Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 00 (2018) 000–000

Procedia Computer Science

www.elsevier.com/locate/procedia

The 16th International Conference on Mobile Systems and Pervasive Computing (MobiSPC)
August 19-21, 2019, Halifax, Canada

Towards an Actor-based Approach to Design Verified ROS-based Robotic Programs using Rebeca

Saeid Dehnavi[a,b], Ali Sedaghatbaf[b], Bahar Salmani[c], Marjan Sirjani[b,*], Mehdi Kargahi[a], Ehsan Khamespanah[a,d]

[a] *School of ECE, College of Engineering, University of Tehran, Iran*
{*sdehnavi, kargahi, e.khamespanah*}*@ut.ac.ir*
[b] *School of Innovation, Design and Engineering, Mälardalen University, Västerås, Sweden*
{ *ali.sedaghatbaf, marjan.sirjani*}*@mdh.se , sdi18002@student.mdh.se*
[c] *Department of Informatik, RWTH-Aachen University, Germany , salmani@informatik.rwth-aachen.de*
[d] *School of Computer Science, Reykjavik University, Iceland , ehsank@ru.is*

Abstract

Robotic technology helps humans in different areas such as manufacturing, health care and education. Due to the ubiquitous revolution, today's focus is on mobile robots and their applications in a variety of cyber-physical systems. ROS is a wll-known and powerful middleware that facilitates software development for mobile robots. However, this middleware does not support assuring properties such as timeliness and safety of ROS-based software. In this paper we present an integration of Timed Rebeca modeling language with ROS to synthesize verified robotic software. First, a conceptual model of robotic programs is developed using Timed Rebeca. After verifying a set of user-defined correctness properties on this model, it is translated to a ROS program automatically. Experiments on some small-scale case studies illustrates the applicability of the proposed integration method.

Keywords: Robotics; Model-Based Design; ROS; Formal Verification; Timed Rebeca; Model Checking; Mobility.

1. Introduction

In previous decades, robots were mostly used to perform repetitive tasks in a single place. With the ubiquitous revolution, mobile robots are finding their place in a wide variety of application areas e.g. space exploration, autonomous transportation, education and health monitoring [1][2]. In most of these applications, mobile robots are a crucial component of networked computational and physical resources, generally referred to as Cyber-Physical Systems (CPSs)

* Corresponding author. Tel.: +46-73-66-20517
E-mail address: marjan.sirjani@mdh.se

[3]. However, regarding the ever-increasing complexity of CPSs, designing mobile robots and programming their behavior are complicated tasks which involve different fields of science and engineering [4]

Robot middlewares e.g. ROS [5] and YARP [6] provide an abstraction layer between the operating system and robotic applications to ease robot application development without the need to deal with low-level complexities. ROS is a standard robot middleware which is currently the principal robotic software framework in well-known companies e.g. Yujin Robotics, Radney Brooks, Hartland Robotics, Yaskawa Motoman and Texte Technologies. While ROS provides a lot of features to facilitate robotic software development, analyzing developed software to guarantee correctness properties is challenging. Although, there are several model-checking techniques and tools that help designers and developers detect faults early, finding a correct implementation of a verified model is mostly problematic. Therefore, integrating a model checking tool with a robot middleware is crucial to have reliable software applications on ROS.

In this paper, Timed Rebeca [7] is used to formally model the robots' behaviors. Timed Rebeca is a timed extension of Rebeca [8], which is an actor-based modeling language supported by a powerful model checker called Afra [9], [10]. Using Afra, we are able to verify several properties on on Rebeca models and also perform performance evaluation on them [11].

The general approach for integrating Timed Rebeca with ROS is to model a robot movement scenario with Timed Rebeca and verify correctness properties such as timeliness and safety on that scenario using Afra. Then, the equivalent ROS implementation of the model will be generated automatically. The generated ROS program can be run on real robots or simulation engines e.g. Stage [12]. In summary, our contributions in this paper can be listed as follows:

- Developing a mapping between a robot movement scenario and a Timed Rebeca model for model checking purposes.
- Developing a mapping between the Timed Rebeca model and ROS programming constructs.
- Automatic generation of the ROS-based robotic programs from Timed Rebeca models.

In the rest of this paper, first we review some previous work related to the scope of this paper in Section 2. Then, we provide some background information on Timed Rebeca and ROS in section 3. In section 4, we present an abstract model for robot movement scenarios and its corresponding Timed Rebeca and ROS elements. Then, we will explain our mapping mechanism in Section 5. A simple case study is presented in 6, and finally some concluding remarks and our future directions are pointed out in section 7.

2. Related Work

The work related to the scope of this paper can be grouped into two categories: 1) formal verification of robotic programs, and 2) code synthesis from formal models. In the following, we will elaborate some outstanding contributions belonging to each category.

Formal Verification: A formal verification approach for industrial robot software is proposed by Webster et al. in [13]. They use SPIN model checker as the verification tool in their proposed approach. Although the result achieved by their approach is considerable, it is limited to a specific robot type ("Care-o-Bot" robot [14]) and is not also generalized to the domain of all ROS programs. Huang et al. propose ROSRV as a runtime verification framework for ROS programs [15]. ROSRV verifies the security and safety of robotic applications by adding a separate ROS node monitoring the behavior of the other ROS nodes in the application. Unacceptable states which lead the robot's behavior to violate the desired properties are predicted and prevented by the monitor node. Apart from the benefits, ROSRV imposes a considerable overhead on the inter-robot communications. Adam et al. [16] have focused on verifying the safety properties of mobile robots at runtime. They have introduced a domain-specific language named DeRoS to define safety properties. Similar to ROSRV, a monitor node is automatically added to the application to monitor the behavior of the other nodes. However, due to the reliance on only a single monitor for the whole application, scalability is the common limitation for both of the above methods.

Code Synthesis: There are some research conducted on robot code synthesis from a model verified by formal methods. Li et al. in [17] propose a code synthesis method to generate executable C++ code from a verified timed automata model. This method is applicable to single-robot applications. Mobility of the robots is also ignored in the

proposed method. SCADE is another tool which has a formal basis, and has been successfully applied to a variety of applications other than robotic programs [18]. Times tool [19] automatically synthesizes C code from Timed Automata models. However, SCADE and Times have no support for ROS, which is the most popular robotic framework in the industry and academy. On the other hand, the advantage of Timed Rebeca over other timed modeling languages like TCCS [20], Real-Time Maude [21], and Timed Automata is its intuitive and easy-to-use syntax and the actor-based paradigm of modeling in which there is no need for any knowledge of formal methods [22].

3. Background

In this section, we present a brief introduction on Rebeca modeling language and Robotic operating System, since developing a correct mapping needs knowledge about the source language and the destination of the mapping.

3.1. Rebeca and Timed Rebeca

Rebeca is an actor-based modeling language developed to facilitate formal verification of concurrent applications. Each Rebeca model consists of a set of *reactive class* declarations and a *main* block. The main block defines instances of reactive classes, called *rebecs*. Each reactive class comprises the following three parts:

- **known rebecs:** the rebecs to which the rebecs of this class may send message to.
- **state variables:** variables that together with the queue content, indicate the state of a rebec of this class.
- **message servers:** specify how a rebec of this class processes incoming messages. Each rebec takes a message from its message queue and executes its corresponding message server. Execution of a message server body takes place atomically (non-preemptively). The behavior of a Rebeca model is defined as the parallel execution of the message servers of the rebecs.

In Timed Rebeca, each rebec has its own local clock, but there is also a notion of global time based on synchronized distributed clocks of all the rebecs. Timing primitives are added to the Rebeca syntax to cover timing features that a modeler might need to address in a message-based, asynchronous and distributed setting. These primitives include: (1) **Delay**: *delay(t)*,it takes *t* units of time to pass this statement, (2) **Deadline**: *r.m() deadline(t)*, after *t* units of time the message is not valid any more and is purged from the queue (timeout), and (3) **After**: *r.m() after(t)*, the message cannot be taken from the queue before *t* time units have passed. A sample Timed Rebeca reactiveclass is depicted in Fig. 1.

3.2. ROS Middleware

In the following, we elaborate the primitive concepts in ROS programming [24].

ROS Nodes: In ROS, the main processing units are called nodes. Nodes are defined to perform basic tasks such as sending/receiving messages and performing computations. There is also a special node called *Master Node* which holds all the information about the nodes in the ROS network. A ROS package consists of nodes, libraries, configuration files, launch files and other related components.

ROS Communication: Communication in a ROS program can take place in a number of ways, but the most common one is *Publish-Subscribe*. In this communication model, there is a *topic* on which publishers send messages, and from which subscribers receive messages. In other words, a topic is a named channel shared between publishers and subscribers. The communication is done in an anonymous way such that publishers and subscribers do not know each other directly. If a node wants to give its information to other nodes, it should include that information in the message bodies.

Distributed Parameter Server: The parameter server is a shared and multivariate dictionary that stores the configuration information.

ROS Messages: As it was mentioned, ROS nodes communicate with each other by sending messages on topics. A message is a simple data structure that consists of a number of typed fields varying from simple types such as Int to complex custom types. A custom type can be defined by adding a *Msg* file in the ROS package.

```
reactiveclass Rebec1(2){

  knownrebecs{
      Rebec2 r2;
      Rebec3 r3;
  }

  statevars{
      int st1;
      float st2;
  }

  //constructor
  Rebec1(int s, float s2){
      st1 = s;
      st2 = s2;
      self.msg1();
  }

  //a message server
  msgsrv msg1(){
      r2.msg2();
      delay(method1());
      r3.msg3() after(2);
  }

  //a method
  int method1(){
      return 3;
  }
}
```

Fig. 1. A sample Timed Rebeca reactive class [23].

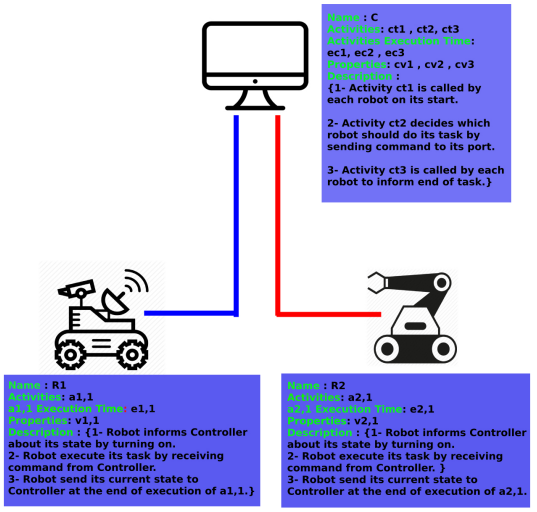

Fig. 2. A simple example of a general robotic program model

4. Problem Formulation

4.1. An Abstract Model for Robot Movement Scenarios

An example of robot movement scenarios is shown in Fig. 2. This example includes two types of mobile robots and a central controller. Each robot has a single activity with a specified execution time. The central controller is the node that decides which robot should run its activity. Each robot reports its current state to the controller after executing its activity, and it has a port to receive controller's commands. In this example, we assume that each robot has only one sensor which provides data to the robot in a specified sensing rate. Each robot has a single activity, which is movement in a direction (R1 in direction X and R2 in direction Y). After movement, each robot should update its position and inform the controller. The controller decides which robot should move at each time and the movement should be done in such a way that no collision happens between the robots. This process is repeated until both robots reach their goal positions which are decided by the controller.

As illustrated by this example, in a mobile robot movement scenario, there are some robots moving in an environment and communicating with each other. The set of robots can be defined as $R = \{R_i : i = 1, 2, ..., M\}$ such that:

$$R_i = \{p_i, V_i, A_i, S_i\}$$

Each robot R_i has a command port p_i which is used as the communication bridge between the central controller and robot R_i. Additionally, each robot has N properties which are defined by set $V_i = \{v_{ij} : j = 1, 2, ..., N\}$. Since robots are responsible to do some activities, K activities are assigned to each robot R_i that are defined by $A_i = \{(a_{ij}, e_{ij}) : j = 1, 2, ..., K\}$ in which each activity a_{ij} has an estimated execution time indicated by e_{ij}. Finally, for sensing the environment and providing the required data, each robot R_i is accompanied with U sensors $S_i = \{(s_{ij}, sr_{ij}) : j = 1, 2, ..., U\}$. Each sensor s_{ij} has a specific function which is run when each unit of data is sensed by the sensor. The time between sensing two units of data is specified by the sensing rate of the sensor which is indicated by sr_{ij}.

In addition to robots, there is also a central controller which monitors the behavior of robots and sends them some commands.The central node has some tasks to do which are modeled by $CT = \{ct_j : j = 1, 2, ..., T\}$. It also may have some variables to store the state of the system that can be modeled by $CV = \{cv_j : j = 1, 2, ..., W\}$. Therefore, the central controller can be defined as follows:

$$CNode = \{CT, CV\}$$

Based on the above discussion, the overall model for the example scenario can be formulated as follows:

$$R = \{R_1, R_2\}$$
$$R_1 = (p_1, V_1, A_1, S_1), V_1 = \{v_{11}\},$$
$$\mathrm{A}_1 = \{(a_{11}, e_{11})\}, S_1 = \{(s_{11}, sr_{11})\}$$
$$R_2 = (p_2, V_2, A_2, S_2), V_2 = \{v_{21}\},$$
$$\mathrm{A}_2 = \{(a_{21}, e_{21})\}, S_2 = (s_{21}, sr_{21})$$
$$CNode = \{CT, CV\}$$
$$\mathrm{CT}=\{(\mathrm{ct}_1, e_{c1}), (ct_2, e_{c2}), (ct_3, e_{c3})\}, CV = \{cv_1, cv_2, cv_3\}$$

4.2. Timed Rebeca Model

As it was mentioned in the previous section, each Rebeca model RM is composed of M reactive classes such that $RM = \{RC_i : i = 1, 2, ..., M\}$. Each reactive class RC_i has N message servers specified by $M_i = \{m_j : j = 1, 2, ..., N\}$. In addition to message servers, there might be M local methods defined for each reactive class, which are defined as $L_i = \{l_j : j = 1, 2, ..., M\}$. The *knownrebec* set of a rebec is specified by $K_i = \{k_j : j = 1, 2, ..., K\}$. Finally, each rebec has a set of S state variables which are defined by $C_i = \{c_j : j = 1, 2, ..., S\}$. Accordingly, each reactive class in the Rebeca model can be defined as follows:

$$RC_i = \{M_i, L_i, K_i, C_i\}$$

4.3. ROS Model

There are different styles of programming developers may use to develop programs in ROS. One common and standard style of programming in ROS is object-oriented, in which developer defines a class per each active node. Then, all the methods and variables related to the node are put in that class. We assume that a ROS program for a robot movement scenario is composed of M different class definition such that $ROS\,Model = \{ROS\,Class_i : i = 1, 2, ..., M\}$. After defining the overall structure of ROS nodes by the defined classes, each active ROS node is defined as an instantiated object of a specific class. Since we may have more than one created object of the same class, the set of active nodes in ROS can be defined as $ROS\,Node = \{rosnode_i : i = 1, 2, ..., B\}$. As it was mentioned in the previous section, there are a number of predefined topics in each ROS program that can be showed as $T = \{t_i : i = 1, 2, ..., N\}$.

Our approach currently supports only the *Publish-Subscribe* mode of communication, in which each node can publish on a specific topic or subscribe to a topic to receive the data published on that topic. Publish state of the node $rosnode_i$ on the topic t_j is indicated by $publish_{ij}$. Therefore, the list of the topics which the node $rosnode_i$ published on, can be showed by $Publish_i = \{publish_{ij} : j = 1, 2, ..., N\}$. In the same way, the list of topics which the node $rosnode_i$ subscribes to them is defined by $Subscribe_i = \{subscribe_{ij} : j = 1, 2, ..., N\}$. Finally, as it was mentioned in the previous section, each topic has a specific message type to be published on the topic. Therefore, the list of message types in the ROS program is defined by $MT = \{mt_i : i = 1, 2, ..., N\}$. In ROS, the data sensed by sensors are published on some specific topics per each sensor. Therefore, if a node wants to process the sensed data, it should subscribe to the topic related to that sensor.

5. Mapping Mechanism

In this section, we present a mechanism to map Timed Rebeca model to the equivalent ROS implementation of a robot movement scenario. At first, we need to make a mapping between the abstract scenario model and Rebeca to find out how to model robot movement scenarios in Timed Rebeca. Then, we automatically generate the equivalent ROS implementation from the Timed Rebeca model by a logical and meaningful mapping strategy.

5.1. Mapping the Abstract Scenario Model to Timed Rebeca

Since Rebeca is a general purpose modeling language, we need to find a mechanism to model robotic programs in Rebeca. In this section, we present a mapping between the abstract scenario model elaborated in the previous section

and Timed Rebeca. **Individual robots to reactive classes:** It was mentioned in section 4 that we have a number of independent robots in each movement scenario, such that each robot has its own properties and behavior. On the other hand, each actor is modeled by a reactive class in Rebeca, and each reactive class can have its state variables, methods and message servers which represent that actor's behavior. We map each robot in the abstract model to a reactive class in Rebeca. Since we may map other entities in the abstract model to reactive classes in Rebeca, we annotate the reactive classes mapped from robots by *@Robot*.

Sensors to recursive message servers: Each robot consists of a number of sensors each of which has a specific function which is run when a unit of data is sensed by the sensor. On the other hand, it is obvious that each sensor has a specific sensing period which indicates the time interval between sensing two units of data. Therefore, we consider a message server annotated with *@Sensor* per each sensor of the robot in its corresponding reactive class. Since reading sensor data is a periodic task, the message server assigned to the sensor should call itself after t units of time such that t is the sensing period of the sensor. This can be done by using keyword *after(t)* in Timed Rebeca.

Activity to message server: Modeling the activities of a robot in Timed Rebeca is done by defining a message server (without annotation) in Timed Rebeca. In other words, for each activity a_i a message server is considered in the robot's reactive class. It is worth mentioning that each activity has an estimated execution time e_i which can be modeled by $delay(e_i)$ in Timed Rebeca.

Property to state variable: Each robot may have some properties. On the other hand, there is a block named *statevars* in Rebeca which includes a set of variables to hold the current state of a rebec. We map each robot property to a state variable in the *statevars* of the corresponding reactive class.

Controller to reactive class: The point which makes the central controller node different from robots in a robotic program, is its ability to directly communicate with all the robots in the system. Therefore, we consider a reactive class annotated with *@Controller* for the central node which has all the rebecs annotated with *@Robot* in its *Knownrebecs* section and vice versa. It should be mentioned that we map controller's tasks and variables to message servers and state variables, respectively.

Command port to @Port message server: The command port introduced in the previous section, is mapped to a message server annotated by *@Port* in the reactive class corresponding to the robot. All the commands and messages should be sent to the robot from the controller node through this @Port message server.

For illustration, the Rebeca model of the robot *R2* in the explained example model in Fig. 2 can be seen in Fig. 3.

5.2. Mapping the Rebeca Model to ROS

Modeling a robotic program in Rebeca gives a modeler the ability to verify properties such as safety and timeliness. However, the Timed Rebeca model is not executable and cannot be run on real robots. Therefore, after verifying the desired properties using RMC, the Timed Rebeca model is translated to a ROS program.

@Robot reactive class to C++ class: In an object-oriented programming language, class is a structure which defines the state and behavior of the objects in the program. Similarly, reactive classes in Rebeca have the responsibility to define the overall state and behavior of the rebecs. ROS provides client libraries in several programming languages such as C++ and Python. In this paper, we use C++ considering its popularity among robotic programmers. In ROS programming, there should be a C++ class per each robot to include all the behavior and states related to the robots. Then, each object of the defined class should be mapped to a ROS node. Therefore, we consider a one to one mapping between Rebeca reactive classes which are annotated by *@Robot* and C++ classes in ROS programs.

Global variables to parameter server entries: In a Rebeca model, there might be some global variables which are accessible to all the active rebecs in the model. Since all the entries in the ROS parameter server are valid and accessible for all the active nodes in a ROS program, we can map global variables in the Timed Rebeca model to entries in ROS parameter server.

State variables to private variables: All the state variables of a reactive class are only accessible in that class. On the other hand, all the private variables defined in a C++ class can only be accessed through the methods defined in that class. Therefore, we map state variables in the Rebeca model to private variables in the ROS program.

Unannotated message servers to methods, subscribers and publishers: Each message server in Rebeca is composed of different parts. Here we find a mapping for each part:

```
@Robot
reactiveclass R2(2){
  knownrebecs{
      C controller;
  }

  statevars{
      //robot id
      int id;

      //robot stopped?
      boolean stopped;

      //robot's current position
      int[2] pos;
  }

  R2(int r_id, int r_x, int r_y){
      id = r_id;
      stopped = true;
      pos[0] = r_x;
      pos[1] = r_y;
  }

  @Port
  msgsrv P2(int cmd){
      if (cmd == 0){
          //stop to avoid collision
          stopped = true;
          controller.CT1(id, pos[0]+1, pos[1]) after
                (1);
      } else {
          //move
          stopped = false;
          self.A21();
      }
  }

  msgsrv A21(){
      pos[0] = pos[0]+1;
      stopped = false;
      @Move(X+1, Y)
      delay(e21)
      //inform controller
      controller.CT3(id, pos[0], pos[1]);
  }

  @Sensor
  msgsrv S21(){
      //sensed data
      int data = ?(0, 1);
      self.S21() after(sr21);
  }
}
```

Fig. 3. The Timed Rebeca model of R2 in Fig. 2

```
#include <Project/R2.h>
int main(int argc, char** argv){
  ROS_INFO("R2_node_started");
  ros::init(argc, argv, "R2_node");
  ros::NodeHandle nh("~");
  std::string sender;
  nh.getParam("sender", sender);
  int id;
  boolean stopped;
  nh.getParam("id", id);
  int[2] pos;
  nh.getParam("pos", pos);
  R2_r2(id, pos[0], pos[1], sender);
}
R2::R2(int r_id, int i_x, int i_y, std::string_sender){
  A21_sub = n.subscribe("R2/A21", 30, &R2::A21Callback, this);
  S21_sensor_sub = n.subscribe(/*sensor topic*/, 30, &R2::S21SensorCallback, this);
  P1_port_sub = n.subscribe("R2/Port", 30, &R2::PortCallback, this);
  CT1_controller_pub = n.advertise<Project::CT1MessageType>("Controller/CT1", 30);
  CT3_controller_pub = n.advertise<Project::CT3MessageType>("Controller/CT3", 30);
  Simulator = n.advertise<Project::SimulatorMsg>("Simulator", 30);
  sender = _sender;
  while(CT3_controller_sub.getNumSubscribers() < 1 || CT1_controller_sub.
        getNumSubscribers() < 1);
  id = r_id;
  pos[0] = i_x;
  pos[1] = i_y;
  ros::spin();
}
void P2::PortCallback(const Project::PortMessageType & thisMeg){
  if(cmd == 0){
      stopped = true;
      controller.CT1(id, pos[0]+1, pos[1]); sleep(1);
  } else {
      stopped = false;
      A21();
  }
}
void R2::A21Callback(const Project::A21MessageType & thisMsg){
  #define param0 thisMsg.param0
  /*...*/
  #define paramN thisMsg.paramN
  sleep(e21);
  Project::CT3MessageType pubMsg;
  pubMsg.sender = sener;
  pubMsg.posX = pos[0];
  pubMsg.posY = pos[1];
  CT3_controller_pub.publish(pubMsg);
  #undef param0
  #undef paramN
}
void R2::S21SensorCallback(const Project::SensorDataMessageType & thisMsg){
  /*process sensed data*/
}
```

Fig. 4. The ROS code of R2 in Fig. 2

1. **Message server name:** Since rebecs in Rebeca communicate in an asynchronous way, it is easy to map rebec communications to the publish-subscribe model in ROS. Therefore, the name of each unannotated message server in Rebeca is mapped to a *topic* name in ROS.
2. **Message server parameter:** Since each topic in ROS should be limited to a specific message type, we map the list of parameters of a message server to a message type in ROS, and then assign that message type to the topic corresponding to the message server.
3. **Message server caller:** In the Rebeca model, there might be a number of reactive classes in which a message server is called. If we assume the caller reactive class is class *A*, then there should be a publisher in the C++ class mapped from reactive class A to publish on the corresponding topic.
4. **Message server owner:** Since the message server caller has a publisher to publish on the topic associated with the message server, there should be a subscriber in the C++ class defining the message server owner to act as soon as the message server is called.
5. **Message server body:** Since the message server body should be run when the message server is called, it will be mapped to the call function of the subscriber defined in the C++ class mapped from the owner reactive class.

@Sensor message servers to subscribers: The difference between unannotated message servers and the message servers annotated by *@Sensor* is their publisher. In fact, in a sensor message server, there is no publisher defined by the developer as the physical sensor publishes the data. Therefore, this type of message server is mapped to a subscriber in the C++ class. Similar to unannotated message servers, the body of this message server is mapped to the call function of the subscriber. It is worth mentioning that, since there is no physical sensor in the Rebeca model

to generate the modeled data, modeler has to define deterministic expressions in the message server body to model different data values received from the sensor.

@Controller reactive class to controller C++ class: In the Rebeca model, the reactive class annotated by *@Controller* is considered as the central controller which has all the system's rebecs in its *knownrebecs* section. Similar to *@Robot* reactive classes, the *@Controller* reactive class is mapped to a C++ class in the ROS model. However, there is a difference between previous C++ class and the controller C++ class; There should be a special topic named *control_bridge* which is used for communication between the central controller and all the nodes in the ROS program.

@Port message server to Port topic: Each robot has a @Port message server which is used for communication between the central controller and the robot. This message server is mapped to a special topic named Port for each robot. Then, the C++ class mapped from the *@Robot* reactive class should subscribe on the specified topic, since the central controller should have a publisher per each robot's port topic. The body of the message servers is mapped in the same way as typical message servers.

Rebeca Entity	**ROS Entity**
@Robot reactive class	Independent C++ class
Global variables	Parameter server entries
State variables	Private variables in C++ class
Message server name	Topic name
Message server parameters	Message type
Message server holder	Subscriber
Message server body	Subscriber callback function
@Sensor message server	Subscriber on real sensor topic
@Contoller reactive class	controller C++ class with publisher on all command topics
@Port message server	Publisher-Subscriber on *Command* topic
Reactive class Constructor Method	C++ class Constructor Method
Local variables in reactive class	Local variables In C++ class
Rebec creation in models main function	Object creation in main function of the model
Sender reference	Sender parameter in the created message type
Delay() function	Sleep() function in ROS
typical logical and mathematical statements	their equivalents in C++

Table 1. Overview of mapping between different parts Of Rebeca models and ROS programs

Basic Robot Actions: We have defined the mapping mechanism for the abstract model so far. But as we are thinking about implementing the ROS version of the model, some basic decisions such as the movement of the robot should be taken into account. In this paper, we only consider robot's movement as the basic action although some other action could be considered. For each movement, the source and destination positions should be specified in the Rebeca model so that we can generate the required codes in the ROS program. Imagine the current position of the robot is (x,y). Then, it can move to any of the eight locations around it. We can model these movements using simple annotations like *@Move(X-1,Y-1)*. It is worth mentioning that this annotation is ignored by Afra during verification of the Rebeca model. In the mapping process, we generate a special topic per each robot which is used by ROS to decide about the robot's movement. In ROS, this topic is known as *CMD*.

For illustration, the ROS program of the Rebeca model presented in Fig. 2 is shown in Fig. 4. It should be mentioned that *Project/R2.h* is the header file defining the overall interface of the C++ class related to *R2* which is not presented here due to space limitations. An overview of the mapping rules between different entities in Timed Rebeca and ROS can be seen in Table 1.

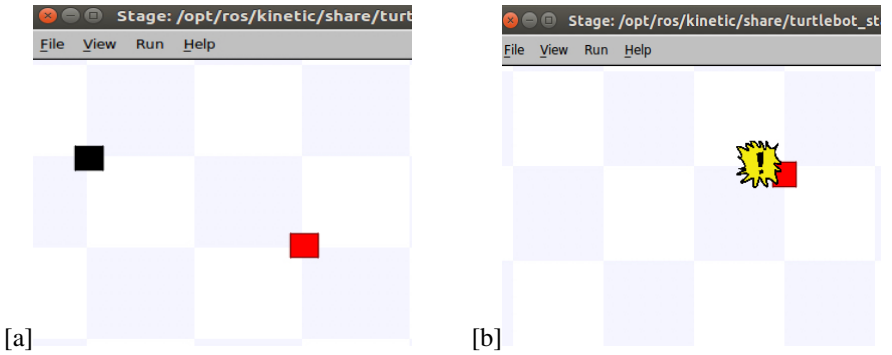

Fig. 5. Screenshots of the simulation for two scenarios

6. Case Study

For investigating the applicability of the proposed mapping mechanism, we developed a tool for automatic generation of ROS programs from Rebeca models using Java programming language. The tool and the case study provided in this paper can be accessed from [25]. For illustrative purposes, we used the Stage simulator to execute the ROS program generated from the Timed Rebeca model. Two different versions of Kabuki and Turtlebot robots were used in this simulation. Moreover, we used Afra for model checking the Timed Rebeca model.

In this case study, we have two mobile robots moving in the environment. We were interested to ensure that the robots would not collide with each other while moving around the environment. Avoiding *collision* of the robots can be considered as a correctness property which should be verified to ensure the safety of the model. The *Collision_Avoidance* property to be verified by Afra can be defined as follows:

```
Property{
    Define{
        xPos = (robot1.pos[0] == 4);
        yPos = (robot1.pos[1] == 2);
    }
    Assertion{
        collision_avoidance: !(xPos & yPos);
    }
}
```

Then we changed the initial positions in the Timed Rebeca model to check if the generated ROS code was correct or not. In the first scenario in which the property *Collission_Avoidance* was satisfied (i.e. there was no collision), the generated ROS code had no collision as well. In the second scenario, we changed the initial positions in the Timed Rebeca model such that the collision happened (i.e. the property was not satisfied). By running the simulator on the generated ROS code using our implemented automatic generator, the collision was also visited in the simulation which shows that our proposed mechanism works correct. In Fig. 5 a screenshot of the simulation environment for the first and the second scenario can be seen.

7. Conclusion and Future Work

As a step toward assuring the quality of robot programs, we proposed a new approach for generating verified robot programs. In this approach, an abstract model of the program is developed and formally verified before the actual program code is generated. Early formal verification allows us to detect and resolve several quality issues before implementing an executable program. The proposed approach is able to verify robot programs, but also it automatically generates executable ROS programs from verified models.

Currently, the proposed approach only supports the publish-subscribe programming style. As future work, we intend to include other popular ROS programming styles (e.g. client-server) in our verification module. Furthermore, we plan to extend the code generator module to support other popular robot middlewares, e.g. YARP.

References

[1] M. Shishehgar *et al.*, "A systematic review of research into how robotic technology can help older people," *Smart Health*, vol. 7, pp. 1–18, 2018.

[2] F. L. Lewis and S. S. Ge, *Autonomous Mobile Robots: Sensing, Control, Decision Making and Applications*. CRC Press, 2018.

[3] E. A. Lee, "Cyber physical systems: Design challenges," in *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, IEEE, 2008, pp. 363–369.

[4] J. Kiener and O. Von Stryk, "Towards cooperation of heterogeneous, autonomous robots: A case study of humanoid and wheeled robots," *Robotics and Autonomous Systems*, vol. 58, no. 7, pp. 921–929, 2010.

[5] M. Quigley *et al.*, "Ros: An open-source robot operating system," in *ICRA workshop on open source software*, Kobe, Japan, vol. 3, 2009, p. 5.

[6] G. Metta *et al.*, "Yarp: Yet another robot platform," *International Journal of Advanced Robotic Systems*, vol. 3, no. 1, p. 8, 2006.

[7] A. Reynisson *et al.*, "Modelling and Simulation of Asynchronous Real-Time Systems using Timed Rebeca," *SCP*, vol. 89, pp. 41–68, 2014.

[8] M. Sirjani *et al.*, "Modeling and verification of reactive systems using rebeca," *Fundamenta Informaticae*, vol. 63, no. 4, pp. 385–410, 2004.

[9] E. Khamespanah *et al.*, "Floating Time Transition System: More Efficient Analysis of Timed Actors," in *Formal Aspects of Component Software - 12th International Symposium, FACS 2015, Rio de Janeiro, Brazil, October 14-16, 2015*, 2016.

[10] E. Khamespanah *et al.*, "An efficient TCTL model checking algorithm and a reduction technique for verification of timed actor models," *SCP*, vol. 153, pp. 1–29, 2018. [Online]. Available: https://doi.org/10.1016/j.scico.2017.11.004.

[11] A. Jafari *et al.*, "Ptrebeca: Modeling and analysis of distributed and asynchronous systems," *Sci. Comput. Program.*, vol. 128, pp. 22–50, 2016. [Online]. Available: http://dx.doi.org/10.1016/j.scico.2016.03.004.

[12] R. Vaughan, "Massively multi-robot simulation in stage," *Swarm intelligence*, vol. 2, no. 2-4, pp. 189–208, 2008.

[13] M. Webster *et al.*, "Toward reliable autonomous robotic assistants through formal verification: A case study," *IEEE Transactions on Human-Machine Systems*, vol. 46, no. 2, pp. 186–196, 2016.

[14] C Schaeffer and T May, "Care-o-bot-a system for assisting elderly or disabled persons in home environments," *Assistive technology on the threshold of the new millenium*, 1999.

[15] J. Huang *et al.*, "Rosrv: Runtime verification for robots," in *International Conference on Runtime Verification*, Springer, 2014, pp. 247–254.

[16] S. Adam *et al.*, "Towards a virtual machine approach to resilient and safe mobile robots," in *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, IEEE, 2016, pp. 1–8.

[17] X. Li *et al.*, "Formal modeling and automatic code synthesis for robot system," in *2017 22nd International Conference on Engineering of Complex Computer Systems (ICECCS)*, IEEE, 2017, pp. 146–149.

[18] F.-X. Dormoy, "Scade 6: A model based solution for safety critical software development," in *Proceedings of the 4th European Congress on Embedded Real Time Software (ERTS08)*, 2008, pp. 1–9.

[19] T. Amnell *et al.*, "Times: A tool for schedulability analysis and code generation of real-time systems," in *International Conference on Formal Modeling and Analysis of Timed Systems*, Springer, 2003, pp. 60–72.

[20] W. Yi, "Ccs+ time= an interleaving model for real time systems," in *International Colloquium on Automata, Languages, and Programming*, Springer, 1991, pp. 217–228.

[21] P. C. Ölveczky and J. Meseguer, "The real-time maude tool," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2008, pp. 332–336.

[22] E. Khamespanah *et al.*, "Timed-rebeca schedulability and deadlock-freedom analysis using floating-time transition system," in *Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentralized control abstractions*, ACM, 2012, pp. 23–34.

[23] A. Jafari *et al.*, *Rebeca user manual*, http://rebbac-lang.org, 2016.

[24] *Ros documentation*, http://wiki.ros.org/Documentation.

[25] S. Dehnavi *et al.*, *Reberos repository*, https://github.com/Sdehnavi/RebeROS.git, 2019.