Integration testing criteria for mobile robotic systems

Maria A. S. Brito*, Marcos P. Santos*, Paulo S. L. de Souza* and Simone do R. S. de Souza*
*Department of Computer Systems
University of São Paulo, São Carlos, SP, Brazil
Email: masbrit,mpereira,pssouza,srocio@icmc.usp.br

***Abstract*—Testing activity applied to mobile robotic systems is a challenge because new features, such as non-determinism of inputs, communication among components and time constraints must be considered. Simulation has been used to support the development and validation of these systems. Coverage testing criteria can contribute to this scenario adding mechanisms for measuring quality during the development of systems. This paper presents a test model and a set of coverage criteria to test the interaction among the components of mobile robotic systems. The model and criteria focus on robotic systems developed in ROS, a Robotic Operational System in which communication is established through publish/subscribe interaction schema. The testing criteria were evaluated using a robotic application. The results confirm that the use of coverage testing criteria has advantages for integration testing of mobile robotic systems.**

I. Introduction

Research on software methods and environments of robotics development has increased over the past decade. The development of robotic systems requires a middleware to provide tools for interfacing with different system modules, hardware abstraction and communication facilities. Many robotic systems are custom-designed for specific projects and involve high costs of software and hardware. Some middlewares used to develop applications for robotics and complex systems exhibit different behaviors and specificities, as well as unique qualities that make them better suited to a particular task [9]. Various robot middlewares have been proposed, such as [11] Player [10], Orocos [5], Orca [4] and ROS [18].

The validation of a mobile robotic application normally includes simulations and software testing techniques. VSTs (Virtual Simulation Tools/Technology) have been widely used in development environments to model, simulate and evaluate different aspects of a system before they are implemented in an objective platform [9]. They reduce the development time because all tasks are grouped into the same environment. The validation of these systems is a challenging task due to the amount of human resources, equipment and technical support required for the production of reliable results and safety assurance. Some tasks may be highly risky and uncertain, especially those that involve transferring results from an offline simulation to the real world [6].

Mobile robots are normally distributed computing systems because they involve many heterogeneous components, multiple computers and devices. A common communication mechanism of such systems is the publish/subscribe interaction schema, which provides a loosely coupled form of interaction in large scale settings and comprises subscribe and publish nodes. Subscribers express their interest in an event, or a pattern of events, and are notified of any event generated by a publisher that has matched their registered interest [8]. Multiple concurrent publishers and subscribers may be connected for a single topic and one node may publish and/or subscribe to multiple topics. Nodes are processes that perform computation. These common features of the publish/subscribe (e.g., event correlation, communication channels, timestamp aspects) require more specific testing approaches, which can show defects in such critical applications. Integration testing activity is an emerging and promising research direction, but it still lacks testing criteria to reveal faults in this domain.

In this direction Kang et al. [13] proposed a simulation-based interface testing automation tool (SITAT) for robot software components which generates and executes test cases in a simulation environment. A similar approach is presented in [12], which proposes a required and provided interface specification-based testing method (RPIST). Both approaches require the specification of the application interfaces. Lim et al. [15] defined a hierarchical testing model and its testing automation framework for robot technology component (RTC). Integration testing is based on the interoperability of hardware and software components. A built-in unit/integration test framework called ROStest was defined to support software testing of the ROS-based systems [2]. ROStest is an integration test suite compatible with xUnit frameworks. Its main disadvantages are the need of writing test code and changes in the source code require changes in the test code.

In concurrent programming, several processes of an application communicate among themselves to solve a problem. A process is a program in execution formed by an executable program, its data, a program counter, other registers and all information required for its execution [22]. This communication feature of the concurrent program resembles the behavior of a mobile robotic application in which several nodes interact to solve a computation. Some approaches of testing integration on concurrent programs explore communication and synchronization among processes [3], [7], [20], [21]. We have revisited these approaches to check the similarities in the testing applied and related to the mobile robotic applications that use the publish/subscribe interaction and, therefore the approach proposed here is inspired on the testing criteria for concurrent programs [19].

This paper proposes an approach of integration software testing for robotic systems to improve the quality of these systems. The robotic systems considered are composed of a set of distributed nodes that communicates by message passing using publish/subscribe schema. Therefore, our approach is

DOI reference number: 10.18293/SEKE2015-191

concerned with the communication among software components implemented in ROS system.

The paper is organized as follows: Section II introduces concepts of publish/subscribe schema and integration testing; Section III provides a motivating example for the problem characterization; Section IV describes our testing approach and the proposed testing criteria in detail; Section V illustrates the application of the approach in a case study; Section VI presents the related work and, finally, Section VII draws conclusions and recommends future work.

II. Background

This section provides some concepts of the publish/subscribe interaction schema and the integration testing addressed.

A. Publish/Subscribe interaction schema

The publisher/subscribe interaction schema studied is implemented in ROS (Robotic Operational System) [18]. ROS is a meta-operating system that consists of an open-source library and tools provided by Willow Garage for the development of robotic applications [18]. It is integrated with tools and libraries, such as OpenCV, a library of programming functions for computer vision of real time, Point Cloud Library (or PCL) for point cloud processing, Gazebo, a multi-robot simulator for outdoor environments, and Player, a network server for the robot control that provides a clean and simple interface to the robots sensors and actuators over the IP network.

A distributed system developed in ROS is formed by many nodes that either run in a single machine or are distributed over different machines. The communication among nodes can be established through two basic mechanisms (Figure 1). The first uses services (synchronous communication) that enable nodes to send or receive requests to/from another node. The second is the publisher/subscribe interaction (asynchronous), in which a node can publish messages in a topic, as well as subscribe from a topic to receive messages from other nodes. Our test model focuses on the publish/subscribe interaction [8].

A node is an executable file that uses ROS to communicate with other nodes by sending messages. A message is a data structure that comprises typed fields. A node sends out a message by publishing it to a given topic. The topic is a name used for the identification of the content of a message. If a subscribe node expresses its interest in an event, or a pattern of events, it is notified of any event generated by a publisher that has matched its registered interest. An event is asynchronously propagated to all subscribers that have registered interest in it.

The publish/subscribe interaction provides the loosely coupled interaction required in such large scale settings. This loosely coupling occurs because of three features: referential decoupling, flow decoupling and time decoupling [8]. When two nodes communicate, they need to know only the type of data they will produce or consume; no extra information is required. This feature is called referential decoupling. Flow decoupling occurs when two processes do not block each other when a message has been sent. Time decoupling enables the transmission or reception of a message at any time. Section III provides an example to illustrate a system described in ROS.

http://www.willowgarage.com/

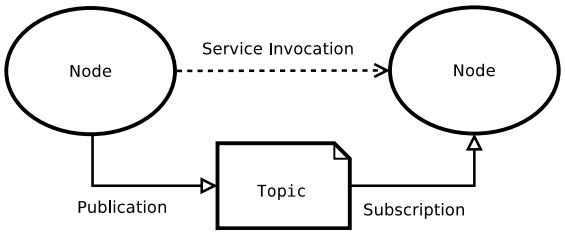

Fig. 1. ROS Communication Model [1].

B. Integration Testing

Different connotations are used for the test phases. We consider the test of the communication among methods of different components (one or more classes) that compose a system an integration testing.

Integration testing should be conducted after the unit testing and the most used test case designs explore the inputs and outputs of data, despite the techniques that exercise specific paths of the source code of the program [17]. This design form is more common because the objective is to exercise the interactions and not the features of units. The integration testing focuses on definitions and uses of variables in different units responsible for the communication. Three types of integration errors can be found [14]: 1) Interpretation error, which occurs when a unit has implemented a functionality different from the specification. Examples include wrong, extra or missing functions; 2) Miscoded call error, which occurs when the developer has inserted a call instruction in the wrong point of the program. Examples are a call instruction on a path or a statement which should not have the call; and 3) Interface error, which occurs whenever the interface standard between two modules has been violated. Examples include incorrect parameters, data types, format and input/output modes. This work explores mainly the miscoded call and interface errors.

III. Motivating Example

This section addresses the identification of an error that may occur in the commnication among components. Figure 2 shows an example of a layout of a robotic application developed using ROS. This application is a simplified version of iRobot Roomba, which explores unknown environments to clean floors. It contains five components, called processes in this paper, namely *Controller*, *Proximity Sensor*, *Collision Avoidance*, *Motor Driver* and *Mapping*, illustrated in the graph generated from the ROS. The communication among these processes is established using the following topics: *odometry*, *proximity*, *cmd_motors*, *velocity*, *location*, *time_To_Impact* and *mapper_activation*.

Listings 1 and 2 show two excerpts of the codes related to the *Collision Avoidance* [16] and *Controller* processes. The *Controller* process receives information from *time_to_impact Callback* topic (line 11) published by the *Collision Avoidance* process and processes it. In the next step, it publishes data in the *cmd_motors* topic (line 29), which will be subscribed by the *Motor Driver* process. The *Controller* process also publishes the robot speed in the *velocity* topic (line 28), which

http://www.irobot.com/Company

will be subscribed by the *Collision Avoidance* process (line 8). Each code has two *callback* functions that contain a code related to the incoming messages. ROS will call the *callback* function once for each arriving message. For example, if the *Controller* process has published data in the *velocity* topic, the *callback* function, called *velocityCallback*, is activated (line 8). The *Collision Avoidance* process will be notified that a process has published data of its interest, because it has subscribed in the *velocity* topic. The *time_to_impactCallback* function is invoked when data have been published in the *time_To_Impact* topic (line 30).

```
1  //Collision Avoidance Process
2  State State; // {working, slowed, stopped,
       crashed}
3  int proximity;
4
5  proximityCallback(int proximityP) {
6          proximity = proximityP;
7
8  velocityCallback(int speed) {
9          if (speed == 0)
10                 time_to_impact = 9999;
11         else
12                 time_to_impact = proximity /
                       speed;
13         if (time_to_impact < 2) {
14                 if (state != working)
15                         state = stopped;
16                 else {
17                         state = crashed;
18                         assert(false); //
                               Failure
19                 }
20         }
21         else
22                 if (time_to_impact < 3) {
23                         state = slowed;
24                         reduceMapping();
25                 }
26                 else {
27                         state = working;
28                         activeMapping();
29                 }
30         publish(time_to_impact);
31 }
```

Listing 1. Excerpt of the *Collision Avoidance* [16] process.

```
1  //Controller Process
2  State state; //{working, slowed, stopped,
       crashed}
3  int time_to_impact;
4  int speed; //{0, 1, 2}
5  int c_motors; //{stop, reduce, working}
6
7  odometryCallback(int odometryX) {
8      odometry = odometryX;
9  }
10
11 time_to_impactCallback(int timeToImpactT) {
12     time_to_impact = timeToImpactT;
13     if (time_to_impact < 2) {
14         c_motors = stop;
15         speed = 0; //break
16         state = crashed;
17         activeMapping();
18     } else if (time_to_impact <= 3) {
19         c_motors = reduce;
20         speed = 1;
21         reduceMapping();
22         state = slowed;
23     } else {
24         c_motors = working;
25         speed = 2;
26         state = working;
27     }
28     publish(speed);
29     publish(c_motors);}
```

Listing 2. Excerpt of the *Controller* Process

Communication problems may occur when the interaction among processes is intense. An example of a defect that can be identified is the different frequency of publication from two processes communicating. Line 30 specifies the publication of the data in the *timeToImpact* topic. If, for any reason, the data have been published faster than the *Collision Avoidance* process can access, some data can be lost. If that happened, after their processing the *Collision Avoidance* process would produce incorrect outputs and the system would exhibit a non-expected behavior.

IV. Integration Testing Approach

Our integration testing approach aims at revealing defects related to the communication among processes of a robotic application. The defined testing criteria exercise the input and output data of the processes, frequency of publishing of the messages and failures of an application. The testing approach uses the source code of a program as input to derive the elements required for each criterion based on a graph. The next step is to create the inputs for the application or the sets of test cases able to cover the required elements. The application is executed and the coverage is analysed for each criterion. New test data can be generated to improve the coverage of the elements required until the maximum coverage has been obtained.

A test model was defined to capture the communication interfaces and data flow among processes. This model is based on the work of Souza et al. [19], [21], who defined the structural testing criteria for MPI (Message Passing Interface) exploring interactions among processes of a concurrent application. Our test model extends this work analyzing specific details of publish/subscribe communication as the loosely coupled among processes, and non-determinism during the processing of the *callbacks* (from threads and queues).

The proposed test model represent the application using a composition of two types of graphs, the graph generated by the ROS and Control Flow Graphs (*CFGs*). A fixed number of processes np is created at the beginning of the application. In our model, p is a process that performs computations and communicates with another process using streaming topics, RPC services, and the parameter server of the ROS. Set of processes P and inter-processes edges T are represented in the Publish/Subscribe-based Def-Use graph (*PSDU*). Each process p of the *PSDU* graph has its own internal structure which is represented by a *CFG*.

A *CFG* is a directed graph that represents the structure of a program as nodes and edges. Nodes in the *CFG* represent

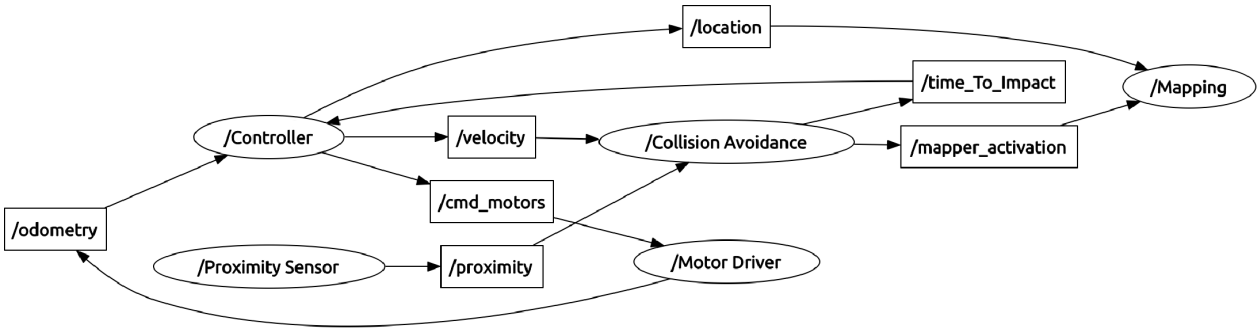

Fig. 2. *ROS* Graph of the iRobot System.

blocks of sequential statements such as if any one statement of the block is executed, then all statements in the block are executed. The edges represent the communication among the nodes of a *CFG*.

A definition (*def*) is a location in the program where a value for a variable is stored into memory. A use occurs in a location where a value of variable is accessed. In this model a use can occur of three forms: 1) computation use (*c-use*), when the variable is used in a computation; 2) predicative use (*p-use*), which occurs when the variable is used in a decision; and 3) communication use (*m-use*), which occurs in a inter-processes edge. A def-clear path for a variable x through the *CFG* is a sequence of nodes, n_1, n_2, .. ,n_n that do not contain a definition of x. The nodes into of *CFG* in this model also represent definitions and uses of variables.

Figure 2 shows an *ROS* graph for the example presented in the Section III. Each node represents a process and each edge represents a message between two processes. Figure 3 illustrates an *PSDU* graph, which is a part of the code of the Listings 1 and 2 with three processes exchanging messages: p_1, p_2 e p_3. Each *CFG* represents a *callback* method and inter-processes communication edges link each graph. For example, in nodes n_2, n_4 and n_5 of the process p_1 data are defined and node n_6 they are published. When this occurs the *publish* method puts data in an output queue of the ROS and return. From this point the middleware allocates threads that delivery this message to the subscribes. For this example, the methods m_1 of process p_2 and the m_1 of process p_3 subscribe messages from m_1 of the process p_1. The method m_1 of process p_1 subscribes messages from m_1 of p_3.

Based on this model, the criteria for the publish/subscribe applications are:

- all-nodes-communication criterion: each process of the *PSDU* graph will be exercised at least once. The input and output interfaces of each process should be exercised by the test case set.
- all-nodes-publish criterion: all processes of the *PSDU* graph that have published messages will be exercised at least once. The interfaces provided will be exercised at least once by the test case set.
- all-nodes-subscribe criterion: all processes of the *PSDU* graph that have subscribed messages will be exercised at least once. The interfaces for the topics that have received data will be exercised at least once by the test case set.
- all-pairs-publish/subscribe criterion: each pair of the *PSDU* graph that consists of a process that publishs data in a topic and another process that has subscribed messages from the same topic must be exercised at least once by the test case set.
- all-multiples-publish/subscribe criterion: more than two processes of the *PSDU* graph must be exercised at least once by the test case set. This criterion exercises the composition of processes. Initially, three processes are tested; next, the number of processes is increased until the maximum number of processes of the graph has been reached.
- all-m-uses criterion: each *m-use* association of the *PSDU* graph will be exercised from the last definition of the variable in a node n_i until the first use of the variable in the subscribe process ps_j, i.e., for each node n_i and each $x \in def(n_i)$, the test set must exercises a path that covers an inter-processes association w.r.t. x and $n_i \in pp_k$.
- all-sequences criterion: different input sequences are exercised for each subscribe process of the *PSDU* graph from different origins (topics). The criterion exercises the order in which asynchronous events are received.

V. Example of an Application

This example is a simplified application of iRobot Roomba presented in Section III. For the application of the testing criteria, the first step is the generation of the elements required for each criterion (Table I) based on the *PSDU* graph. Table I shows some required elements. The second step is the generation of the test cases for covering the required elements. A test input for this application is data from the *Proximity Sensor* process and the expected output is a command from the *Motor Driver* process for the actuators of the robot (wheels). One test set is able to cover the required elements was generated based on the testing criteria.

Defects were inserted into the programs for the evaluation of the testing criteria in revealing faults. These defects focus mainly on the communication among the application processes. Three types of defects were inserted: 1) changes in

TABLE I. SOME REQUIRED ELEMENTS FOR THE EXAMPLE IROBOT.

Criteria	Required Elements
All-nodes-communication	$(n_6^{1,1})$, $(n_{11}^{2,1})$, $(n_8^{3,1})$, $(n_1^{1,1})$, $(n_1^{2,1})$, $(n_1^{3,1})$, ...
All-nodes-publish	$(n_6^{1,1})$, $(n_{11}^{2,1})$, $(n_8^{3,1})$, ...
All-nodes-subscribe	$(n_1^{1,1})$, $(n_1^{2,1})$, $(n_1^{3,1})$, ...
All-pairs-publish/subscribe	$((n_6^{1,1}), (n_1^{2,1}))$, $((n_{11}^{2,1}), (n_1^{1,1}))$, $((n_6^{1,1}), (n_1^{3,1}))$, $((n_8^{3,1}), (n_1^{1,2}))$, ...)
All-multiples-publish/subscribe	$((n_{11}^{2,1}), (n_1^{1,1}), (n_6^{1,1}), (n_1^{3,1}))$, $((n_6^{1,1}), (n_1^{2,1}), (n_{11}^{2,1}), (n_1^{4,1}))$, ...
All-m-uses	$(n_2^{1,1}, n_6^{1,1}, n_1^{3,1}, c\_motors)$, $(n_1^{3,1}, n_8^{3,1}, n_1^{1,2}, odometry)$, ...
All-sequences	(p_1, t_1, t_2), (p_1, t_2, t_1), (p_4, t_2, t_1), ...

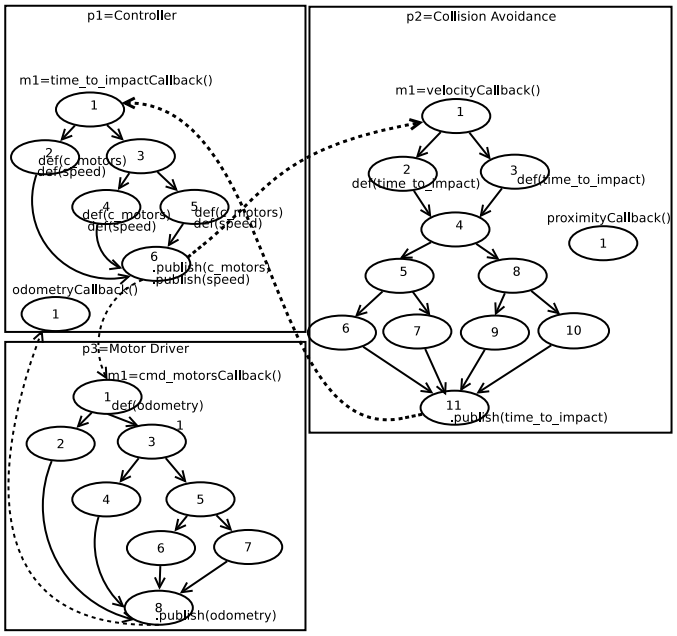

Fig. 3. Example using *PSDU*.

the frequency of publication of the processes, which occurs when a process sends data faster than the capacity of the receptor to take them, or otherwise; 2) non-publication of expected data by the processes, when an expected datum is not sent from a process; and 3) changes in the queue size of a topic that received data from different process, occurs when the queue size is smaller than necessary regarding the frequency of publication in a topic from different processes. Data overwritten in the queues of the topics might result in problems for the robot.

Three programs were generated from each defect, therefore, 9 programs were employed for the evaluation of the testing criteria. Table III shows the results of the execution of test cases in the programs. The first column refers to the identifiers of the programs with defects; the second column shows the results of the application for the test set. Y indicates an expected output, therefore, the test case did not reveal the defect and N indicates an unexpected output, meaning the test case could reveal the defect in the program.

The testing criteria were manually applied and no drivers or stubs were used. According to Table III, only a defect was not revealed by the test cases (Program 2c), because the inserted defect did not change the expected output. All other defects were revealed by test cases. The test cases achieved 100% coverage for the elements required. The advantage of the testing criteria proposed is they support the tester in the control of the integration testing activity. The tester choise the better strategy based on this purpose, such as to test only publish process can be used the *all-nodes-publish* criterion or if he need test integration of various processes he can use the *All-multiples-publish/subscribe* criterion. The focus of the criteria is not on the generation of the test cases, but on the support of the systematization of the activity, when the testing criteria can help in the selection of the parts of the code or specific combinations of processes to be tested during the integration testing.

TABLE II. TEST CASES.

Input	Expected outputs
-1	stop
1	stop
2	reduce
3	working
100	working

The results from manual application of the criteria shown the testing activity of various processes of a mobile robotic system can be supported by the use of integration testing criteria. The program was executed more than once to cover all required elements of *all-sequences* criterion for example. A testing tool which instruments the code, generates and integrates the *CFGs* for all methods can help in this task. Other specific features of *publish/subscribe* schema need more atention as race conditions, deadlocks and concurrent threads in the processing of *callbacks* will be explored in next studies with support of a testing tool.

VI. CONCLUSIONS AND FUTURE WORK

This manuscript has addressed aspects of testing mobile robotic systems and emphasized the publish/subscribe interaction schema. Specific characteristics of the publish/subscribe systems, such as non-determinism, restrictions of time and synchronization of the process have not been totally covered by the existing testing approaches.

TABLE III. RESULTS OF THE EXECUTIONS ON THE PROGRAMS WITH DEFECTS

Identifier	Test set				
	p = -1	p = 1	p = 2	p = 3	p = 100
1a	Y	Y	N	N	N
1b	Y	Y	Y	Y	N
1c	Y	Y	Y	N	N
2a	Y	Y	N	N	N
2b	Y	Y	N	N	N
2c	Y	Y	Y	Y	Y
3a	Y	Y	N	N	N
3b	Y	Y	N	N	N
3c	Y	Y	N	N	N

We have proposed a family of integration testing criteria to publish/subscribe systems. Seven testing criteria were defined for a systematic exploration of communication among components in a mobile robot. The application is represented by an *PSDU* graph generated from the ROS meta-operating system and *CFGs* for the identification of the elements required for these criteria. The defects identified would not usually be revealed with the use of simulations only or unit testing because the focus of our model is on exploring the internal characteristics of each process and its influences in the publish/subscribe schema. An example shown the ability of our approach to support the integration testing activity.

We intend to use more complex systems as case studies to test concurrenct aspects when more intense computation is involved. In addition comparing our testing criteria with other testing approaches (simulation using VSTs, for instance). We are currently developing an coverage analysis tool to support the integration testing criteria.

ACKNOWLEDGMENT

The authors acknowledge the Brazilian funding agencies FAPESP, under processes 2013/03459-4 and 2013/01818-7 and CAPES, under process DS-8435201/M, for the financial support provided for this research.

REFERENCES

[1] Ros/concepts. http://wiki.ros.org/ROS/Concepts. [Acessed 18/10/2014].

[2] A. G. Araújo. ROSint - integration of a mobile robot in ROS architecture. Master's thesis, University of Coimbra, Coimbra, Portugal, 2012.

[3] Y. Ben-Asher, Y. Eytani, E. Farchi, and S. Ur. Producing scheduling that causes concurrent programs to fail. In *Workshop on Parallel and Distributed Systems: Testing and Debugging*, pages 37–39, 2006.

[4] A. Brooks, T. Kaupp, A. Makarenko, S. Williams, and A. Oreback. Orca: A component model and repository. In *Software Engineering for Experimental Robotics*, volume 30, pages 231–251. 2007.

[5] H. Bruyninckx. Open robot control software: the OROCOS project. In *Int. Conference on Robotics and Automation*, volume 3, pages 2523–2528, 2001.

[6] Q. Chen, L. Wang, Z. Yang, and S. Stoller. HAVE: Detecting atomicity violations via integrated dynamic and static analysis. In *Fundamental Approaches to Software Engineering*, volume 5503, pages 425–439. 2009.

[7] Z. Chen, X. Li, J. Y. Chen, H. Zhong, and F. Qin. SyncChecker: Detecting synchronization errors between MPI applications and libraries. In *Parallel Distributed Processing Symposium (IPDPS)*, pages 342–353, May 2012.

[8] P. Eugster, P. A. Felber, R. Guerraoui, and A. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003.

[9] L. C. Fernandes, J. R. Souza, G. Pessim, P. Y. Shinzato, D. O. Sales, V. Grassi Jr., K. R. L. J. Branco, F. S. Osorio, and D. F. Wolf. CaRINA intelligent robotic car: Architectural design and implementations. *Journal of Systems Architecture*, 2013.

[10] B. P. Gerkey, R. T. Vaughan, K. Stoy, A. Howard, G. Sukhatme, and M. J. Mataric. Most valuable player: a robot device server for distributed control. In *Int. Conference on Intelligent Robots and Systems*, volume 3, pages 1226–1231 vol.3, 2001.

[11] M. Y. Jung, A. Deguet, and P. Kazanzides. A component-based architecture for flexible integration of robotic systems. In *Int. Conf. on Intelligent Robots and Systems*, pages 6107–6112, Oct 2010.

[12] J. S. Kang and H. S. Park. RPIST: Required and provided interface specification-based test case generation and execution methodology for robot software component. In *Int. Conference on Ubiquitous Robots and Ambient Intelligence*, pages 647–651, 2011.

[13] S. S. Kang, S. W. Maeng, S. W. Kim, and H. S. Park. SITAT: Simulation-based interface testing automation tool for robot software component. In *Int. Conference on Control Automation and Systems*, pages 1781–1784, Oct 2010.

[14] H. K. N. Leung and L. White. A study of integration testing and software regression at the integration level. In *Conference on Software Maintenance*, pages 290–301, 1990.

[15] J. H. Lim, S. H. Song, T. Y. Kuc, H. S. Park, and H. S. Kim. A hierarchical test model and automated test framework for RTC. In *Int. Conf. on Future Generation Information Technology*, pages 198–207, 2009.

[16] C. Lucas, S. Elbaum, and D. S. Rosenblum. Detecting problematic message sequences and frequencies in distributed systems. In *Int. Conf. on Object Oriented Programming Systems Languages and Applications*, pages 915–926, New York, NY, USA, 2012.

[17] R. S. Pressman. *Software Engineering: Practitioner's Approach*. McGraw-Hill, 6 edition, 2005.

[18] 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*, 2009.

[19] P. S. L. Souza, S. R. S. Souza, and E. Zaluska. Structural testing for message-passing concurrent programs: an extended test model. *Concurrency and Computation: Practice and Experience*, 26(1):21–50, 2014.

[20] S. R. S. Souza, P. S. L. Souza, M. C. C. Machado, M. S. Camillo, A. S. Simão, and E. Zaluska. Using coverage and reachability testing to improve concurrent program testing quality. In *Int. Conf. on Software Engineering and Knowledge Engineering*, pages 207–212, Eden Roc Renaissance Miami Beach, USA, Jul 2011.

[21] S. R. S. Souza, S. R. Vergilio, P. S. L. Souza, A. S. Simão, and A. C. Hausen. Structural testing criteria for message-passing parallel programs. *Concurrency and Computation: Practice and Experience*, 20(16):1893–1916, 2008.

[22] A. S. Tanenbaum. *Modern operating systems*. Second edition, 2001.