Towards Automated Online Diagnosis of Robot Navigation Software

Alexander Kleiner[1], Gerald Steinbauer[2], and Franz Wotawa[2] *

[1] Institut für Informatik, Albert-Ludwigs-Universität Freiburg
Georges-Köhler-Allee, D-79110 Freiburg, Germany
kleiner@informatik.uni-freiburg.de
[2] Institute for Software Technology, Graz University of Technology
Inffeldgasse 16b/II, A-8010, Austria
{steinbauer,wotawa}@ist.tugraz.at

Abstract. Control software of autonomous mobile robots comprises a number of software modules that typically interact in a very complex way. Their proper interaction and the robustness of each single module strongly influences the safety during navigation in the field. Particularly in unstructured environments, unforeseen situations are likely to occur, causing erroneous behaviors of the robot. The proper handling of such situations requires an understanding of cause and effect within the complex interactions of the system.
In this paper we present an approach which is able to automatically derive a model of the communication behavior within a component-orientated control software. The model can be used for online diagnosis in order to increase system robustness during runtime. We demonstrate model learning and system diagnosis on three different robot systems which were controlled by software modules communicating based on the widely used IPC (Inter Process Communication) standard. The demonstrated learning and diagnosis was carried out without any a priori knowledge about the systems.

1 Introduction

Control software of autonomous mobile robots comprises a number of software modules which interact in a very complex manner. Because of this complexity and other reasons like bad design and implementation there is always the possibility of failure during runtime. Such failures can have different characteristics like crashes of modules, deadlocks or wrong data leading to a hazardous decisions of the robot. In order to have truly autonomous robots operating for a long time without or with limited possibility of human intervention, e.g., planetary rovers exploring Mars, rescue robots searching for victims in unknown terrain, robots have to detect, localize, and to recover from failures.

In [1, 2] the authors presented a MBR (model-based reasoning) framework for the control software of autonomous robots using the consistency-based diagnosis techniques of Reiter [3]. Models were created manually by analyzing the structure

* The authors are listed in alphabetical order.

of the software and its communication behavior during runtime. However, for large or partially unknown systems, manual modeling turns out to be suboptimal. Therefore, it is desirable to automatically create system models, either from a formal specification or from observations.

In this paper we present an extension of previous work that allows to automatically derive a model of the structure and the communication behavior within a component-orientated control software. The idea is to use the different communication behaviors between the modules of the control software in order to monitor the status of the system and to detect and localize faults. The algorithm generates a communication graph, showing all modules as vertices and their interactions as edges. Each edge is defined by a particular type of message, e.g., reading of a laser range finder, or a position computed by the localization module, and the condition under which the message occurs, e.g, triggered by inputs, sporadically, or periodically with a specific frequency. From this graph structure, a set of logical clauses is extracted based a component-based modeling schema [4]. Please refer to [1, 2] for more details. Furthermore, for each edge an *observer* is generated that is parameterized according to the learned communication behavior of the link. During runtime, observers are continuously monitoring communication between the modules. If they observe abnormal communication, the diagnosis engine is automatically triggered for determining the reason of failure.

The model learning approach was tested with the control software of the *Lurker* robots [5] used in the RoboCup Rescue league, a multi-robot team of *Zerg* robots [6] used in the RoboCup Rescue simulation league, and the Telemax robot designed for the TechX challenge [7]. The control software of these systems utilizes the *IPC* communication framework [8], which is a very popular event-based communication library used by a number of robotic research labs worldwide. However, the algorithm can simply be adapt to other event-based communication frameworks, such as for instance *Miro* [9].

MBR has been actively studied in the past. The Livingstone architecture by Williams and colleagues [10] was used on the space probe *Deep Space One* to detect failures in the hardware and to recover from them. It has also been successfully applied for fault detection and localization in digital circuits and car electronics and for software debugging of VHDL [4]. In [11] the authors show the application of MBR for the diagnosis of a group of robots in the health care domain. The system model comprises interconnected finite state automata. In [12] MBR was presented for monitoring component-based software. The behavior of software components was modeled by Petri nets, where nodes represented the state of components, and transitions the interactions. Verma and colleagues [13] utilized particle filters to estimate the state of the robot and its environment. These estimations together with a model of the robot were used to detect failure situations.

The reminder of this paper is structured as follows. In Section 2 the model learning from observed communication and in Section 3 the model-based diagnosis are discussed. In Section 4 we present experimental results and conclude in Section 5.

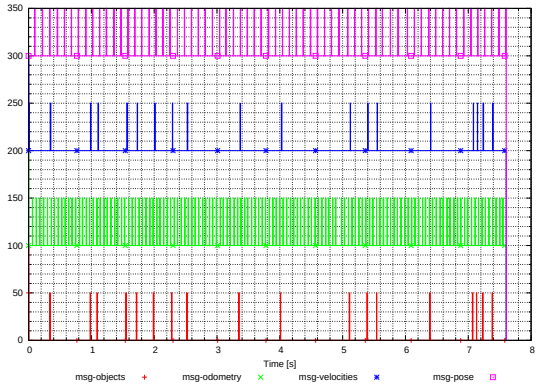


Fig. 1. Recorded communication of the example robot control software. The peaks indicate the occurrence of the particular event.

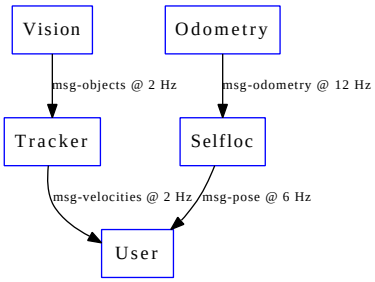


Fig. 2. Communication graph learned from the recorded data of the example control software.

2 Model Learning

Control systems based on IPC use an event-based communication paradigm, i.e. software modules provide data by publishing events, and other modules subscribe for this events in order to receive the data shortly after submission. Typically a central module is in charge of handling all communication, which can also be utilized for recording and monitoring all events. For example, the central server of IPC is able to record the type of the event, the time the event was published or consumed, the content of the event, and the names of the publishing and the receiving modules. In our implementation we use this data for creating a model of the system. Formally, an event is defined as follows.

Definition 1 (Event) *An event e is a tuple* (l,t_P,t_C,d,P,C) *where:*

- l *is the label or name of the event* e
- t_P *is the time the event was published*
- t_C *is the time the event was consumed*
- d *is the data or payload of the event*
- P *is the publisher of the event*
- C *is the consumer of the event*

Please note that if an event is consumed by multiply, each consumption is separately recorded.

Figure 1 depicts the recorded events while running a simple control software example that comprises only five modules with a simple communication structure. In the example there are two data paths, one for processing self-localization, and another one for tracking objects. Whereas the software modules *Odometry*, *Vision*, and *SelfLoc* provide data on a regular basis, the *Tracker* module provides data only if objects have been detected in the data published by the *Vision* module. Figure 1 shows the timing of event publishing, and Figure 2 the extracted

communication graph. Communication graphs are not only useful for diagnosis, they also expressively visualize the relation of modules from a larger or partially unknown control software. In the following model learning algorithm will be described based on this example.

2.1 The communication graph

At a first step the algorithm extracts a communication graph from the data, where nodes represent different software modules, and edges different events that are exchanged between the modules. Each event is represented by at least one edge, whereas edges can also connect to multiple receiving modules originating from a single publishing module. Formally, the communication graph can be defined as following:

Definition 2 (CG) *A communication graph (CG) is a directed graph with the set of nodes M and the set of labeled edges C, where:*

- *M is a set of software modules sending or receiving at least one event.*
- *C is a set of connections between modules, the direction of the edge points from the sending to the receiving module, the edge is labeled with the name of the related event.*

Please note that the communication graph may contain cycles. Usually such cycles emerge from hand shaking mechanisms between two modules. The algorithm for the creation of the communication graph can be formalized as following:

computeGraph
Input: a set of recorded events E
Output: a set of nodes M and edges C

1. Let M be the empty set.
2. Let C be the empty set.
3. For all $e \in E$:
   (a) If $p(e) \notin M$ add $p(e)$ to M.
   (b) If $c(e) \notin M$ add $c(e)$ to M.
   (c) If $(p(e), c(e), l(e)) \notin C$ add $(p(e), c(e), l(e))$ to C
4. Return M and C.

The algorithm starts with an empty set of nodes M and edges C and then iterates trough the set E of all recorded communication events. If either the sender or the receiver are not in the set of the nodes the sender or the receiver is added. If there is no edge pointing from the sending to the receiving node with the proper label, a new edge with the appropriate label is added between the two modules. The functions $p(e)$, $c(e)$, $l(e)$ return the publisher, the consumer and the label of an event c. Moreover, we define the two functions $in: CO \mapsto 2^C$ and $out: CO \mapsto 2^C$ which return the edges pointing to and from a node.

2.2 The communication behavior

In a next step the behavior or type of each event connection is determined. For this purpose we consider the output and input edges of the publishing node, and the recorded timing of each communication via these edges. We distinguish the following event types: triggered event connection (1), periodic event connection (2), bursted event connection (3) and random event connection (4). In order to describe the behavior of a connection formally we define a set of connection types $CT = \{periodic, triggered, bursted, random\}$ and a function $ctype : C \mapsto CT$ which returns the type of a particular connection $c \in C$. The type of an event connection is determined by tests like measurements of the mean and the standard deviation of the time between the occurrence of the events on the connection, and comparison or correlation of the occurrence of two events. The criteria used to assign an event connection to one of the four categories are summarized below:

triggered In oder to determine if an event connection is a triggered event connection, the events on connection $c \in out(m)$ are correlated to the events on the set of input connection to the software module $I = in(m)$. If the number of events on connection c, which are correlated with an event on a particular connection $t \in in(m)$, exceed a certain threshold, connection t is named as trigger of connection c. The correlation test looks for the occurrence of the trigger event prior the observed event. Note each trigger event can only trigger one event. If connection c is correlated with at least one connection $t \in in(m)$ connection c is categorized as a triggered connection. Usually, such connections are found in modules performing calculations only if new data is available.

periodic On a periodic event connection the same event regularly occurs with a fixed frequency. We calculate from the time stamps of the occurrence of all events a discrete distribution of the time difference between two successive events. If there is a high evidence in the distribution for one particular time difference, the connection is periodic with a periodic time of the estimated time difference. For a pure periodic event connection one gets a distribution close to a Dirac impulse. Usually, such connections are found with modules providing data at a fixed frame rate, such as a module sending data from a video camera.

bursted A bursted event is similar to the periodic event but its regular occurrence can be switched on and off for a period of time. A event connection is classified as bursted if there exist time periods where the criteria of the periodic event connection hold. Usually, such connections are found with modules which do specific measurements only if the central controller explicitly enable them, e.g., a complete 3d laser scan.

random For random event connections none of the above categories match and therefore no useful information about the behavior of that connection can be derived. Usually, such connections are found in modules which provide data only if some specific circumstance occur in the system or its environment.

In the case of the above example, the algorithm correctly classified the event connections *odometry*, *objects* and *pose* as periodic and the connection *velocity* as triggered with the trigger *objects*.

2.3 The observers

In order to be able to monitor the actual behavior of the control software, the algorithm instantiates an observer for each event connection. The type of the observer is determined by the type of the connection and its parameters, estimated by the methods described before. An observer raises an alarm if there is a significant discrepancy between the currently observed behavior of an event connection and the behavior learned beforehand during normal operation. The observer provides as an observation O the atom $ok(l)$ if the behavior is within the tolerance and the atom $\neg ok(l)$ otherwise. Where l is the label of the corresponding edge in the communication graph. The observations of the complete control software OBS are the union of all individual observations

$$OBS = \bigcup_{i=1}^{n} O_i$$

where n is the number of observers.

Observers can be instantiated for either triggered, periodic, bursted, or random connections. The *trigger* observer raises an alarm if within a certain timeout after the occurrence of a trigger event no corresponding event occurs or if the trigger event is missing prior the occurrence of the corresponding event. In order to be robust against noise, the observer uses a majority vote for a number of succeeding events. The *periodic* observer raises an alarm if there is a significant change in the frequency of the events on the observed connection. The observer checks if the frequency of successive events does vary significantly from the specified frequency. For this purpose, the observer estimates the frequency of the events within a sliding time window. The *bursted* observer is similar to the periodic observer. It differs in the fact that it starts the frequency check only if events occur and does not raise an alarm if no events occur. Finally, the *random* is a dummy observer which alway provides the observation $ok(l)$. This observer is implemented for completeness.

2.4 The system description

The communication graph together with the type of the connections is a sufficient specification of the communication behavior of the robot control software. This specification can be used in order to derive a system description for the diagnosis process. It is a description of the desired or nominal behavior of the system. In order to be able to be used in the diagnosis process, the system description is automatically written down as a set of logical clauses. This set can easily be derived from the communication graph and the behavior of the connections. The algorithm to derive the system description can be formalized as following:

computeSD
Input: the communication graph with nodes M and connections C
Output: a set of clauses

1. Let SD be the empty set.
2. For all $c \in C$:
   If $host(p(c)) \neq host(c(c))$
   (a) If $ctype(c) = triggered$ add

$$\neg AB(p(c)) \bigwedge_{t \in trigger(c) \wedge t \in in(p(c))} ok(t) \wedge$$

$$\wedge \neg AB(host(p(c))) \wedge \neg AB(host(c(c))) \rightarrow ok(c)$$

   to SD
   Else add

$$\neg AB(p(c)) \wedge \neg AB(host(p(c))) \wedge \neg AB(host(c(c))) \rightarrow ok(c)$$

   to SD
   Else
   (b) If $ctype(c) = triggered$ add

$$\neg AB(p(c)) \bigwedge_{t \in trigger(c) \wedge t \in in(p(c))} ok(t) \rightarrow ok(c)$$

   to SD
   Else add

$$\neg AB(p(c)) \rightarrow ok(c)$$

   to SD
3. For all $m \in M$:
   Add

$$\bigwedge_{c' \in out(m)} ok(c') \rightarrow \neg AB(m)$$

   to SD
4. Return SD.

The functions $p(c)$ and $c(c)$ returns the publishing and receiving module of an event connection c. The function $host(m)$ returns the host a particular module m is running on. The algorithm starts with an empty set SD. For every event connection in two steps, clauses are added to the system description. In the first step, a clause for forward reasoning is added. The clause specifies if a module works correct and all related inputs and outputs behave as expected. Depending on the type of the connection, we add the following clause to SD. If connection c is *triggered*, we add a clause that states that if the module and all related inputs work as expected, also the output has to work as expected. Otherwise a clause is added that states if the module works as expected also the output has to work as expected (see Line 2). $\neg AB(m)$ means that the module m is not abnormal and the module works as expected. The atom $ok(c)$ specifies that the connection c behaves as expected. Moreover, if the hosts of the sending and receiving module of a connection c are different a fact that the network interfaces of these modules have to work correct is added, e.g., $\neg AB(host(p(c))$.

In a second step, a clause for backward reasoning is added. The clause specifies if all output connections c' of module m behave as expected, the module itself has to behave as expected (see Line 3). Figure 4 depicts the system description obtained for the above example control software.

3 Model-based diagnosis

For the detection and localization of faults we use the consistency-based diagnosis technique of [3]. A fault detectable by the derived model causes a change in the behavior of the system. If such an inconsistency between the modeled and observed behavior emerges, a failure has been detected. Formally, we define this by:

$$SD \cup OBS \cup \{\neg AB(m) | m \in M\} \models \bot$$

where the latter set says that we assume that all modules work as expected.

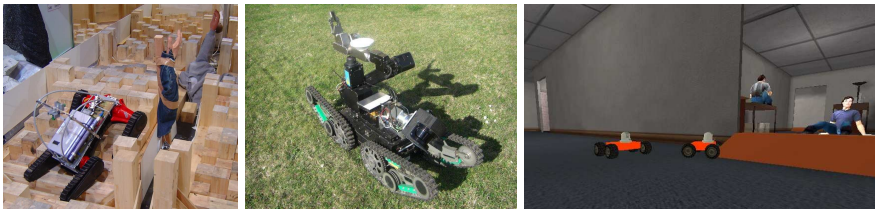
In order to localize the module responsible for the detected fault, we have to calculate a diagnosis Δ. Where Δ is a set of modules $m \in M$ we have to declare as faulty (change $\neg AB(m)$ to $AB(m)$) in order to resolve the above contradiction. We use our implementation[3] of this diagnosis process for the experimental evaluation of the models. Please refer to [1, 2] for the detail of the diagnosis process.

4 Experimental Results

In order to show the potential of our model learning approach, the approach has been tested on three different types of robot control software. We evaluated whether the approach is able to derive an appropriate model reflecting all aspects of the behavior of the system. The derived model was evaluated by the system engineer who has developed the system. Moreover, we injected artificial faults like module crashes in the system, and evaluated if the fault can be detected and localized by the derived model.

A small example control software The example software from the introduction comprises five modules. The module *Odometry* provides odometry data at a regular basis. This data is consumed by the module *SelfLoc*, which does pose tracking by integrating odometry data, and providing continuously a pose estimate to a visualization module *User*. The module *Vision* provides position measurements of objects. The module *Tracker* uses this measurements to estimate the velocity of the objects. New velocity estimations are only generated if new data is available. The velocity estimates are also visualized by the GUI. Figure 1 shows the recorded communication of this example. Figure 2 depicts the communication graph extracted from the recorded data. It correctly represents the actual communication structure of the example, and shows the correct relation of event producers and event consumers.

[3] The implementation can freely be downloaded at http://www.ist.tugraz.at/mordams/.



(a) (b) (c)

Fig. 3. Three autonomous navigation systems that have been evaluated. (a) The rescue robot Lurker, (b) the Telemax robot, and (c) a team of four Zerg robots during exploration in the USARSim environment.

Moreover, the algorithm correctly identified the type of the event connections. This can be seen by the system description the algorithm has derived which is depicted in Figure 4. It also instantiated the correct observer for the four event connections. A periodic event observer was instantiated for *odometry*, *objects* and *pose*, and a triggered event observer was instantiated for *velocities*. Figure 4

1. $\neg AB(Vision) \rightarrow ok(objects)$
2. $\neg AB(Odometry) \rightarrow ok(odometry)$
3. $\neg AB(Tracker) \wedge ok(objects) \rightarrow ok(velocities)$
4. $\neg AB(Selfloc) \rightarrow ok(pose)$
5. $ok(objects) \rightarrow \neg AB(Vision)$
6. $ok(odometry) \rightarrow \neg AB(Odometry)$
7. $ok(velocities) \rightarrow \neg AB(Tracker)$
8. $ok(pose) \rightarrow \neg AB(Selfloc)$

Fig. 4. The system description automatically derived for the example control software.

depicts the extracted system description. Clauses 1 to 4 describe the forward reasoning. Clauses 5 to 6 describe the backward reasoning. Clause 3 states that the module *Tracker* works correctly only if a velocity event occur only after a trigger event. For instance, Clause 6 states that if all output connections of module *Odometry* work as expected, consequently the module itself works correct. This automatically generated system description was used for diagnosis tests, where we randomly shutdown modules and evaluated the fault concluded by the system. During all tests the faults were identified properly.

Autonomous exploration robot Lurker In a second experiment we recored the communication of the control software of the rescue robot Lurker [5] while the robot was autonomously exploring an unknown area. The robot is shown in Figure 3 (a).

The control software of this robot is far more complex as in the simple example since it comprises software modules enabling autonomous exploration of

rough terrain. Figure 5 (a) shows the communication graph derived from the recorded data. The numbers in the labels of the edges denote the average frequency of events on the connections. Please note that a frequency of 0 Hz means the actual frequency is below 1 Hz. From the communication graph and the categorized event connections a system description with 70 clauses with 51 atoms and 35 observers was derived. After a double check with the system engineer of the control software it was confirmed that the automatically derived model maps the behavior of the system.

Autonomous exploration robot Telemax. In this experiment we record data from the software system of the autonomous Telemax robot, shown in Figurer 3 (b), which has been designed for the *TechX Challenge*. The communication was recorded from active software modules for controlling the robot to detect, enter, and operate an elevator.

The communication graph and the system description were derived from the recorded data. The communication graph comprises 18 nodes (software modules) and 51 edges (connections). From the communication graph and the categorized event connections a system description with 63 clauses with 63 atoms and 51 observer was derived. Due to space limitation we omit the picture of the graph in this paper. A review of the system engineer confirms that the generated graph and system description reflect the desired structure and behavior of the system.

Autonomous exploration with a group of Zerg robots. In this experiment we record data during an autonomous exploration run of a group of four Zerg robots within the USARSim environment used in the RoboCup Rescue Virtual Robot League [14]. The robots are shown in Figure 3 (c).

A central control station coordinates the exploration of the individual robots. The central station module and the control software of the robots run on different hosts. From the recorded communication of the central software we extract the communication graph, the categorized event connections and a system description with 48 clauses with 44 atoms and 36 observer was derived. Figure 5 (b) shows the communication graph derived from the recorded data

This system description was used in a diagnosis experiment. During an autonomous exploration run we switched-off the network interface of robot 3 and 4. This failure situation was immediately recognized by 3 observers which raised an alarm. The output of all observers (36 literals) together with the above obtained system description were insert into the diagnosis engine. Based on the system description and the observations, the engine concluded the correct root cause of the problem, i.e., the network interface of robot 3 and 4: $AB(zerg3)$ and $AB(zerg4)$. It has to be noted that these root causes could not be directly observed. This result clearly shows the benefit of model-based diagnosis for the robustness of robot navigation software.

5 Conclusion and Future Work

In this paper we presented an approach which allows the automated learning of communication models for robot navigation software. The approach is able to

automatically extract a model of the behavior of the communication within a component-orientated control software. Moreover, the approach is able to derive a system description which can be used for model-based diagnosis. The approach was successfully tested on *IPC*-based navigation software like the one used by the rescue robot Lurker. Since *IPC* is widely used, our approach is instantly usable on many different robot systems.

The presented implementation can be extended for model learning on any component-based system using an event-based publisher-subscriber mechanism for communication. Currently, we are working on a port for *Miro*-based systems. In future work, we will work on methods that also analyze the content of messages, e.g., methods that are able to distinguish between data under normal and abnormal conditions. We believe that more context knowledge will further increase the robustness of model-based reasoning.

References

1. Steinbauer, G., Wotawa, F.: Detecting and locating faults in the control software of autonomous mobile robots. In: 16th International Workshop on Principles of Diagnosis (DX-05), Monetrey, USA (2005) 13–18
2. Steinbauer, G., Mörth, M., Wotawa, F.: Real-Time Diagnosis and Repair of Faults of Robot Control Software. In: RoboCup 2005: Robot Soccer World Cup IX. Volume 4020 of Lecture Notes in Computer Science., Springer (2006) 13–23
3. Reiter, R.: A theory of diagnosis from first principles. Artificial Intelligence **32**(1) (1987) 57–95
4. Friedrich, G., Stumptner, M., Wotawa, F.: Model-based diagnosis of hardware designs. Artificial Intelligence **111**(2) (1999) 3–39
5. Kleiner, A., Dornhege, C.: Real-time Localization and Elevation Mapping within Urban Search and Rescue Scenarios. Journal of Field Robotics (2007)
6. Ziparo, V., Kleiner, A., Nebel, B., Nardi, D.: RFID-based exploration for large robot teams. In: Conference on Robotics and Automation. (2007) 4606–4613
7. DTSA: Techx challenge. http://www.dsta.gov.sg/index.php/TechX-Challenge (2008)
8. Simmons, R.: Structured Control for Autonomous Robots. IEEE Transactions on Robotics and Automation **10**(1) (1994)
9. Utz, H.: Advanced Software Concepts and Technologies for Autonomous Mobile Robotics. PhD thesis, University of Ulm, Neuroinformatics (2005)
10. Muscettola, N., Nayak, P.P., Pell, B., Williams, B.C.: Remote agent: To boldly go where no AI system has gone before. Artificial Intelligence **103**(1-2) (August 1998) 5–48
11. Micalizio, R., Torasso, P., Torta, G.: On-line monitoring and diagnosis of a team of service robots: A model-based approach. AI Communications **19**(4) (2006) 313 – 340
12. Grosclaude, I.: Model-based monitoring of component-based software systems. In: 15th International Workshop on Priciples of Diagnosis, Carcassonne, France (2004) 155–160
13. Verma, V., Gordon, G., Simmons, R., Thrun, S.: Real-time fault diagnosis. IEEE Robotics & Automation Magazine **11**(2) (2004) 56 – 66
14. Balakirsky, S., Carpin, S., Kleiner, A., Lewis, M., Visser, A., Wang, J., Ziparo, V.A.: Towards heterogeneous robot teams for disaster mitigation: Results and Performance Metrics from RoboCup Rescue. Journal of Field Robotics **24**(11-12) (2007) 943–967

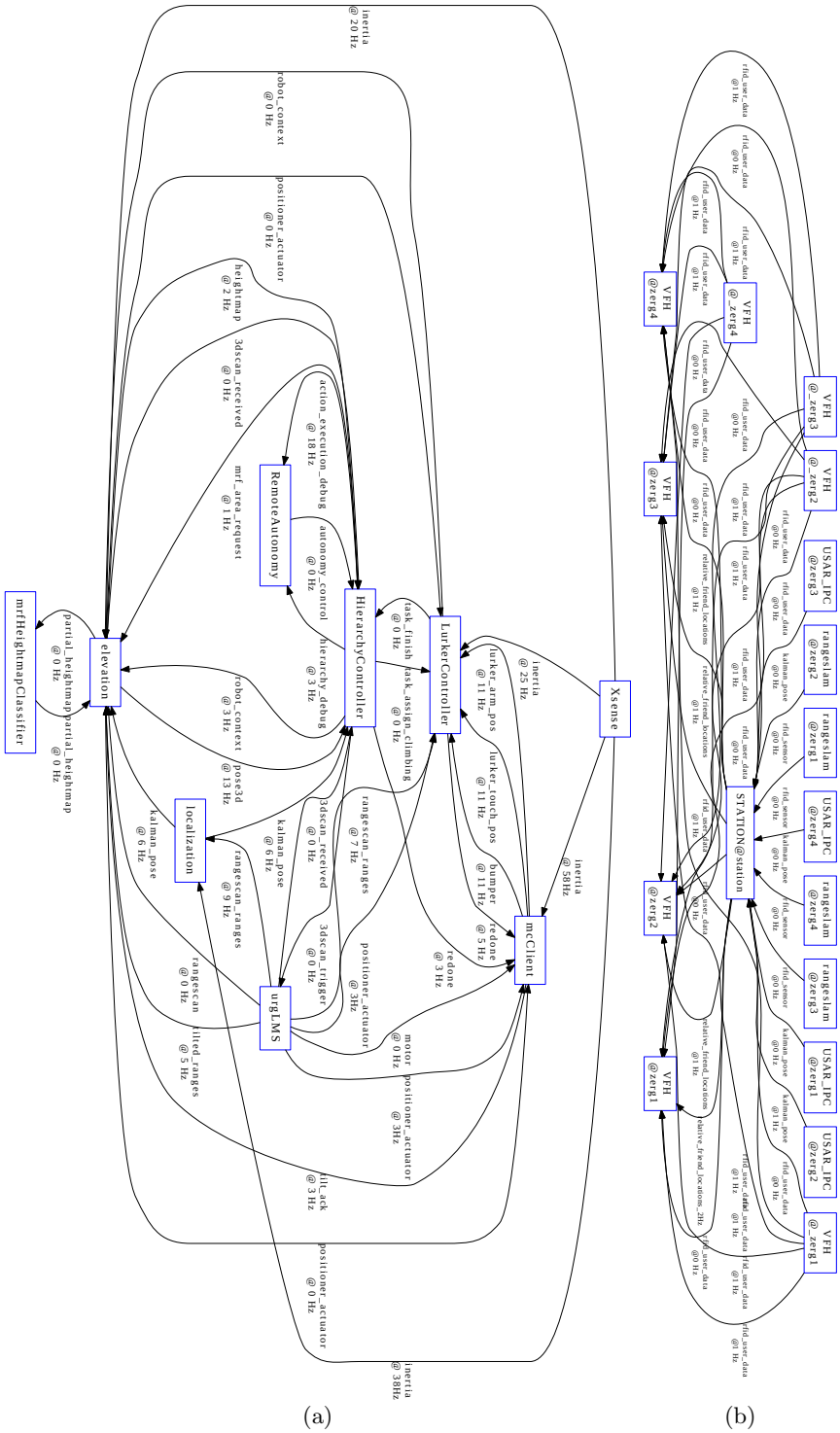


Fig. 5. Two learned communication graphs. (a) Communication graph of the Lurker robot. (b) Communication graph of the central module for the multi robot scenario with the Zerg robots. The name of the host a module is running on is depicts in the label of the node.