, reducing the impact to approximately 40% of the *Trad* set.

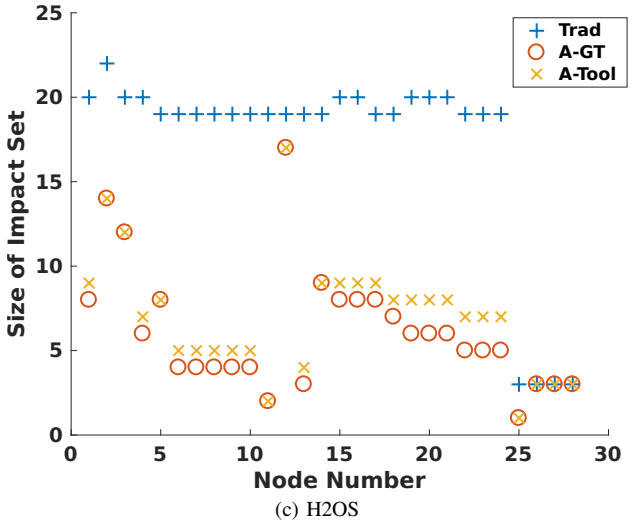
We now look at those results in more detail, checking the range of impact set sizes as different components in a



(a) COB



(b) PR2



(c) H2OS

Fig. 7: Impact set size reduction when applying an approach assuming the component in the x-axis changed.

system change. Figure 7a 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 the decreasing order of their impacted depth in the dependency graph. Leaf nodes in the dependency graph have no impact when changed, so they are not shown. Pluses represent the impact set produced by *Trad*, circles represent the impact set produced by *A-GT*, and the crosses represent the size of the impact set generated by *A-Tool*. Five (5) of COB's 46 nodes, all with large impact sets under *Trad*, are noticeably reduced by the proposed approach. *A-Tool*, however, could not improve on *Trad* when the reported sets had a handful of components. In Figure 7b, for PR2, we note that the reduction achieved by *A-Tool* over *Trad* is limited. *A-GT* 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 7c, for H2OS, illustrates yet a different scenario with major gains in reduction independent of what component was changed. The architecture of H2O is such that most nodes are highly data-coupled but not always rate-coupled, which means that *A-Tool* can provide on average impact sets of less than 8 nodes while *Trad* delivers sets of 19 nodes on average.

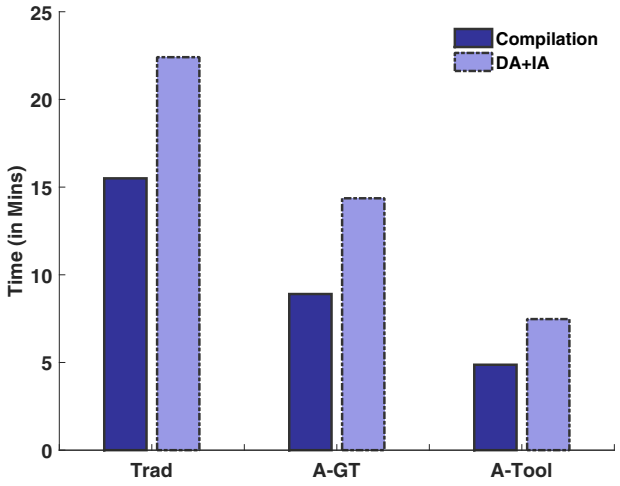


Fig. 8: Overhead Analysis.

C. Overhead Analysis

We measure the tool's runtime performance regarding the analysis overhead when compared against that of compiling the system (without our analysis). Figure 8 shows, for each system, the time to compile the system and to analyze it. The time to analyze a system was computed as the average of the analysis times where each component was assumed to be changed. Out tool took approximately 30% longer than compilation, ranging from three to eight extra minutes when ran on a laptop with Intel i7-2670QM processor (8 cores, 2.20 GHz) running Ubuntu 16.04 with 12 GB RAM.

VI. Related Work

Section I and II provide an overview of the relevance of communication rate in robotic systems. In this section, we discuss work related to impact analysis and provide details on the novelty of the proposed approach.

Arnold and Bohner [5] defined Impact Analysis (IA) as the approach for identifying what to modify to accomplish a change or the potential consequences of a change. In this work, we focused on the latter. Impact Analysis techniques can be classified as static, dynamic, or hybrid. 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 the impact sets by considering every potential input. They can vary in the type and granularity of dependency captured. For example, *Imp* [4] uses static program slicing with impact analysis to analyze larger codebase. While Chianti [21] captures atomic changes in source code and uses the system call graph to report the impact sets. Within the realm of distributed system analysis, others have performed static analysis that can generate alternative and more detail representations of publish and subscribe systems (e.g., [11], [19]). Incorporating such a more detailed representation is part of our future work.

Dynamic impact analysis techniques rely on the execution of the code, rendering results that depend on particular inputs used to drive the execution. They typically consume execution trace data, where trace can be, for example, executed functions [13]. Given such traces, these techniques analyze the temporal relations of the elements in the trace (e.g., always before, always after) to derive their potential dependencies. In the context of distributed systems, Cai and Thain [7] recently introduced a dynamic IA targeting communication channels. Our work is different in that we focus on publish and subscribe constructs, and in particular their rates. From the perspective of the proposed approach, we recognize the potential of dynamic impact analysis to help us observe the publish and subscribe channels linked to launch configuration files. Incorporating such information into our tool would result in a hybrid impact analysis approach, conceptually similar to others like SD-Impala [16] but still unique in its focus on the rates of publish and subscribe channels.

VII. Conclusion & Future Work

We presented an approach 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. The approach is more precise than existing impact analysis approaches. We have shown its potential through a manual examination and an automated tool for ROS. In the three case studies, the tool reduced the impact set that developers must process by up to 41% of alternative approaches.

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 the 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.

References

[1] Care-o-bot robot. http://www.ros.org/wiki/Robots/Care-O-bot.

[2] "clang": a C language family frontend for LLVM. http://clang.llvm.org/.

[3] Pr2 robot. http://www.ros.org/wiki/Robots/PR2.

[4] M. Acharya and B. Robinson. Practical Change Impact Analysis Based on Static Program Slicing for Industrial Software Systems. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 746–755.

[5] R. Arnold and S. Bohner. Impact analysis-Towards a framework for comparison. In *, Conference on Software Maintenance, Proceedings*, pages 292–301, Sept. 1993.

[6] J. Beder. yaml-cpp, a yaml parser and emitter for c++. https://github.com/jbeder/yaml-cpp.

[7] H. Cai and D. Thain. Distia: A cost-effective dynamic impact analysis for distributed programs. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 344–355, 2016.

[8] D. T. Coleman, I. A. Sucan, S. Chitta, and N. Correll. Reducing the barrier to entry of complex robotic software: a moveit! case study. *Journal of Software Engineering for Robotics*, 5(1):3–16, 2014.

[9] J. Ellson, E. Gansner, L. Koutsofios, S. C. North, and G. Woodhull. Graphviz—open source graph drawing tools. In *International Symposium on Graph Drawing*, pages 483–484. Springer, 2001.

[10] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 2003.

[11] J. Garcia, D. Popescu, G. Safi, W. G. Halfond, and N. Medvidovic. Identifying message flow in distributed event-based systems. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, pages 367–377, 2013.

[12] A. Kapoulkine. pugixml: Light-weight, simple and fast xml parser for c++ with xpath support. https://github.com/zeux/pugixml.

[13] J. Law and G. Rothermel. Incremental dynamic impact analysis for evolving software systems. In *14th International Symposium on Software Reliability Engineering*, pages 430–441, Nov. 2003.

[14] E. A. Lee and S. A. Seshia. Introduction to embedded systems: A cyber-physical systems approach. http://LeeSeshia.org, 2015. ISBN: 978-1-312-42740-2.

[15] I. Lütkebohle, R. Philippsen, V. Pradeep, E. Marder-Eppstein, and S. Wachsmuth. Generic middleware support for coordinating robot software components: The task-state-pattern. *Journal of Software Engineering for Robotics*, 2(1):20–39, 2011.

[16] M. C. O. Maia, R. A. Bittencourt, J. C. A. d. Figueiredo, and D. D. S. Guerrero. The Hybrid Technique for Object-Oriented Software Change Impact Analysis. In *14th European Conference on Software Maintenance and Reengineering*, pages 252–255, Mar. 2010.

[17] J.-P. Ore, S. Elbaum, A. Burgin, and C. Detweiler. Autonomous aerial water sampling. *Journal of Field Robotics*, 32(8):1095–1113, 2015.

[18] A. Paikan, U. Pattacini, D. Domenichelli, M. Randazzo, G. Metta, and L. Natale. A best-effort approach for run-time channel prioritization in real-time robotic application. In *Intelligent Robots and Systems (IROS)*, pages 1799–1805, Sept 2015.

[19] R. Purandare, J. Darsie, S. Elbaum, and M. Dwyer. Extracting conditional component dependence for distributed robotic systems. In *Intelligent Robots and Systems (IROS)*, pages 1533–1540, Oct 2012.

[20] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5, 2009.

[21] X. Ren, B. G. Ryder, M. Stoerzer, and F. Tip. Chianti: A Change Impact Analysis Tool for Java Programs. In *Proceedings of the 27th International Conference on Software Engineering*, pages 664–665, 2005.

[22] A. Rusakov, J. Shin, and B. Meyer. Concurrency patterns for easier robotic coordination. In *Intelligent Robots and Systems (IROS)*, pages 3500–3505, Sept 2015.