



Mälardalen University
School of Innovation Design and Engineering
Västerås, Sweden

Thesis for the Degree of Master of Science in Engineering -
Intelligent Embedded System 30.0 credits

MODELLING AND ANALYSING COLLABORATING HEAVY MACHINES

Jayasoorya Jayanthi Surendran Nair
jjr15001@student.mdh.se

Examiner: Cristina Seceneleau
Mälardalen University, Västerås, Sweden

Supervisor 1: Stephan Baumgart,
Volvo CE, Eskilstuna

Supervisor 2: Marjan Sirjani
Mälardalen University, Västerås, Sweden

October 10, 2017

Acknowledgements

I would like to express my heartfelt thanks to my supervisors, Mr. Stephan Baumgart and Prof. Marjan Sirjani for their continuous support during this thesis work. Their patience, motivation, knowledge and guidance have helped me in the succession of best results and also in my own personal development. I am thankful to my Co-supervisor Dr. Ali Jafari for his suggestions and advice. I also want to thank my examiner Prof. Cristina Seceneleau, who gave me useful suggestions in the thesis report.

I am grateful for having the opportunity to perform my master thesis at Volvo Construction Equipment. I want to thank the colleagues from System Architecture Department and Electric Site development team for their help and kindness. A special thanks to Mr. Ted Samuelsson, for spending his time to clarify and answer my questions. I wish them all the best for their future work. I would like to express my thankfulness to Mr. Stephan again, for giving me this opportunity to conduct my thesis work under his supervision. His suggestions and comments has helped a lot to improved the whole work.

I also want to give special thanks to my parents and my in-laws for always believing in me, offering their most caring support and enthusiasm.

Finally and most importantly, I would like to express my gratitude and love to my husband Ajesh and our little daughter Aishwarya. Their company, hugs, and unconditional support have strengthened me through this challenging experience.

Abstract

Cyber Physical Systems (CPS) consist of embedded computers and networks for controlling and monitoring the physical processes associated with it. In practice, these systems are complex in nature. Also most of the CPS systems handle concurrent and real time operations and they are mostly distributed in architecture. Suitable analysis techniques need to be used in order to design and develop a CPS system. Formal verification is a branch of software engineering which is used to verify the correctness of a design. Model checking is one technique used in formal verification, where a formal model of the real system is developed. This model is used for verify the correctness of the system design against its specified requirements. For model checking, a formal model and a formal specification language is required. There are several formalism available to represent concurrent distributed systems such as Petrinets, Actor models etc. Actor models are suitable for modeling the functional behaviour of distributed and asynchronous systems. If formal verification support could be incorporated with an actor model, it shall be beneficial in analyzing cyber physical systems. In this thesis, we use an actor based modelling language which is supported by formal semantics to analyze a complex cyber physical system. This study aims to analyze the distributed nature and concurrent behaviour of the system and not timing constraints. The system selected for this study is a safety critical system which consists of several autonomous machines which are operating in a fleet manner carrying out concurrent operations at a site. We also analyze the distributed design architecture of autonomous machine individually. The design implementation of the system is developed in a Robot Operating System and the actor based modelling language used for developing the models is 'Reactive Objects Language' (Rebeca). Thus major tasks of the study include understanding the ROS based system, transformation of ROS elements to Rebeca models and verification of system specific properties. The study outcomes include identification of few mapping patterns between ROS and Rebeca. During this process a reusable algorithm describing the procedure of transforming any ROS code to Rebeca model was developed. Our study results hence proves the potential of Rebeca in verifying real robotic applications.

Table of Contents

1	Introduction	6
1.1	Thesis Motivation (Problem)	6
1.1.1	Project Definition (Solution)	7
1.2	Research Questions	8
1.3	Research Methodology	9
1.4	Contributions	10
1.5	Report Overview	10
2	Background and Related Works	11
2.1	Distributed Design Architecture of Autonomous Vehicles	11
2.2	Robot Operating System	13
2.3	Formal Methods	14
2.4	Rebeca	17
2.5	Rebeca Model Checker Tools	18
2.5.1	Rebeca Model Checker (RMC)	18
2.5.2	Afra	18
2.6	Related Works	19
2.6.1	Formal verification on Robotic systems	19
2.6.2	System Analysis Using Actor Based Modelling	20
3	ROS to Rebeca Mapping	22
3.1	Understanding HX system ROS source code	22
3.2	Approach to the Model Development	24
3.3	Model Development of HX system design architecture	26
3.4	Evaluation of Properties and their Results	29
3.4.1	Properties of Server Model	29
3.4.2	Verification Results of Server Model Properties	30
3.4.3	Discussion on Server Properties	32
3.4.4	Properties of Machine Model	34
3.4.5	Verification Results of Machine Level Properties	35
3.4.6	Discussion on Machine Properties	36
3.5	Pattern Mapping from ROS to Rebeca	38
3.5.1	ROS Nodes	38
3.5.2	ROS Message Passing	38
3.5.3	ROS Variables	39
3.5.4	ROS Distributed Parameter System	40
3.5.5	ROS member Functions	41
3.5.6	ROS shared Variables	41
3.5.7	ROS Bags	43
3.5.8	ROS Display functions	43
3.5.9	ROS Services	43
3.5.10	ROS Topics	47
3.5.11	ROS Parameter Server	49
3.6	A Pseudo Code for transforming ROS code to Rebeca Model	50
3.6.1	Pseudo code Description	51
4	Electric Site Fleet Management Operation	54
4.1	Understanding Electric Site Case study	54
4.2	Abstracted Fleet Management Prototype	55
4.3	Timed Rebeca Model of Fleet management Prototype	56
4.4	Verification of Fleet Management Model and Results	58

5 Conclusion and Future Work	60
5.1 Summary	60
5.2 Discussion	60
5.3 Future Work	61
5.4 Validity Threat	61
References	65

List of Figures

1	Left figure: Volvo's HX autonomous Hauler, Right figure: Electric Site Fleet Management Prototype	8
2	Research Methodology Used	9
3	A Generalized AGV architecture	12
4	An overview of ROS computation level	14
5	An overview of Verification Techniques	15
6	Model Checking Process	17
7	A typical class definition in Rebeca[1]	19
8	High Level software design architecture	23
9	An overview of procedure followed for mapping process	25
10	Representation of System model	26
11	Server Actor Model Representation	28
12	Actor model for Machine Level Model	28
13	Verification Result of Property 1	31
14	Verification Result of Property 2	31
15	Verification Result of Property 3	32
16	Verification Result of Property 1 against fault Injection model	32
17	Verification Result of Property 2 against fault Injection model	33
18	Verification Result of Property 3 against fault Injection model	33
19	Verification Result of Property 4	35
20	Verification Result of Property 5	36
21	Verification Result of Property 6	36
22	Verification Result of Property 4 against fault Injection model	37
23	Verification Result of Property 5 against fault Injection model	37
24	Verification Result of Property 6 against fault Injection model	38
25	Mapping of ROS Node to Rebeca Actor	39
26	Mapping of ROS Message Passing to Rebeca KnownRebecs & Message Servers . .	40
27	Mapping of ROS local variables to Rebeca statevariables	41
28	Mapping of ROS parameter system to Rebeca Environmental Variable	42
29	Mapping of ROS member function to Rebeca message server	42
30	Pattern Mapping for ROS Service Mechanism	46
31	Pattern Mapping for ROS Publish-Subscribe Mechanism	50
32	An overview of Mapping Patterns between ROS and Rebeca	53
33	An Overview of Fleet Management Model Development	54
34	Fleet Operation Prototype	55
35	Fleet Path for Operation Procedure	56
36	Representation fleet management model	58

1 Introduction

1.1 Thesis Motivation (Problem)

“Cyber-Physical Systems (CPS) are integration of computation with physical processes” [2]. Physical processes can be wireless sensor networks, Internet of Things etc. which requires monitoring and control using embedded computers. Major applications of CPS include traffic control and safety, advanced automotive systems, avionics, critical infrastructure control, distributed robotics and so on. Most of these systems are distributed real time in nature, with integrated sensors and actuators which can manipulate the nature of interactions. Also “Cyber-physical systems by nature will be concurrent” [2]. Concurrent systems have timing dependency which is crucial and any small variation of that can lead to bigger consequences. Also one failure in concurrent systems could be contagious and spread across the system. This could lead to large-scale catastrophic failure triggered even by a small mistake. One typical example for this scenario is the ‘2003 blackout incident’ in North-Eastern United States [3]. Due to an outage of a single transmission element of electricity generation, the parallel transmission paths become overloaded and this overloading cascaded to a condition where electricity production and distribution for the entire city was jeopardized. The source reason of outage in single transmission element can occur due to various unnoticed factors such as aging of equipment, mis-operation of a protective device or even environmental factors. If proper control actions are not taken, such events can cascade easily and can result in catastrophe. Thus, CPS system maintains a higher degree of coordination between physical and computational elements and this factor introduces safety and reliability requirements when compared to general purpose computing. “The economic and societal potential of such systems is vastly greater than what has been realized and major investments are being made worldwide to develop the technology”[2].

For developing CPS, we need to select out a suitable technique to analyze the system and its end-requirements. CPS analysis requires focus on its distributed architecture, real time constraints and concurrent nature. However, this thesis focuses in analyzing and verifying only the distributed and concurrent nature of CPS systems, but not on to their timing or interaction between physical and software components. There are a wide range of analysis techniques such as testing, simulation, assertion check, formal verification methods and model checking. It needs to be assured that the developed system is inline with expected requirements. Anyhow the selected analysis technique shall be able to manage the complex behaviour of CPS system using levels of abstraction. As the complexity increases, reliability assurance requires inexorable testing. Also most of these systems are designed to be operated over a longer period of time under several environmental conditions and thus scope of testing becomes extremely large. Similarly simulation testing are effective at detecting an error quickly and easily, but when it comes to complex systems it is not time effective.[4]. “Formal specification and verification, an alternative approach which guarantees that the requirements or the desired properties are satisfied for any possible execution of the system, have therefore been active areas of research in the distributed system community for more than thirty years.”[5].

Formal verification is a field of software engineering, where correctness of a design or algorithm can be verified against its end requirements or properties using mathematical proofs. Model Checking is one approach in formal verification method. A formal model could help in both qualitative and quantitative analysis of system development without considering its implementation details. Significance of model checking is that, it verifies for system state at least once. There exists several model checking tools such as UPPAAL, SPIN, PRISM, Java Pathfinder etc. which are based on modelling languages such as Timed Automata, PROMELA, PEPA/Plain MC, Java and so on. Actor based modelling is another concept for modelling concurrent systems. This too is a mathematical model, where actors are basic entities of concurrently executing objects that communicate exclusively via asynchronous messages[6]. Actor models have lesser semantic gap between formal verification approaches and real applications. As engineers this modelling approach shall be easier and more convenient to use. Thus, a combination of actor model and formal verification method shall be a suitable technique to analyze a distributed concurrent CPS system. Reactive Objects Language (Rebeca) is an actor based modelling language, which is supported by formal methods

and tools. Rebeca models shall be translated into existing model checker languages using 'Rebeca verifier tool' so as to enable property verification using the models. Modular verification and abstraction techniques are used to reduce the state space and makes it possible to verify complicated reactive system." [7]

1.1.1 Project Definition (Solution)

The goal of this thesis can be defined as an analysis study of a distributed concurrent cyber physical system, using an actor based formal modelling language. We have selected a case study of 'Electric Site Project' with Volvo Construction Equipment. The scope of this project includes developing autonomous machines for material transport for an electrified quarry site. These machines are designed to operate in a fleet manner. Currently these are machine prototypes with a carrying capacity of 15 tons and are completely battery driven. They are designed for loading, unloading and charging in a cyclic manner. The machines are designed to operate autonomously and coordinated by a centralized server control. These machines are termed as 'HX prototype hauler' and they fall under the category of CPS systems. Also they are safety critical systems, since the failure of a system could lead to consequences that are determined to be unacceptable[8]. Similarly the site operation, termed as 'Fleet Management Operation' consists of several concurrent processes and interdependent operations forms a CPS system. Hence the aim of this study is to analyze distributed nature of HX prototype machines and concurrent behaviour of Fleet Management Operation using Rebeca actor modelling language. Fig1 shows HX autonomous hauler and an overview of Fleet Operation at Electric Site.

The design of HX machine software and fleet management prototype are developed in a robotic framework termed as 'Robot Operating System(ROS)'. "ROS is a framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms"[9]. ROS provides all essentials for developing robotic applications as a complete package. Thus, in order to achieve our thesis goal, the ROS code of HX machine design has to be analyzed and it shall be transformed to or represented as Rebeca actor models. Modelling of fleet management prototype is based on the inputs from experts at the organization and it does not require ROS to Rebeca mapping process. Hence a major work in this thesis is to develop a true model of the real electric site system. This process foresees few challenges which are described as follows. Firstly developing an actor model of a ROS based system involves reverse mapping of robotic components to actor models. Mapping of robotic components to actor models involve potential inconsistencies which needs to be solved.

The inconsistencies are due to the differences in framework terminologies between ROS and Rebeca. In order to solve this problem, relevant ROS components have to to be identified and they have to be functionally transformed to corresponding Rebeca representation. Nevertheless, it is expected that components that cannot be potentially transformed from ROS to an actor representation have to be properly abstracted, without abstracting significant functional details. On the other hand, attempting for a complete and direct transformation of the ROS based system to actor model can lead to a condition where the state space generated by model execution grows exponentially, resulting in a scenario termed 'state space explosion', which makes the model checking process unusable. The success of model checking and verification solely depends on the quality of the developed model, whether the model truly represents all the required functionalities and has specified required properties to be verified. Therefore while abstracting the real system, modelling must be done in such a way that it captures crucial features, parameters and constraints of the real system while avoiding a state space explosion.

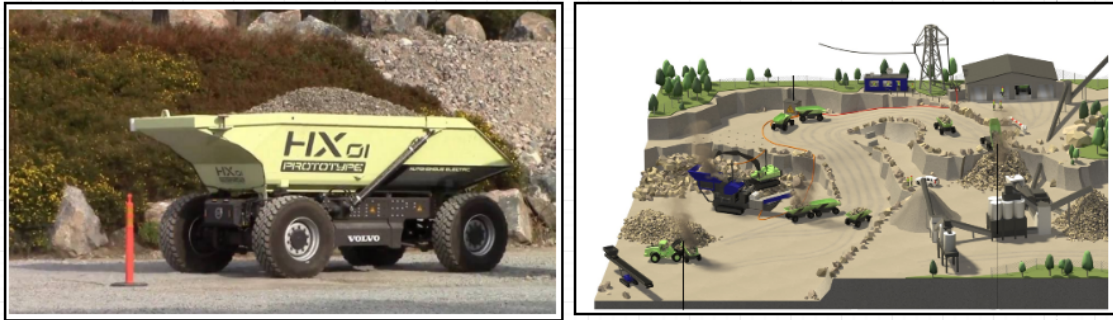


Figure 1: Left figure: Volvo's HX autonomous Hauler, Right figure: Electric Site Fleet Management Prototype

1.2 Research Questions

In this section, we describe the questions that are aimed to be answered by the end of this study. Considering the points presented in 1.1, three questions are defined as the research questions at the start of the study. The questions are defined from different phases of this research, which are listed as below.

RQ1: How can ROS code be mapped to Rebeca model automatically?

This question focuses on the process of automating ROS code transformation to Rebeca model. Since there are foreseen challenges in this process, the aim is to develop a specific algorithm or procedure with which this mapping could be done without much complication. The answer for this question shows a possibility for extending this study to one further level where, ROS code for Electric Site system can be directly mapped to Rebeca model.

RQ2: What are the challenges involved while reverse mapping a robotic system to an actor based formal model?

One of the main challenge involved in developing a formal model of a robotic system is that the systems are complex in architecture. They shall be various tasks of varying priorities within the system. Thus, it is a not an easy task to combine them to a single representation. Also due to numerous state transitions and computations, the developed model can result in a huge state space while verifying it. Such inconsistencies that are encountered while reverse-mapping the robotic source code to an an actor-based model need to be properly handled.

RQ3: What are the properties that can be derived and possibly to be verified using Rebeca models?

The properties are usually certain requirements of the system that have to be satisfied under any conditions. A satisfied property from model checking verification assures that the system might also also satisfy that particular property. Thus, this question focuses to identify relevant and obligatory requirements of the system and get it verified using Rebeca models.

1.3 Research Methodology

This section includes the research method used for addressing the above research questions. This section includes the research method used for addressing the above research questions. We used a multimethodological approach introduced presented by Nunamaker et.al[10] in 1990, as a reference for developing research method suitable for this study. He defines four main phases for this approach **Theory Building**, **System Development**, **Observation** and **Experimentation**. We expanded each of the phase in such a way that back iterations are allowed within every phase. Figure 2 shows step by step details of research methodology used in this thesis work.

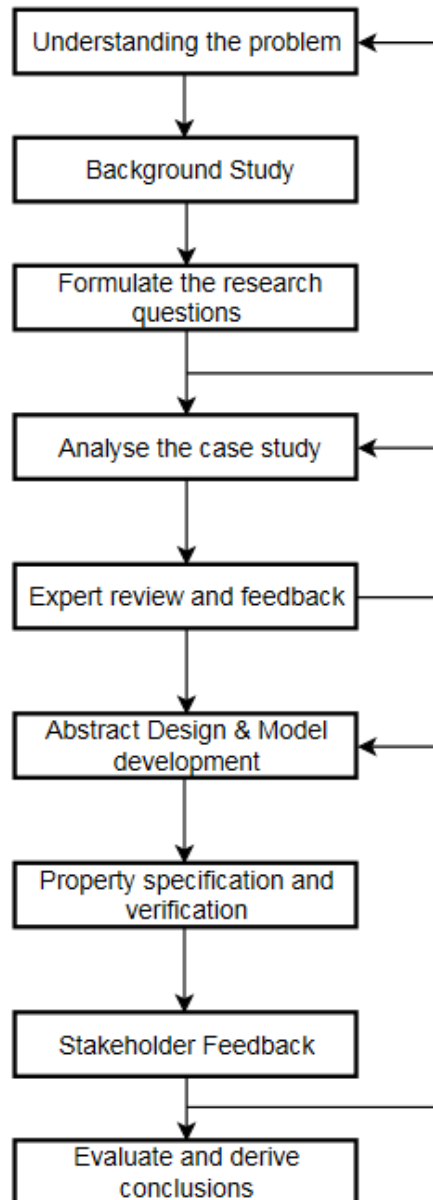


Figure 2: Research Methodology Used

The initial step was to conduct understand the problem. This was done by conducting a background study related to the context. It was then followed by defining research questions, which helped to shape up the thesis. This set of process was iterated and more clarity was

obtained in understanding the thesis. Based on the above knowledge and with the reference of selected case study, an initial analysis was made with respect to certain include and exclude criteria. The analysis outcomes were used for developing abstract system design and further to develop the models. Design system requirements are specified as properties and they are verified against the system models. Now based on the feedback and review comments from stakeholders, the system development phase was iterated with several rounds of experimentation. This helped to fine tuning of verification process. Lastly the models are evaluate and conclusions are derived out of it. The results are documented in the report.

1.4 Contributions

This section presents major contributions of this thesis study, in solving the problem described in Section 1.1. The major contributions out from this study are as listed below:

- **Mapping Patterns from ROS to Rebeca**

A major part of this thesis is to transform ROS code to Rebeca models and to verify system specific properties using the developed models. This process resulted in connecting Rebeca semantics to corresponding ROS terminologies. Thus, a list of mapping patterns between ROS and Rebeca were derived at the end of the study.

- **An algorithm for automating ROS to Rebeca Mapping Process**

While deriving the mapping patterns between ROS and Rebeca, we also identified a standard procedure to develop these mappings. We structured the procedure as an algorithm which can be applied on any ROS code for transforming to Rebeca model.

- **Identification of relevant properties and their verification**

This is a stand-alone contribution for this particular case study. We identified certain system specific properties which were identified while understanding the system design and also found to be relevant properties while referred to other similar studies.

1.5 Report Overview

An overview of this report structure can be explained as below:

- Chapter 2 includes 'Background' information and 'Related Works' for the thesis. This chapter gives a prerequisite knowledge about the topics dealt with the study and state of the art for this title.
- Chapter 3 presents 'ROS to Rebeca Mapping Process'. This chapter includes description on HX hauler case study, analysis of ROS code, Model development of HX system and property evaluation. This chapter also includes a list of identified 'Mapping Patterns from ROS to Rebeca' and an algorithm for the automatic mapping of any ROS code to Rebeca models.
- Chapter 4 presents 'Fleet Management Operation' prototype description and model development. This chapter also includes model checking of properties of this prototype model and their results.
- Chapter 5 is the final section in this report under the title 'Conclusion' and it includes a summary, discussion and possible future works for this thesis.

2 Background and Related Works

In this section, we present an essence of background study conducted during the initial phase of research. Since this study touches a number of areas, such as Formal verification, actor based modelling, ROS framework etc. this section is divided into short subsection. A brief explanation required to understand each of this topics are included under each subsection. The state-of-art of this topic is also included at the end of this section.

2.1 Distributed Design Architecture of Autonomous Vehicles

The systems that we are dealing within this study are autonomous vehicles/machines which have a distributed design architectures. Background studies show that, this design architectures are used for most of the automated vehicles which are termed as 'Automated Guided vehicles(AGVs)'. "AGVs are cyber physical systems, with autonomous interacting units equipped with sensors and actuators to interact with their environment and to allow them to be intelligent, cooperative, and communicative"[11]. These systems has sensors for perceiving its environment conditions and this information is used to compute and take decisions for the action to be taken and finally responds using actuators. They are generally used for industrial applications such as materials handling systems, manufacturing or construction systems and so forth[12]. In most of the cases, they shall be operating in a fleet manner where several AGVs operates together by interacting with each other[12]. In this case, they are not 100% autonomous, there exist a central control system which helps to coordinate between them and to supervise the entire process [12]. This refers to a distributed architecture of AGV systems. A wireless LAN could be used in order to provide uninterrupted communication within the AGV system's distributed software. Also the prime focus while designing AGV systems would be to develop a control system which enables path routing, collision free navigation, obstacle avoidance, battery management, no deadline misses and no deadlock conditions [13]. Several algorithms have been developed for motion planning strategies for these machines to cope with open environments and to reach final destination. In addition to this reliability and safety has to be ensured while designing such systems, as any negligence can lead to bigger consequences. David González et.al states that "Despite of some remarkable results obtained up to now, there is still a long way to go before having fully automated vehicles on public roads, including technical and legal unsolved challenges" [14].

In practice, software design used for any AGV design architecture is distributed in nature[15]. A overview of general AGV control system is shown in Figure 3[12]. In this architecture, a supervisory control is located at the server side and it schedules job priorities and assigns individual machine tasks. Also a global route planner routes path segments for all the machines within the operating site. Now this paths segments shall be distributed to every machines in the fleet. At machine level, individual control system is located for each AGV. It consist of sensors which shall perceive environmental data and helps in self localizing the machines. This helps in avoiding obstacles and local paths shall be calculated and this information is sent to actuator controls. Each request driven from server shall be distributed as internal processes for each machine. Therefore in AGV system it is required to have an internal co ordination as well. This helps to track vehicles each other, there by avoiding collisions between them and also to avoid any deadlock condition. This structure of distributed software control system is the base for any AGV with modifications as per their application and they provides flexibility, space utilization, safety and operational cost efficiency. Industries demands for flexible, cost effective and less maintenance demanding solutions[12]. This can be achieved only with a distributed system, however it increases the complexity[12]. Thus, it is a challenge to find the right balance[12].

Moving on to design algorithms, there exists several navigation strategies and routing algorithms for AGVs. AGVs can be considered as a 'Multi Agent System', where several agents are closely coupled to work together[15]. "The main functionalities that an AGV transport system has to fulfill is assigning incoming transport tasks to appropriate AGVs, routing the AGVs through the warehouse efficiently while avoiding collisions and deadlocks, and maintaining the AGVs batteries" [16]. Thus, the major focuses in AGV system design and development are selection of right routing

algorithms, localization techniques, obstacle avoidance strategies and ensuring a collision free navigation. The navigatory tracks are usually predefined and is used for global path planning at server end. However, based on sensory inputs and perceived environmental information, local path has to be computed and updated frequently. Kalman Filter Localization algorithm (KFL)/ Extended KFL algorithm or Markov algorithm and Monte Carlo algorithms etc. are the most preferred ones for computation of self localization in AGVs. Similarly path planning algorithms such as Dijkstra's algorithm, state lattices or Adjacent matrix method are quite popular ones to this end [14].

Being said that, AGVs are still associated with challenges when it comes to real time planning computations[14]. "The limited time for generating a new free collision trajectory with multiple dynamic obstacles is an unsolved challenge" [14]. Even though it is less expensive to use distributed control algorithms than a high performance computer, they are time consuming and are at a risk of not meeting with timing constraints. Also there exists certain ambiguity in perception and control constraints. Researches has been conducted in this area to properly handle ambiguities with perceived information and thus to develop a better real time environment knowledge[17]. One such study is presented by Gianluca Antonini and Michel Beierlaire in 2005[17]. More recent development involves human factor, a driver who can interact with the system through Human Machine Interface (HMI)[18]. However it creates newer challenges such as sensor fusion uncertainties, driver knowledge etc. for generating safety navigation.

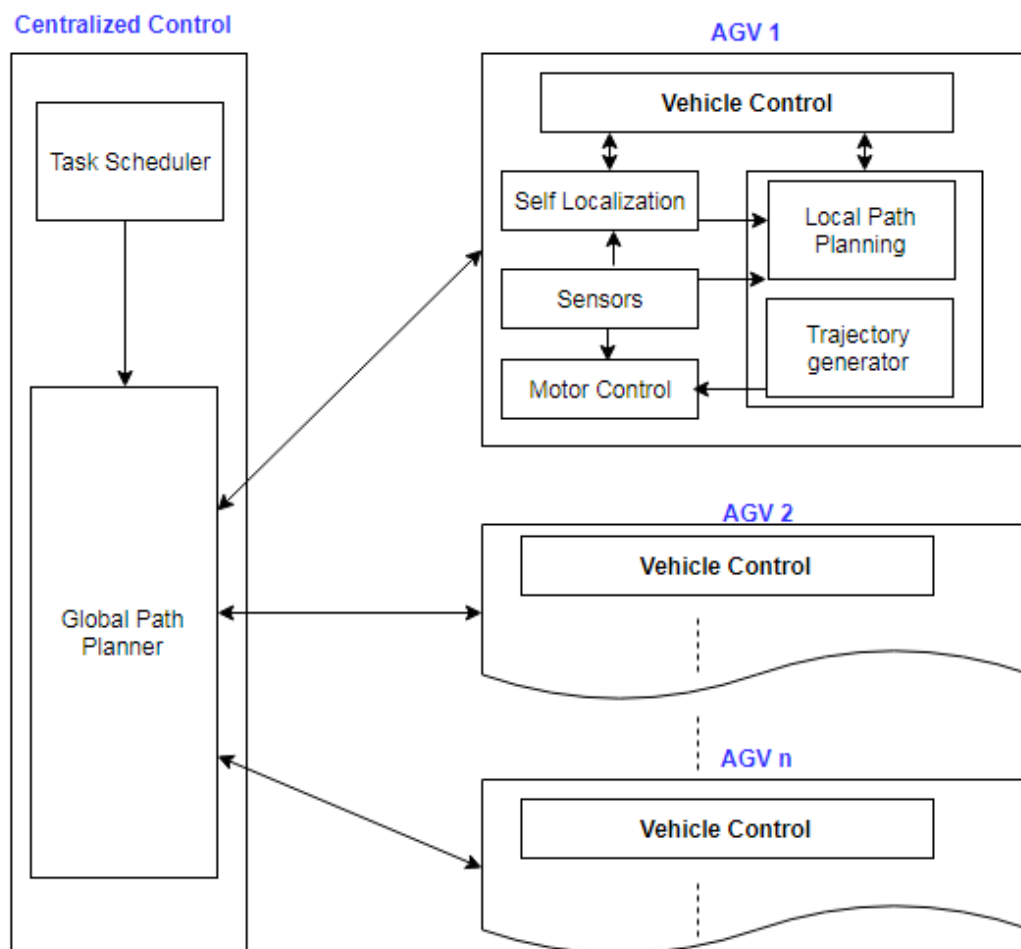


Figure 3: A Generalized AGV architecture

2.2 Robot Operating System

Robot Operating System (ROS) is a licensed, open source middle-ware framework for developing robotic applications and is becoming a de facto standard in this area[19]. ROS allows service oriented communication between the network components. It also supports hardware abstraction, device drivers, libraries, visualizers, message-passing, package management etc. and thus aids a developer with low level coding requirements for developing robotic systems[19]. Furthermore, end to end software debugging is supported in ROS with the full source code of ROS being publicly available. This acts as a complete reference for the developers and it is a community where they too can contribute[20]. ROS framework development was started with STanford AI Robot (STAIR) project. and later in 2007 further established at Willow Garage as open source distributed software under Berkeley Standard Distribution License (BSD). Recently ROS infrastructure management was taken over by Source Robotics Foundation 6 (OSRF)[20]. ROS is not a real time operating system, but with its rich features and multilingual programming convenience made its popularity to grow tremendously and it has integrated with numerous development platforms such as Matlab, Arduino, Android, Labview etc[21].

The fundamental processing units in ROS are called 'Nodes'. A node is designed to perform specific tasks such as publishing data, acquisition of data, data computation etc[9]. Node names shall be unique and all the nodes are registered to a 'Master node', which keeps track of all the nodes in the network. This helps to establish peer to peer communication among nodes and helps to find each other. Nodes, libraries, configuration files and all other dependent element for a function forms a ROS package[9]. All data communication occurs in form of **ROS messages**. Thus, a ROS package structures the network functionalities in a ROS system. There are several ways by which communication occurs in ROS system, majorly via **Topics** and **Services**. Topics are means of communication by which message transfer occurs by a *publish-subscribe scheme*.

Communication through topics is the most basic protocol used in ROS framework[9]. Topics are unique channels, designed for unidirectional data flow through which nodes exchange ROS messages. They have special semantics which separates the source information from its reception. A 'Topic' can be considered as a named channel or bus. A node which is sending out messages publishes it to the topic. Similarly the node which receives the message has to subscribe to that particular topic in order to access the published data[9]. One topic can be used by several nodes to publish their data, also multiple subscribers can access the same topic as well. On the other hand services offer communication through *request-reply scheme*. That means a node can offer or serve services under a specific service name and the client node can avail the service by sending a request. As soon as the service is available the server node shall reply to the client node.

Services is defined by a pair of messages: one for request and other for reply. Another communication form is called **ROS Bags** of *store-retrieve scheme* and it uses a remote procedure call (RPC) protocol which runs inside ROS master[9]. This means that the functions or procedures used by these parameter servers are accessible through normal RPC libraries. It shall be in a shared location accessible via network. Information can be stored during run-time, and it is later retrieved to playback. For example, sensor data or continuous key inputs can be stored when it is processing and later to check its behaviour, it can be retrieved and analyzed. **Parameter Server** in ROS can store data of various data types and it helps to manipulate required parameters for a ROS system[9]. Figure 4 shows an overview of ROS computational level.

Despite of numerous possibilities and advantages of using ROS framework, there is no widely accepted solution to review or verify ROS operating system. However, several approaches like run time monitoring, static analysis techniques, model checking etc. exist have been proposed to achieve this[22]. Colin Angle, co-founder and CEO of iRobot states that unless it is made stable and secure, ROS cannot be used for critical solutions such as military, space and security[22]. In addition to this, lack of authentication procedure for communication between master and node makes it vulnerable to attack just by matching environmental variables with IP address and port number of master[23]. Similarly ROS Bags can be hacked by intercepting the Master node message and could

be saved to bag files. Also XMLRPC communication used in ROS can be easily intervened by attackers since there is no encryption scheme for it. By registering a service with same name, hackers can intercept the control of previous service[23]. Countermeasures for all the above drawbacks are available, however their assurance level is not officially accepted by industries.

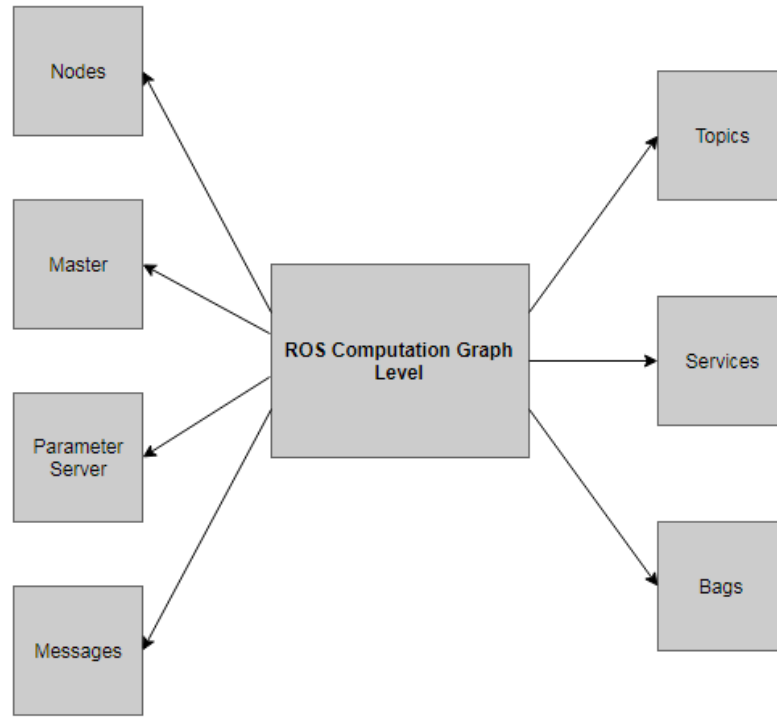


Figure 4: An overview of ROS computation level

Nevertheless, companies like Yujin Robotics in Korea is using ROS to develop robots for their research and academic fields[22]. Similar is the case with newer companies like Rodney Brooks, Heartland Robotics, Yaskawa Motoman, Adept Technology etc. These initiatives show that there is a growing trend of ROS being used in the academic community as well as for industrial robotic applications. However, the availability of a freeware alternative is an associated challenge for these companies. Designing the robotic framework and varying response with respect to applications is another challenge, which is equally important to solving technical problems in robotics. Despite these challenges, using ROS for robotic development can bring up 'software standardization', as it requires only a low level of coding competence. According to Helen Greiner, another founder of iRobot, such software standardization can help researcher engineers and smaller organizations to release their products without much re-invention of all the required technologies[22].

2.3 Formal Methods

"Formal methods are mathematical techniques, often supported by tools, for developing software and hardware systems"[24]. They can be used for analysis and verification at any part of the program life-cycle and are often supported by tools. J. Woodcock et al. has cited that "Formal method tools can provide automated support needed for checking completeness, traceability, verifiability, and re-usability and for supporting requirements evolution, diverse viewpoints, and inconsistency management"[24]. "Formal methods have been recognized as a critical, and often expected, design-time component for autonomous and life-critical systems such as aircraft and spacecraft"[25]. Formal methods can be categorized into **Formal Specification** and **Formal verification**[26]. Formal Specification is used for describing or representing a system and its desired properties formally, whereas Formal verification involves analyzing the system for the specified properties. Formal verification can be defined as a field in software engineering, which uses mathematical basics to

prove the correctness of a design by verifying against its formal specifications[27]. Since they are supported mathematically, the results provides a confidence in the validity for design correctness.

Formal verification (FV) methods have been used for analysis and verification of complex and safety critical systems such as air traffic control, defense etc [28]. These methods are based on discrete mathematics and computer aided tools in order to describe and analyze properties of hardware and software systems. During system development, a higher percentage of total effort is invested for verification and debugging of the development process[29]. This verification process becomes expensive and exhaustive as the complexity of the system increases. Verification can be classified into various categories based on the techniques used. Figure 5 shows an overview of verification methods available[30]. **Simulation** is a sophisticated and easy to understand technique used for system verification. They are usually on-spot on detecting any error, however simulation process are time consuming, it consumes a lot of resources and also it is vulnerable to human error[29]. Emulation is the process of recreating system design using a special purpose emulation system. This is also not a cost effective solution for verifying complex systems.

The final alternative is formal verification, where issues related to design ambiguities are verified by automated and exhaustive system space exploration. Also FV techniques can be applied even at the early stages of development, thus it is cost effective and less time consuming. This is an active area of research in the distributed community for more than 30 years [31]. FV approach includes two factors: 1.) A formal specification for describing the system and its requirements 2.) Analysis method to verify whether the system specifications meets the specified requirements. In order to formally represent the system and its requirements, certain formalism standards has to be used. The most popular ones to this end are **Temporal Logics, Propositional Logic or Boolean Logic, Predicate Logic** and so on. Formal methods can be arithmetic verification, property checking or equivalence checking as depicted in Figure 5. As described in Section 1.1.1, we are focusing on Model Checking techniques for this study, hence analyzing or understanding about other formal method techniques are out of scope for this thesis.

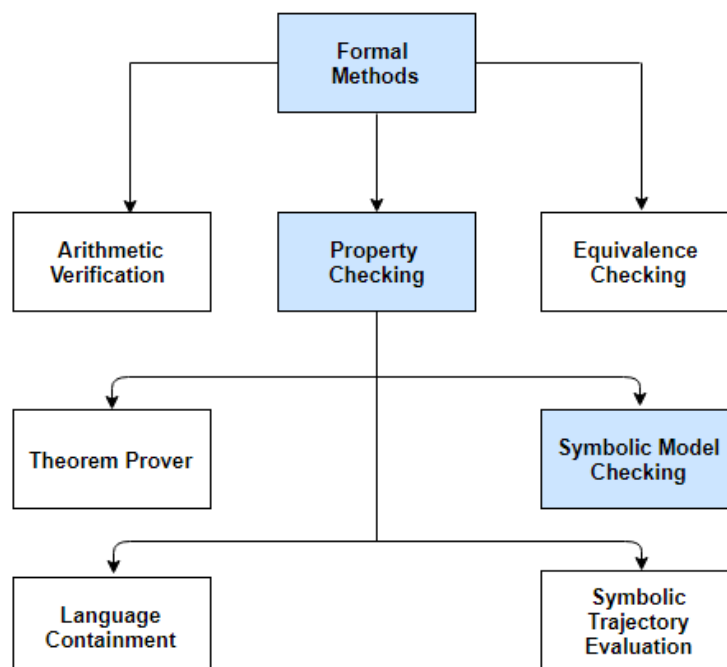


Figure 5: An overview of Verification Techniques

Model checking is a widely used technique for formal verification of distributed systems. It was developed in early 1980's by Clarke and Emerson and by Queille and Sifakis[25]. Since it is automated, verification of even complex systems can be done in lesser period time when compared to other formal verification techniques. It works by effectively examining the complete reachable state space of a model in order to determine whether the system satisfies its requirements or desired properties"[31]. Thus, Model checking is a process of verifying whether the system conforms with its specified requirements. As the properties are verified as satisfied, model checking iteration goes on and finally a state space representing all possible state transitions of the system shall be generated. On the other hand, if the any of the property is violated, the model checking process stops and a counter example showing the path of property violation is generated. This process is given in Figure 6. By analyzing this counter example, we can identify which transition of the system state resulted in the failure. Also model checking performs an exhaustive searching of all states in a systematic way, thus an error captured during model checking is always a real error. However, when it comes to complex systems the state space generated can be really huge. This is because when various component states are combined, number of states grows exponentially. ie. a system with 'm' states and 'n' variables, shall result in $m * (2(\exp)n)$ states. This is a risk for 'state space explosion' problem, which is the greatest disadvantage with any model checking method[32].

"Temporal logics are modal logics with special operators for time". Time can either be interpreted to be linear or branching. The most common logics are the linear time logic *Linear Temporal Logic* (LTL), and the branching time logic *Computation Tree Logic* (CTL)[33]. Formal specification of system requirements requires a representation language and an associated tool for verification. Linear Temporal Logic (LTL) was introduced by Amir Pnueli in 1977, to describe the behaviour of a system by a single formula[31]. Property specification in model checking using temporal logic supports to specify various concurrency properties. Using Temporal Logic, system properties shall be specified. 'Safety' and 'Liveness' are two main properties which shall be verified using any model checking tools. Safety properties verify that there exists no state with undesired behaviour and liveness properties verifies that there exists at least one state where desired behaviour eventually exists [32]. Using temporal logic, these properties can be given as below:

If p is a property in system S , LTL formulas can be specified as below:

$$G(p) \tag{1}$$

$$F(p) \tag{2}$$

Equation 1 states that property p is globally true in the system. Equation 2 states that property p is finally true in some state of the system.

Similarly, if p and q are two properties in the system, we can specify another LTL formula as below:

$$G(p \rightarrow Fq) \tag{3}$$

Equation 3 states that whenever property p holds in the system, property q will be finally true.

There exists several model checking tools such as UPPAAL, SPIN, Java Pathfinder and so on which emphasizes on the modeling of process synchronization and coordination, not computation and is also not meant to be analyzed manually[30]. Various model checkers are based on various programming languages and each of them are developed for various modelling strategies. For example, UPPAAL is a model checker tool used for modelling real time systems as networks of timed automata with extended data types[34]. It uses finite control structure and real value clocks for communicating through defined channels and shared variables [34]. SPIN is another model checker used for the development and verification of concurrent and distributed systems[35]. In this thesis we use an actor model, Rebeca for modelling a distributed and concurrent AGV system. We have included a brief description about rebeca, its background and its associated model checking tools.

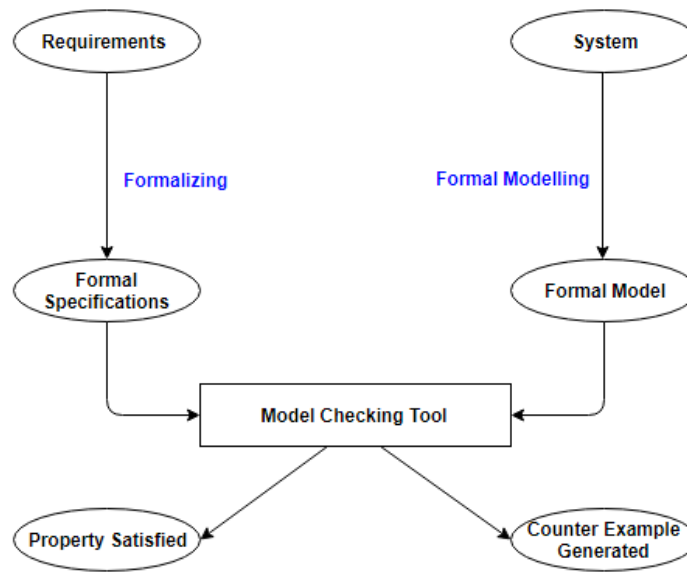


Figure 6: Model Checking Process

2.4 Rebeca

“Rebeca is an actor-based language with a formal foundation, designed in an effort to bridge the gap between formal verification approaches and real applications”[7]. It is used for modelling concurrent applications based on an operational interpretation of the actor model. An actor model is a programming standard for developing reliable and maintainable concurrent software. They represents dependencies between various network ties. In actor modelling, an actor is the basic entity who communicates with each other via messages. These messages need not be processed immediately when received . They can be stored unless the receiver is ready to process it. Thus, an actor can manipulate its own state, change its own behavior, send messages to other actors and spawn more actors. In other words, actors are distributed, autonomous objects that interact via asynchronous message passing[36]. The flexibility of abstracting complexities makes actor model suitable for several concurrent applications. Examples of these applications include designing embedded systems, wireless sensor networks, multi-core programming and designing web services[36]. Analysis of real-time systems using actor model is not in use even though actor models gained popularity in analyzing distributed and concurrent systems. Thus, a few timed actor-based modeling languages were introduced such as RT-synchronizer and Timed Rebeca[37].

Rebeca is an actor based modelling language which is designed to enable formal verification of actor models[36]. Rebeca provides a model-driven development approach with a formal basis. It bridges the gap between software engineers and formal method community. Rebeca modelling is used for those applications which involves event-driven systems with asynchronous message passing. Rebeca allows non-blocking communication and non-preemptive function execution. It support neither explicitly receiving messages nor shared variable mechanism. Potential of Rebeca has been extended by researchers and it has been proposed to provide the ability of modeling and verification of distributed systems with real-time constraints. For this a concept of Floating Time Transition System (FTTS) which significantly reduce the state space generated when model checking with Timed Rebeca models[37]. Using Timed Rebeca, verification for deadlock freedom and schedulability analysis can be performed.

“Modular verification and abstraction techniques are used to reduce the state space and make

it possible to verify complicated reactive systems”[38]. Various techniques are used in abstracting a complex real time system, while preserving its property specification using temporal logic. This helps in reducing the state space of model which makes it more suitable for model checking process. Rebeca looks very similar to a high level programming language like Java even though it is a modelling language with formal semantics and formal verification support. In Rebeca, communication takes place by asynchronous message passing between Rebecs and execution of the corresponding message servers. Each message is put in the queue of the receiver rebec and specifies a unique method to be invoked when the message is serviced. Rebeca communications are event driven, where the messages that are sent among actors are considered as events. Each rebec (actor) picks a message (event) from its message queue (memory) and executes the corresponding message server (function/tasks). This execution takes place in a non preemptive manner, that means priorities on tasks cannot be set. Thus, communication of the model is achieved by asynchronous message passing which is non blocking for both sender and receiver[38].

The message is placed in the message queue of corresponding rebec and it stays there until it is served by the receiver rebec. This queue length shall be defined by the model developer based on the model design. Timed Rebeca which is a timed extension of rebeca is a high-level actor-based language capable of representing functionality and timing behaviors at an abstract level. The timing constraints are specified in Rebeca using three keywords: delay, after and deadline. The keyword 'delay' represents the passing of time for one actor. Keyword 'after' is used whenever a message has to be delivered 'after x timeunits'. The Keyword 'deadline' indicates that if the message is not served by receiver within specified deadline time, it shall get purged. In Figure 7, a basic syntax and structure for a reactive class is shown.

In first line, the numeral given in brackets next to rebec name is the message queue size which is defined by the model developer. It is defined that a pure actor model has unbounded queue length in theory, but in implementation queue size has to be defined. Second line is the syntax for defining the other rebecs/actors to be communicated by this rebec. Third line holds state variables for this reactive class. It is used to represent the current state of the actor. The state variables can be of int, byte, short, boolean types. Further this rebec shall be initialized. Various message servers has to be defined which are the functions to be executed. Message servers can have input parameters which can be of different types. Methods are local to the corresponding rebec and it can be only be called by the message servers or other methods inside this rebec. A method can return value whenever it is called and it can send a message to only the rebec containing it. This rebec is accessible by using keyword 'self', which is the reference to a rebec.

2.5 Rebeca Model Checker Tools

2.5.1 Rebeca Model Checker (RMC)

Rebeca language provides model checker tools which acts as the front-end to translate the codes into existing model-checker languages. Thus, it enables to verify their properties[1]. Modere was the first direct model checker of Rebeca which was developed in 2005. Modere performed LTL model checking on Rebeca models[1]. Later RMC was developed for direct model checking of Rebeca models without using back-end model checkers. With this development, properties could be specified based on the state variables of rebecs. RMC evolved with newer versions to make it more advanced, stable and reusable. This was achieved by decoupling model checker algorithm, state space storage mechanism and input model translator. RMC can be considered as a compiler which translates the input Rebeca model to a set of C++ files. These generated C++ files are compiled to an executable file. This file execution applies the model checking algorithm and generates the verification result. Verification result shall include the generated state space of the model which is saved in an XML format and termed as 'statespace.xml'

2.5.2 Afra

Afra model checker was developed to integrate several modelling environments including Rebeca and SystemC. It used reduction techniques to tackle the state explosion problem, where ever ap-

```

reactiveclass Rebec1(2) {
  knownrebecs { Rebec2 d;}
  statevars{}
  Rebec1()
  { self.msg1();
    /* the constructor */}
  msgsrv msg1()
  { d.askForService();}
  msgsrv msg2()
  { /* Handling message 2*/}
  void method1(int param1)
  { /* method definition */}
  int method2()
  { /* method definition */
    return intValue;}
}

```

Figure 7: A typical class definition in Rebeca[1]

plicable. System C models are the inputs to Afra along with its LTL or CTL properties and the the model shall be verified. Like any other model checking tools, if any of the specified property is not satisfied, a counter -example is displayed. This counter example shows a trace of system state transition where the property was violated. In case of System C models, Afra translates SystemC codes to Rebeca. It then utilizes the Rebeca verification tool set to verify the given properties. Afra uses the concept of Floating Time Transition System (FTTS) for the analysis of Timed Rebeca models[39]. FTTS can significantly reduce the state space of the model to be explored. Here the focus is on event-based properties and but not the constraints. The current version of Afra model checker available is 'Afra 3.0' and this is used in this study. Afra needs Java Runtime Environment (JRE) version 1.5 or higher. Afra needs a C/C++ compiler for model-checking tools and for compiling SystemC sources. In windows an open-source C/C++ compiler, the GNU C Compiler, in Cygwin package bundle needs to be used.

2.6 Related Works

2.6.1 Formal verification on Robotic systems

Raju et al. presented a work on formal verification of a ROS based robotic application using timed automata modelling language in [40]. In this work, they proposes an approach to model and verify the communication between nodes in ROS system. A case study on a physical robot, Kobuki, has been focused and its selected properties are verified using UPPAAL model checker. This work is closely similar to our thesis even though the modelling language and application is different. Conclusively the study conducted in [40], presents a formal representation of selected ROS based application and selected properties verification using the model checker. Followed by formal model development based on the case study and verification of its safety and liveness properties is conducted. Their proposed model could find parameters to validate properties such as continuous availability of sensor messages, continuous motion status etc.

Tichakorn Wongpiromsarn and Richard M. Murray presented a study on formal verification of an autonomous vehicle system, Alice, in [41]. The case study and approach presented in this paper is very similar to our work. The distributed nature of the system under their case study is exploited in systematic way that the entire system level requirements has been decomposed into a set of component level requirements. This helps to model and verify without much complexity.

State consistency between different software system modules and safety and liveness properties have been verified in this study. Even though this study is related to our thesis in terms of using formal verification methods to analyze an AGV system, they use a combination of model checking and theorem proving methods whereas we are focused on actor based model checking with formal method support. Also they use CTL and LTL statements for property specification, whereas since Afra supports only LTL statements, we use LTL statements for property specification.

A formal verification approach has been presented in [42], where industrial robotic system properties have been verified using a model checker tool called 'SPIN'. However, these robots are not built on ROS framework. The three properties which have been looked at are deadlock detection, non collision and kill-switch violations (an emergency stop situation depending on variable value which is checked periodically). This paper presents mapping of robotic systems into PROMELA models and verifying them using SPIN model checker. This is similar to our work where robotic system is mapped to Rebeca models and verified using Afra tool. However as the robotic systems cannot be directly mapped into the modelling language used for the study, a compiler optimization technique is used for closing the semantic gap.

Webster et al. proposed a formal verification of ROS-based autonomous robotic assistant "Care-O-bot"[43]. This work is closely related to our thesis study for three similarities 1) ROS based autonomous robot 2) formal verification 3) model checking. Also a non deterministic approach is used in the model that can cover a larger set of user activities. Similar to [42], this work uses PROMELA modelling language with SPIN model checker. This work was much specific to the robot and it focused on verifying properties on high level decision making rules. Hence they are proposing a proof of concept showing the assurance of their model in representing the real system and the level up to which it is conformed to the system requirements.

In [44] model development specification and model checking of multi robotic system are presented. The use of formal methods in safety critical applications are presented in this study. Similarly problem scenarios which could be solved using their formal specification and verification is depicted in this paper. The work done in this paper is not directly related to our thesis study, however it provides an overview about formal verification procedure in robotic framework. Cowley and Taylor proposed a static verification of robot behaviour in [45]. They propose a static and formal approach of safety requirements as well as a self analysis of expected states at every sequence of actions on the system in this study. However the modelling approach used in this work is based on finite state systems and timed automata whereas the model checker used in this thesis is based on floating time transition system (FTTS).

2.6.2 System Analysis Using Actor Based Modelling

In [46], Khamespanah et al. presents schedulability analysis of a distributed wireless sensor and actuator network (WSAN) using actor based model checking. They developed models to find an optimized schedule to use resources while satisfying timing constraints. The WSAN is represented as a collection of actors. Their abstraction and composition properties are used to build a true model of WSAN behaviour which is extracted from node level. A comparison of this approach with traditional informal analysis is based on assumptions and measurements, trial and error etc. But each of these faced several practical difficulties because of the complexity of concurrent and distributed nature of WSAN systems. With their proposed approach and solution, the authors could assess the performance and functional behaviour of the system throughout the design and implementation phases. Also the developed models helped to determine the parameter values, which lead to highest system efficiency. In addition, by using the actor based modelling approach, the size of the system state space could be reduced by 50 to 90%. This study is related to our work as they also use a case study on a distributed concurrent system, use of Rebeca modelling language and Afra model checker tool. However it does not involve a mapping process between a robotic framework to actor model checking.

Brynjar Magnusson et al. presented a work in [47] on Event based Analysis of Timed Rebeca

Models using SQL. In this work, authors presented an approach of actor based model analysis using simulation and event based property specifications. From simulator end, occurrences of events are stored in a relational database and the model checking is performed against it. In any model, the reactive behaviour of the system is modelled with asynchronous communication. Here each actor can make local decision, send messages and determine how to respond to the next message. The major outcomes from this work were implementation of a tool set to simulate the developed TR models, development of an event based property language for TR models, a tool set to map formal models to SQL queries and a tool for analyzing executing SQL queries results. However, the authors have realized the necessity of finite simulation traces of TR models in order to perform simulation based analysis. Hence every simulation has to be stopped periodically to get finite simulation traces, whilst the optimum point to stop the simulation solely depends on the model and its properties. This work is related to our work in such a way that offline run-time verification has been focused as the second major contribution in our work. Also the concurrent behaviour of the elements has been modelled using non deterministic choices. However similar to the previous one, this work also does not involve any robotic framework modelling and its verification.

In [48], Khamespanah et al. presents modelling and analyzing of distributed and asynchronous systems using Rebeca actor based language. They states that actor model is a well established paradigm to model the functional behaviour of distributed and asynchronous systems. This work presented the semantics for a formal presentation which is approachable and user friendly. Also a reference for any implementation effort. In this study, a case study of 'Network on Chip (NoC)' architecture has been modelled and analyzed using the above developed modelling language. They verified desired reachability properties and no-deadline-miss property of the developed models in this work. They also have used parallel composition approach for Rebeca models for faster model checking. Here the approach is to map each component of the system in a 'Probabilistic Timed Rebeca' model to a Probabilistic Timed Automaton (PTA). This is followed by a parallel composition of all the developed PTAs. PRISM model checker is then used as model checker for the PTAs and to verify its probabilistic properties. The experimental approaches used in this paper and its object oriented implementation has been referred to develop models for our study.

Jafari et al. conducted a study[49] to verify safety properties of larger models using statistical model checking. The authors provide analysis technique and its tool-set for verifying functional correctness and performance evaluation of real time distributed systems using asynchronous message passing. Multiple simulations of the system is executed and the mean value of model correctness is computed for a specified property to be verified. For performance evaluation, the mean response time to compute performance measures of the model is computed. This study uses several case studies to experiment with verification of several safety properties. They could conclude that the efficiency and applicability of statistical model checking approach depends only on the size of our models. Thus, by increasing the number of rebecs/actors and the message passing between them, the approach can be applied for more complicated system. This study is related to our work in this regard. The modelling techniques has been referred from this work. The inspiration to perform reconfiguration analysis on the operational behaviour mentioned in our case study was also reinforced from this work.

3 ROS to Rebeca Mapping

In this section, we describe the entire process of ROS to Rebeca mapping using the HX machine case study approach. The distributed nature of HX machine's software design is analyzed and transformed to Rebeca model in order to verify the specified properties. This process consists of several steps which can be listed below. In this section, each of the below steps are described as subsections.

- Knowing the HX machine architecture by understanding the ROS source code
- Developing Rebeca Models by abstracting relevant details from the ROS source code
- Identifying relevant and beneficial system properties and their verification
- Deriving mapping patterns between ROS to Rebeca
- A reusable algorithm to transform from any ROS code to Rebeca model

3.1 Understanding HX system ROS source code

Understanding a ROS source code requires prerequisite knowledge in ROS framework terminologies and structure. Thus, getting familiar with ROS programming is the first task in understanding ROS source code. Since ROS provides inbuilt provisions for hardware, device drivers, libraries, visualizers, message-passing, package management etc. and with basic ROS knowledge, a code walk through was done to understand the HX design implemented in ROS. From the high level design diagram, as given in Figure 8, an overview of design architecture is obtained. While creating a high level design diagram, developers have exploited the distributed architecture of HX system in such a way that they are modularized into **1.) Server Module** and **2.) Machine Module**. In Figure 8 section with gray background represents machine module and section with white background represents server module. This modularization is made only for better understanding and representation of the design architecture.

In real system, Server Module acts as the centralized control for the fleet operation and coordination among the vehicles and environment where as Machine Module guides the vehicle to navigate autonomously by perceiving sensor information and reacting through actuators. ROS code executes the system by making the components of these modules to communicate and coordinate with each other. An overview of design logic can be described as follows: Fleet Control located at server coordinates the fleet of HX machines while operating. It has two mandatory inputs of current wheel loader position and a global path planner. Fleet Control assigns machine missions and task for individual HX machines in the fleet and track their status. On machine side, sensor data is perceived and is used for computing an obstacle free path and it also coordinates with traffic control component located at server side. HX machines are autonomously navigating vehicles as well as they can be controlled by server side in case of collision possibility.

The basic processing unit of ROS framework is termed as 'Node', thus by identifying the nodes we figured out the total processing units in the node which are executable and communicating with other nodes. We identified most of the components from design diagram in the ROS code structure. however, few components were not completed implemented at the time of conducting this study. Their functionalities were discussed with the developers for understanding the complete picture of HX system design. The nodes are **Fleet Control**, **Machine Control**, **Traffic Control**, **Path Server**, **Server Obstacle Detection**, **Motion Control** and **Perception**. All these nodes are initialized and defined with variables and member functions in order to carry out specific tasks. By going through the member functions of each of these nodes, their functionalities can be derived. However, these functions and other member classes defined in the nodes are loaded with implementation logic and details. We pay less attention to these data and the focus was to look out for relevant functionalities and communication mechanism. As explained in Section 2.2, major communication mechanism used for ROS systems are via **1.) Topics** and **2. Services** The major services used are *MapStructure.srv*, *Path.srv*, *MachineMissionEta.srv*, *MachineMission.srv*,

LoadComplete.srv, *FleetMission.srv* and major topics used are for publishing the following messages: *FleetDecisionPoint.msg*, *FleetStatus.msg*, *FleetTask.msg*, *FreePath.msg*, *machineCmd.msg*, *MachineStatus.msg*, *MotionStatus.msg*, *ObstacleDetectionCmd.msg*, *TrafficControlCmd.msg*, *MotionOdom.msg* and so on. This code does not use any parameter servers for store-retrieve scheme of communication. At the time of conducting this study, sensor node and perception node was not functionally implemented and they were simulated in Matlab in order to verify other node's functionalities.

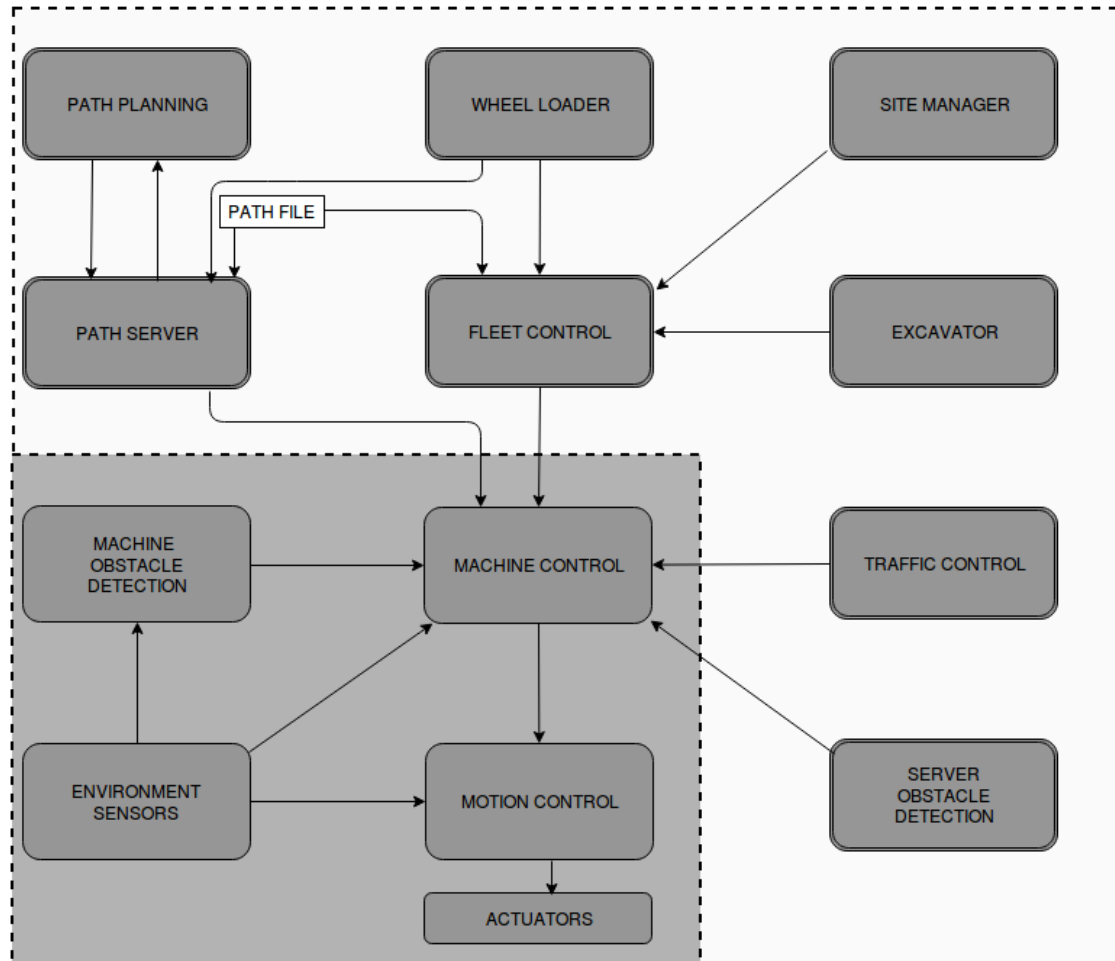


Figure 8: High Level software design architecture

Even though member functions are implementation specific, they are also covered during code walk through process. This gave an insight of certain critical information such as stand alone behaviour of traffic control from server side. This component manages traffic in the fleet and this is achieved by setting up a critical region and gateways in the path segments, which are highly prone to collisions. These functionalities are achieved by calculating the proximate distance between adjacent machines and estimated time of arrival for the next machine to the segment. *void trafficControl::gatewayControl()*, *void trafficControl::proximityCalc()*, *void trafficControl::setGateways()*, *bool trafficControl::calcIntersectCircleWL()* are certain functions which are critical in managing traffic on collision probable regions.

3.2 Approach to the Model Development

Once the code walk through is completed on the ROS code, we developed an UML class diagram which represents all the nodes in the source code. Each class included its associated variables, its data types, member functions of the class etc. Also the class diagram included computational and implementation details such as functions calculating position of machine with respect to x, y, z axes etc. Nevertheless, the entire code details are not required to develop its Rebeca model. Hence, in order to extract the relevant details we perform an analysis study with respect to certain criteria as given below:

Include Criteria (IC):

Below are the criteria defined during ROS code analysis for including the extracted entities.

- IC 1: All the active nodes from ROS source code.
Active nodes are those which shall be in executing state. These nodes has to be sending or receiving data to establish the communication.
- IC 2: All the interface connections
All the required inputs for the system as well as communication required for HX-HX or HX-WL connections are identified.
- IC 3: Nodes communicating with each other
Every node shall be communicating to an another one or more than one node. The nodes which are communicating with each other shall be identified.
- IC 4: Type of data (message)shared across the nodes (as Services or Topics)
ROS communication are of two types- via Services and via Topics. Each of these communication mechanism are modelled differently in Rebeca. Thus these mechanisms are distinctly identified.
- IC 5: Functional details of member functions
Member function in ROS specifies what needs to be performed by each node. This shall be identified to develop the functionalities in model.

Exclude Criteria (EC):

Following criteria is set so as to exclude certain extracted entities from analyzed data.

- EC 1: Implementation specific or computational details
Implementation specific details such as distance or position computation or mathematical calculation are not extracted from ROS code.
- EC 2: Print Functions or Display messages
More specific functions such as print statements or display commands are not logical to be an input for model development, hence they are ignored.
- EC 3: Hard coded variables
Variables which are directly assigned to a value to be used for the function implementation cannot be used for model development.
- EC 4: ROS specific functions ROS built in functions such as *advertise()* , *callback queue()*, *nodeHandle* etc. are not extracted.

Step 1: The initial phase of the model development process is to extract relevant features from ROS code. We represent this information using a class diagram, with all extracted data in it. Figure 9 shows the entire process, starting with creating a CLASS DIAGRAM with all extracted ROS code details. This unit includes all the required inputs for model development. Information used for this phase are derived from source code and based on include and exclude criteria. This is achieved by doing an entire code walk through and by analyzing each element against the defined criteria. During this process we have also included required inputs for the system, since those are

required for developing the complete model.

Step 2: The next task is to identify and map the extracted components to match with their connections so as to establish the communication similar to that of in real system. In this thesis, we have taken an approach to use UML design diagrams to represent the ROS code details as well as to represent the extracted details. Thus, an UML activity diagram is developed as shown in Figure 10. This diagram is just one level above the Rebeca model. It represents all the processes and connections relevant to the system. The model developer can use this diagram as a complete reference while developing his model. However, this UML diagrams are not mandatory for Rebeca model development. But it helped to cover all the necessary details of the complex system without missing any data.

Step 3: The third and final step is to develop the Rebeca model based on activity diagram. However, due to the complexity we have developed two different models to represent the entire system. The first model focused to represent the server design details and communication of server components towards machine; whereas the second model focused to develop machine specific design details. To represent the Rebeca model, UML use case diagram is used in this thesis report. Rebeca models look like a coded Java program and it is a tedious job to explain line by line, hence UML use case diagram serves the purpose of representing an abstract view of Rebeca model.

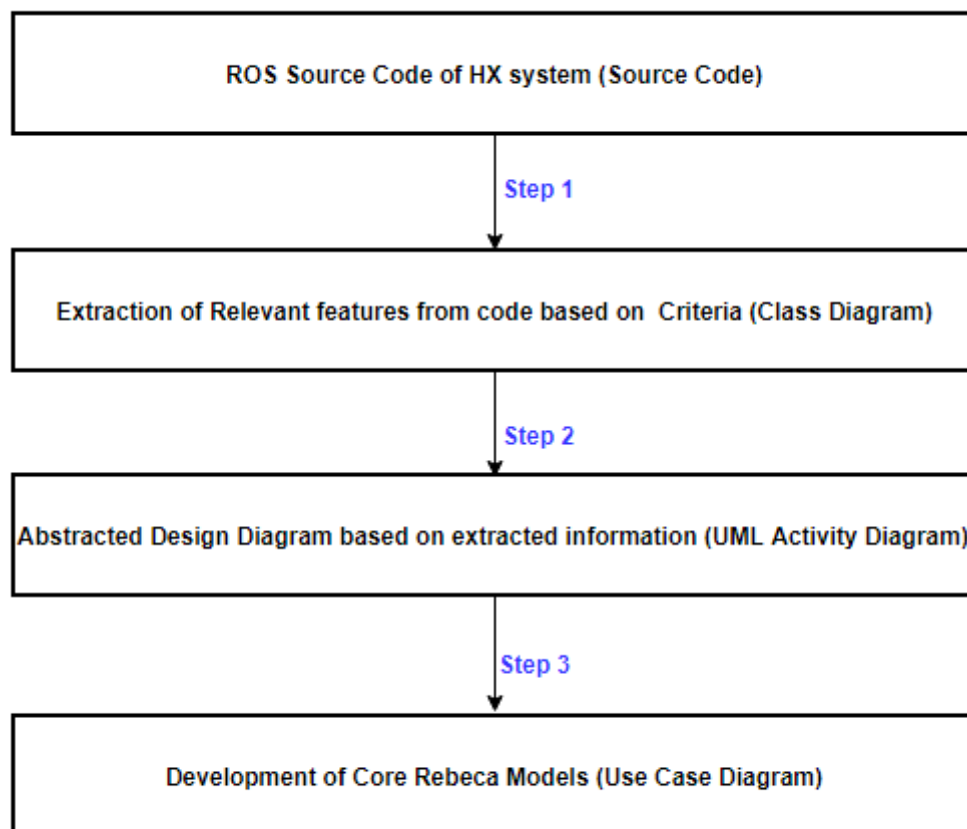


Figure 9: An overview of procedure followed for mapping process

3.3 Model Development of HX system design architecture

In this section, we describe the process of developing Rebeca models based on the information extracted from HX machine’s source code. The developed models shall not replicate ROS framework semantics and it shall be a pure Rebeca actor model of HX machine design architecture. The first step is to abstract relevant information. For this purpose the approach taken in this study is to develop an UML activity diagram representing all the nodes and their mapping/relation between each other. This step is not mandatory, however it helped as a reference while developing Rebeca model. Figure 10 shows representation of various nodes and their communication strategies in the ROS code.

In this stage, the actions or functions to be performed at each node is represented in rectangular boxes. Start state and temporary pausing of machines are represented with black circle and rounded black circle respectively. Decision making conditions are represented with diamond notation. It does not carry over any ROS semantics or computation details. This is the level of abstraction that we used, so as to extract input information for developing models. As seen in the Figure 10, there are several concurrent processes involved in this system. While verifying the model each and every possible state of this system shall be verified once against specified requirements to check whether they are satisfied or not. For clarity and to reduce complexity, two Rebeca models have been developed to represent the system. They are termed as 1.) **Server Model** and 2.) **Machine Model**, even though they are not independent models representing server and machine behaviour respectively. They together help to identify significant properties related to server-machine coordination. The models are developed using Core Rebeca Language, since we are not considering timing constraints of the system. The next step is to design and develop Rebeca models with the help of the UML representation as a reference.

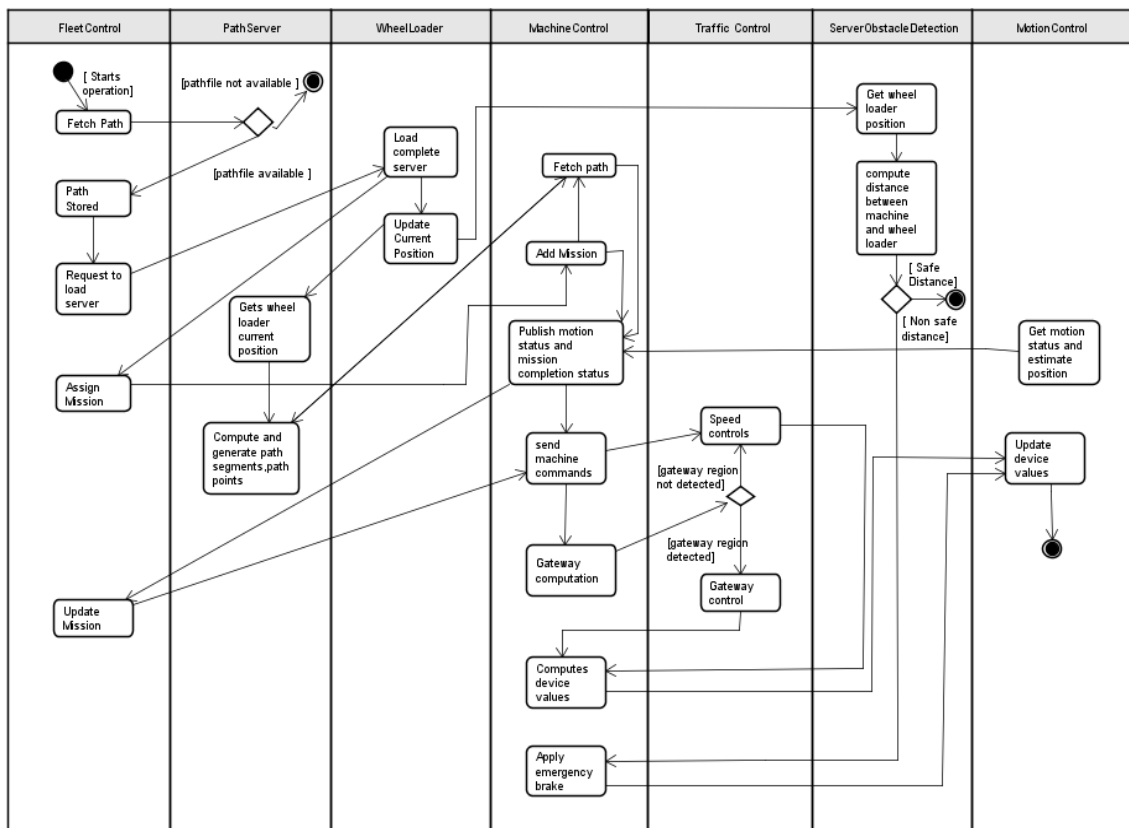


Figure 10: Representation of System model

In the first model, we use seven actors to represent server-machine co ordination.This model focuses to represent communication of 'Fleet Control' with other server components and how it establishes

communication with individual HX machines. Each node involved in this process are represented as capable of independent action as an actor. This model consist of seven actors including *Machine Control*, *Traffic Control*, *Fleet Control*, *Site Manager*, *Path Server*, *Excavator*, and *Wheel Loader*. Note that there are some actors (*Site Manager* and *Excavator*) which were not included in the abstracted reference from ROS code. These components came from design diagram and they have been included for completion of the model. Since we are not focusing on the functionality aspect, server obstacle detection component is not included in this model. The maximum message capacity for each actor is set to be 15. The actors needs to communicate each other are defined as their type in main class. As seen in Figure 10, Fleet Control tries to fetch site path file from Path Server upon starting the fleet operation. This fetching is done via wireless connection in real system, where there is possibility of connection not being established. This condition is represented using non-deterministic statements in Rebeca model. The sending of status commands to and from Fleet control and other Rebecs is achieved by asynchronous message passing using 'message servers'.

Thus, communication is established via message sending between the Rebecs, however an event shall be triggered for the actor upon receiving the message. This response for this event shall be sorted based on the pre-defined priorities and responds accordingly. For example, on receiving wheel loader status commands, the fleet control actor shall store this information to its internal message buffer. Fleet control then request for site path file, which has a higher priority input and waits for it. Depending on receiving the requested information, fleet control can continue the execution or drop it. This way, message passing is kept as the basic means of communication in the system, but its strategies are integrated by the actors. Moving on, the state variable *transport task* models the navigation mission in machine control node. Now that the communication is established between Path server to Fleet Control to Machine Control using message passing, we will see how to abstract computational details of Traffic Management node from ROS code. So this node is designed with several member functions computing safe distances between the machines while operating, braking distance, setting of gateway distance etc. The traffic control node is handling HX-HX interactions and smooth braking around the wheel loader. The server obstacle detection is just handling HX-WL (personal safety) with emergency braking. Since this is a static model representation, operating time computations are ignored here. We model the local state behaviour of the node using state variables. Thus, we model traffic management activities such as gateway setting, speed and brake control functions etc to be invoked upon triggering this Traffic Control actor. Figure 11 shows a Rebeca model representation for major server related activities. The dotted lines denotes that they are inputs for the server components and are not the part of real server system.

In the second model, four actors (*'Machine Control'*, *'Machine Obstacle Detection'*, *'Sensor'*, *'Motion Control'*) are used to represent individual machine design functionalities. Each HX machine is an AGV, which perceives environmental information through sensors and reacts back through actuators. This model also uses Figure 10 as reference, with a focus of different functionalities from first model. Here machine navigation is modelled by receiving sensor inputs, updated path file information and mission assignments from fleet control. These events are modelled by asynchronous message passing among corresponding actors. Apart from machine components, message server representing *Server Obstacle Detection* is included in this model which can directly control speed and brake values of individual machines. Again this is for completion of machine activity modelling. Further, Machine control is defined with no state variables in the actor definition, these variables and values are passed as input parameters in the main class where actors are declared. These parameters includes input commands (mission assignment) from fleet control, distributed path file from path server and wheel loader position updates. Server Obstacle Detection node being event driven, it is modelled using non deterministic expression. Commands to actuators/devices are represented as they are sent to Motion Control actor, where we assume that motion control component leads to corresponding actuators.

From ROS code, we observed that sensor node includes various kind of sensor information such as Lidar data, Odometry data, IMU and GNSS data. These information together helps to measure current position and status of the machine. However, this involves detail computational logic and

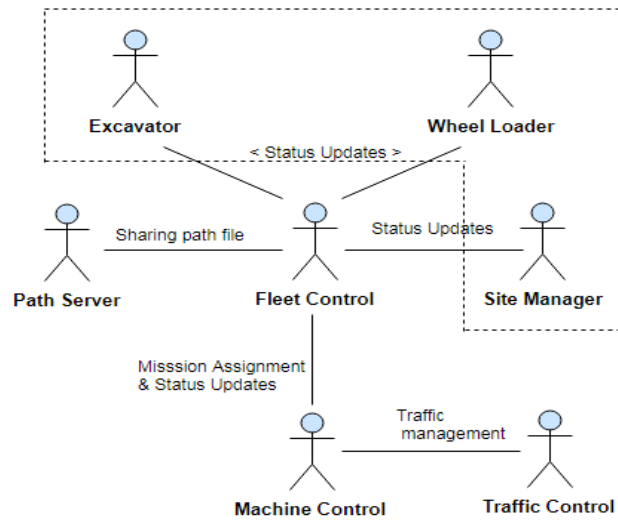


Figure 11: Server Actor Model Representation

hence we discussed with experts to get an understanding of its significance. From their opinion, Lidar information is prone to unavailable connectivity during operation and upon this condition, machine navigation can become erroneous to follow predefined track. This is considered as one property to be verified with this model. Now based on this information, while specifying sensor properties, we represent it as 'lidar sensor data'. Perception node also involves computational details for generating obstacle free navigation path. Hence the model representing this node acts as generating free path upon receiving sensor data, otherwise not. Another attribute about HX machine is that machinecontrol always possess a correct knowledge of wheel loader's current position, since these are mobile elements in the operating site.

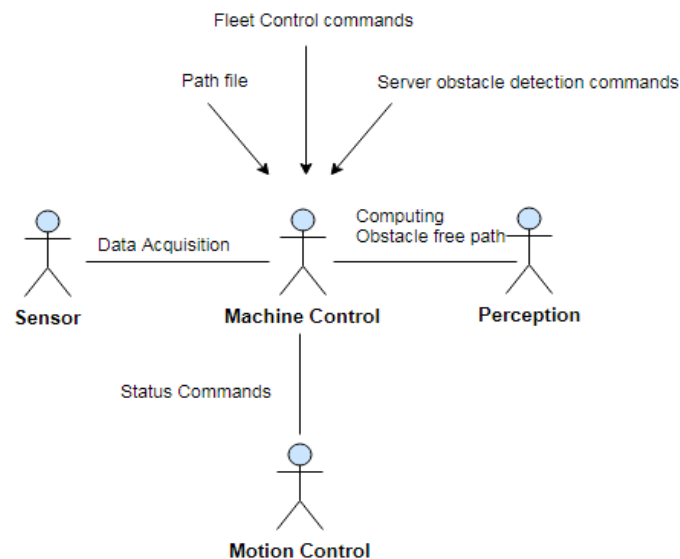


Figure 12: Actor model for Machine Level Model

3.4 Evaluation of Properties and their Results

In this section, we will specify HX system design specific properties and their evaluation using the developed models. As explained in 3.3, two models are developed for clarity and less complexity. It also helped to identify relevant properties of the system. Selected properties from Server model are discussed first, followed by its evaluation results. After which properties from Machine model shall be described along with its evaluation result.

3.4.1 Properties of Server Model

For a given model and its specified property, model checking is based on an extensive exploration of the reachable state space of the system and by verifying whether there exists a state where the specified property is satisfied or not[31]. Here the selection of properties are based on the binding parameters for the system operation with the focus on Liveness and Safety related specifications. 'Liveness' is one significant property to be verified in distributed systems. Satisfied liveness properties of the system ensures the operational progresses while concurrent system components have critical sections. Similarly 'Safety' should be verified for any safety critical systems. A safety property ensures that nothing bad happens in the system. For example, a permanent STOP or deadlock situation/ no outgoing transitions or erroneous behaviour etc. are included in safety properties. In Rebeca model, all the properties are specified using LTL statements. Thus, the specified properties for server model are as below:

A. Control commands from 'Machine Control' node leads to triggering of 'Traffic Control' functions

From the design it was extracted that traffic control node do not depend on any components located at server but only on Individual machine's status commands. I.e traffic control node has a dependency on machine control commands for the activation of traffic management function. The traffic management function's response can trigger either speed and brake signal or gateway control signal. Gateway control is required by traffic management when machine moves through critical section of the fleet segment. Similarly speed and brake values are needed by traffic management during a 'navigation' event occurrence other than machine is in paused or queued state. However, those details are abstracted and we verify only the functionality. This property can be specified using LTL formulae as below:

```
G [(machinecontrol.transporttask)-->
  F((trafficcontrol.speedandbrakecontrol) || (trafficcontrol.gatewaycontrol))]
```

In Afra, in order to specify LTL statements, the variables used in the model have to be first defined. m1, m2, m3, m4 are statements to define the associated variables used in the model for specifying the above property. After this, we specify safety properties as LTL expressions.

```
m1= machinecontrol.transporttask ;
m2= trafficcontrol.speedandbrakecontrol;
m3= trafficcontrol.gatewaycontrol;
m4= m2 || m3;
```

```
Property 1 : G [m1 -> F (m4)]
```

This property 1 verifies that whenever transport task variable is true, it leads to a state where variable speedandbrakecontrol and gatewaycontrol are true and this condition holds globally in the model. In other words always when the transporttask commands are sent, finally either speedandbrakecontrol signal or gatewaycontrol signal is enabled.

B. At any point, 'Machine Control' node has a true knowledge of current position of 'wheel loader'

It is a requirement from the design and from the experts that current position of wheel loader must be known to the machine. This is ensured for error free execution of machine tasks. From the ROS code, we learned that it is via the distributed path segments generated by 'pathserver'

node, machine control obtains updated wheel loader position. However, we verify this property of concurrent operation. It can be specified using LTL formulae as below:

```
G[(wheelloader.setwheeloaderStatus)→ F(machinecontrol.transporttask)]
```

In Afra, we first define the variables used in the model using expressions m5 and m6. Property 2 specifies that whenever wheel loader status is set to be true, it finally leads transport task variable to be true. In other words, always when the wheel loader position is set, finally the machine perform its transport task.

```
m5= wheelloader.setwheeloaderStatus;
m6= machinecontrol.transporttask;
```

```
Property 2 : G [m5 → F (m6)]
```

C. Fleet control node requires three mandatory inputs in order to perform error free fleet coordination

Once site manager enables fleet operation, the fleet control requires three inputs for starting the process. This includes site path file (which contains path information about the entire site), wheel loader status(to know the current status of wheel loader) and Excavator status (to know the status of excavator status). These are incoming inputs to server from components outside the server, these inputs are significant as it assures a level of safety in operation. Property specifying this requirement can be represented using LTL formulae as below:

```
G[(wheelloader.setwlststatus)&&(excavator.setexstatus)&&(fleetcontrol.sitepathfile)
→ F(fleetcontrol.assignmission)]
```

In Afra, we define the variables used in the model for specifying this property. Statements m9 to m13 defines it and Property 3 verifies that fleet control cannot perform error free fleet coordination, if any of the required inputs are not available. In other words, whenever 'setwlststatus' from wheel loader, 'setexstatus' from excavator and 'sitepathfile' from fleetcontrol are true, finally 'assign mission' of fleetcontrol shall becomes true.

```
m9= (wheelloader.setwlststatus);
m10= (excavator.setexstatus);
m11= (fleetcontrol.sitepathfile);
m12= (fleetcontrol.assignmission);
m13= m9 && m10&& m11;
```

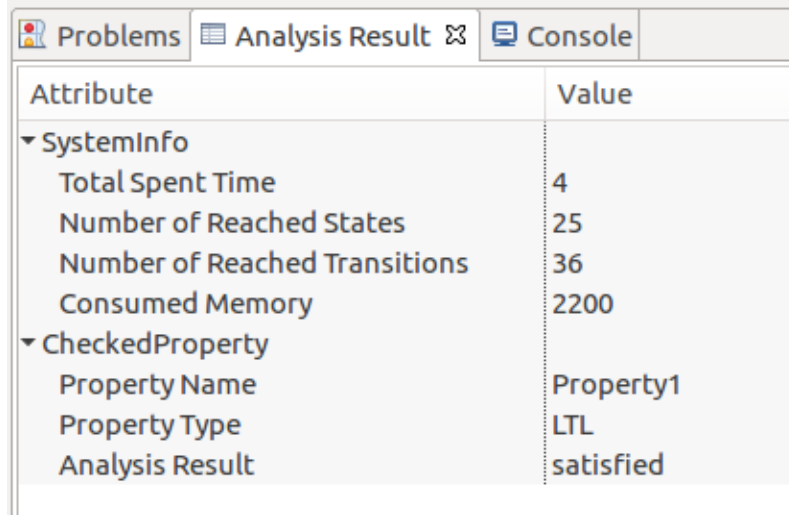
```
Property 3 : G [m13 → F (m 12)]
```

3.4.2 Verification Results of Server Model Properties

In this section, we present the results of verified properties of Server model. All the properties are verified using Afra3.0 model checker. We found that the specified properties are all satisfied. In this section, screen shots of verification result for each property are included.

Property 1 verifies whether machine control commands leads to the activation of traffic management functions. Traffic management function activation is represented by a boolean variable 'transport task'. The verification result shows that this property holds for server model. The verification result for property 1 ensures that whenever machine control sent its control commands, it invokes the activation of traffic management functions in traffic control node. Further verification result shows that the traffic control component response can trigger either speed and brake signal or gateway control signal.

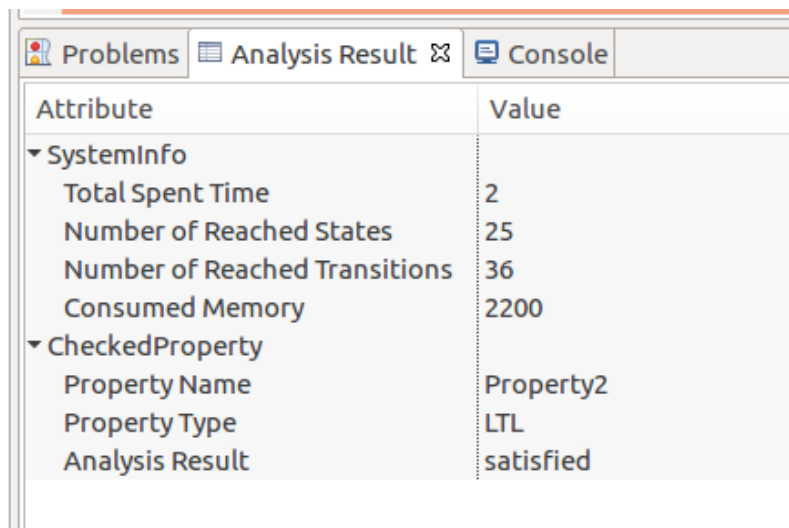
Property 2 verifies that machine control has a correct knowledge of active wheel loader's current status at any given point of time. This property is significant because if machine control is not aware of wheel loader current status, it may result in performing error in assigned task. This property ensures following sequence of actions. The wheel loader's current position shall be updated with path server. The path server then sends distributed path file to machine control which contains latest status of all resources including wheel loaders. Here two conditions are verified. 1) Condition



Attribute	Value
SystemInfo	
Total Spent Time	4
Number of Reached States	25
Number of Reached Transitions	36
Consumed Memory	2200
CheckedProperty	
Property Name	Property1
Property Type	LTL
Analysis Result	satisfied

Figure 13: Verification Result of Property 1

“wheeloaderstatus=true” eventually leads to “taskexecution” in machine control. 2) Condition “wheeloaderstatus=false”, which means status is not known, eventually leads to a state where “task execution” in machine control is not active. On model checking, we got both these conditions satisfied. The screen shot for analysis result is as given in Figure 14. The result report consists of total number of states verified, number of state transitions, total time taken to complete the verification process in seconds and so on.



Attribute	Value
SystemInfo	
Total Spent Time	2
Number of Reached States	25
Number of Reached Transitions	36
Consumed Memory	2200
CheckedProperty	
Property Name	Property2
Property Type	LTL
Analysis Result	satisfied

Figure 14: Verification Result of Property 2

Property 3 verifies that three mandatory inputs are required for fleet control to start assigning mission to individual machines. This property is significant as it impacts the general fleet operation activity. The binding inputs are wheeloaderstatus, excavator status and the sitepath file. Thus, we verify that absence of one of these inputs can lead to a state, where mission assignment by fleet control pauses temporarily. Similarly we also verify the ability of fleetcontrol to assign machine mission when all the three inputs are available. This property was verified as satisfied and screenshot of the analysis result is given in fig15.

Attribute	Value
▼ SystemInfo	
Total Spent Time	0
Number of Reached States	25
Number of Reached Transitions	36
Consumed Memory	2200
▼ CheckedProperty	
Property Name	Property3
Property Type	LTL
Analysis Result	satisfied

Figure 15: Verification Result of Property 3

3.4.3 Discussion on Server Properties

A satisfied property result from model checker refers that the verified property may also be true in the real system. However, to make sure that our model has represented an abstracted version of real system, we injected faults into the model and verified the properties again. In other words, if model is not representing the true system, then the properties can fail. When properties are failed, model checker generates a counter example to show the trace of execution which caused the violation of properties.

With respect to server model, the property 1 verified that control commands from machinecontrol node leads to activation of traffic management functions in trafficcontrol node. control commands from machine control node was represented using a boolean variable 'transporttask' and traffic management functions in trafficcontrol node was represented using two boolean variables 'speedandbrakecontrol' and 'gatewaycontrol'. So whenever transporttask is set to true, it leads to the state where either speedandbrakecontrol=true or gatewaycontrol=true. In order to inject a fault to violate this property, we changed the value of speedandbrakecontrol variable inside the message server which represents traffic management function. The faulty model thus represent that whenever traffic management message server is called, speedandbrakecontrol shall be set deactivated (false). Now this model was verified against property 1, and the result was verification failed with a generated counter example. The result console is as shown in Figure 16.

Attribute	Value
▼ SystemInfo	
Total Spent Time	1
Number of Reached States	32
Number of Reached Transitions	51
Consumed Memory	2816
▼ CheckedProperty	
Property Name	Property1
Property Type	LTL
Analysis Result	counter example

Figure 16: Verification Result of Property 1 against fault Injection model

Further the model was modified with a fault to verify the property, that a status update from wheel loader shall finally reaches to machinecontrol node. To do this, the variable representing wheelloader status update in machine control node was not set to true, whenever wheel loader sent its update. The faulty model represent a system in which machine control node has no knowledge about the current wheel loader position. This model was verified against property 2 and its model checking result shows a failed property check with counter example. The result obtained from Afra is shown in Figure 17.

Attribute	Value
▼ SystemInfo	
Total Spent Time	0
Number of Reached States	32
Number of Reached Transitions	51
Consumed Memory	2816
▼ CheckedProperty	
Property Name	Property2
Property Type	LTL
Analysis Result	counter example

Figure 17: Verification Result of Property 2 against fault Injection model

To verify property 3 against a fault injected model, the model was slightly modified from real system representation. Property3 states that fleet control operation requires three inputs -from site manager, wheel loader and from site path file. We made sitepath file unavailable at the class definition of fleet control in Rebeca model, in which case variable 'vehiclepaus' shall be set to true. Instead we hard coded vehiclepaus variable to true and verified the model. The faulty model represents that even if sitepath file is unavailable, vehicle shall not get paused. This model was verified against property 3 and the result generated a counter example. Result screen is shown in Figure 18. The number of state transitions verified for this model checking process is 51 and a total of 32 states were verified for this process.

Attribute	Value
▼ SystemInfo	
Total Spent Time	0
Number of Reached States	32
Number of Reached Transitions	51
Consumed Memory	2816
▼ CheckedProperty	
Property Name	Property3
Property Type	LTL
Analysis Result	counter example

Figure 18: Verification Result of Property 3 against fault Injection model

3.4.4 Properties of Machine Model

Coming to machine model, we have identified requirements related to individual machine operation. As mentioned in the Section 3.4.1, liveness property refers that “something good will eventually happens” and safety property refers where “something bad will never happen”. Below are the selected properties to be verified in machine level model. These properties were identified based on inspiration from a work done by Tichakorn Wongpiromsarn and Richard M. Murray [31].

A. Non-availability of sitepathfile to 'machinecontrol' node leads to a 'machine paused' state

Machine Control requires three binding inputs to perform its operation, the first one being site path file which includes the distributed path information from 'pathserver' node. Machine navigation takes a temporary pause if it does not receive sitepathfile input in order to avoid any undesired events. It could be due to lost connectivity with path server node. The operation shall be resumed on receiving this file input. This property can be expressed using LTL statement as below:

```
G[(!machinecontrol.sitepathfile) -> F (motioncontrol.vehiclepause)]
```

In Afra, the variables are defined first in order to specify the above property. Property 4 verifies that the condition variable sitepathfile is not true finally leads to a state where vehiclepause=true and this condition is globally true in the model.

```
m14= !machinecontrol.sitepathfile;
m15= motioncontrol.vehiclepause;

Property 4 : G[m14 -> F(m15)]
```

B. Application of Server emergency brake signal leads to stopping of vehicle movement

Server obstacle detection is a server component which helps in HX-wheel loader coordination actions. In machine model we have included this component as a message server. This is because the main function of this block is to control the machine movement when detected with an active wheel loader in the close proximity of machine, thereby preventing a collision scenario. It can be put into LTL statement as below:

```
G [(machinecontrol.serveremergencybrake) -> F(motioncontrol.vehiclestop)]
```

In Afra, the variables serveremergencybrake and vehiclestop are defined as below and the Property 5 verifies that whenever the variable serveremergencybrake is true in the model, it finally leads to a state where variable vehiclestop is true.

```
m16= machinecontrol.serveremergencybrake;
m17= motioncontrol.vehiclestop;

Property 5 : G[m16 -> F(m17)]
```

C. Non- Availability of Sensor data can lead to temporary vehicle pausing state.

Similar to property 1, this property is specified to ensure the availability of sensor data inputs to the machinecontrol node. Now sensor node consists of a combination of several sensor devices such as Lidar, Odometry, IMU etc. From expert knowledge, lidar data is most prone to non available during operation. Hence we focused only on assuring lidar data availability as their input which is vital in computing machine position. This is because lidar sensor measures target distance by sending a pulsed laser light to target and measures the reflected pulses to get the distance value.

LTL statement specifying this property is :

```
G[(!sensor.sendlidardata) -> F(motioncontrol.vehiclestop)];
```

In Afra, the variables `sendlidardata` and `vehiclestop` are defined first. Property 6 verifies that whenever lidar sensor data is unavailable in the model (which is denoted as the variable `sendlidardata` is false), it finally leads to a state where variable `machinestop` in motioncontrol node is set to true.

```
m18 = !sensor.sendlidardata;
m19 = motioncontrol.vehiclestop;

Property 6 : G[m18 -> F (m19)]
```

3.4.5 Verification Results of Machine Level Properties

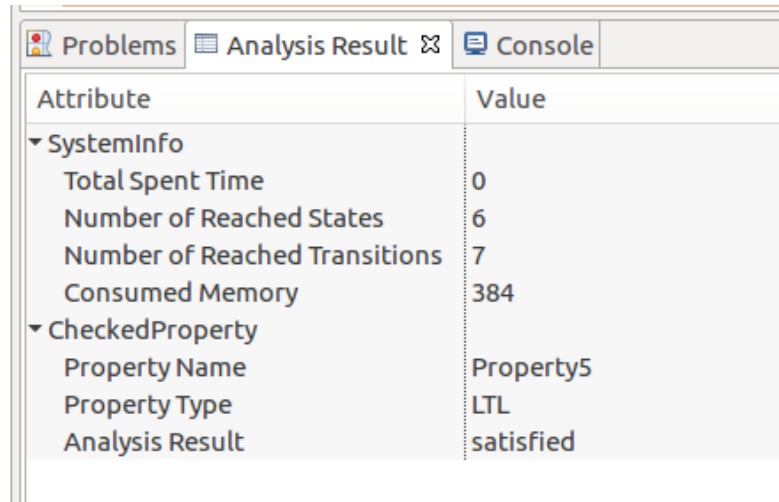
Property 4 verifies that non availability of `sitepathfile` leads to a temporary vehicle paused state. Based on the model, the specified LTL property verifies that a condition in which variable `sitepathfile` when set to false finally leads to another state where variable `vehiclepause = true`. This property shall hold for the model in all execution trace. The screen shot of analysis result is as shown in Figure 19. A total number of 7 state transitions were verified for verifying this property.

Attribute	Value
▼ SystemInfo	
Total Spent Time	0
Number of Reached States	6
Number of Reached Transitions	7
Consumed Memory	384
▼ CheckedProperty	
Property Name	Property4
Property Type	LTL
Analysis Result	satisfied

Figure 19: Verification Result of Property 4

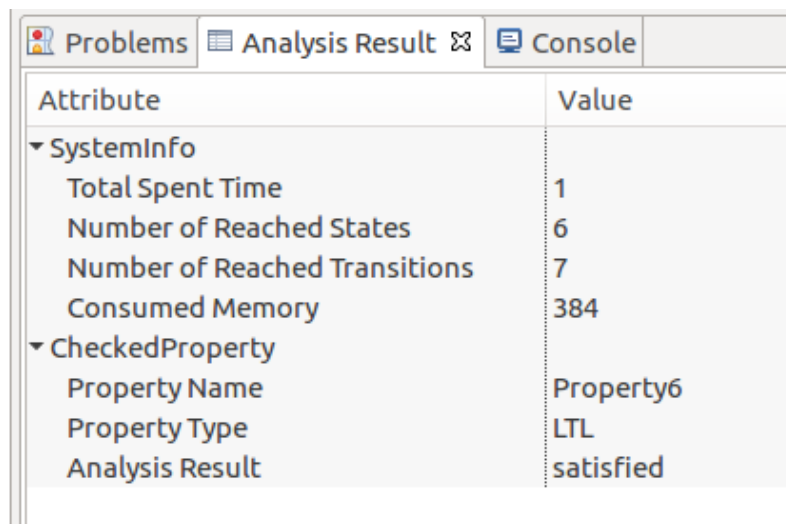
Property 5 verifies that whenever a braking command is published from server obstacle detection, machine is immediately stopped until the braking command is released. This is to ensure HX-wheel loader collision avoidance. Based on the model, the specified LTL property verifies that a condition in which variable `serveremergencybrake = true` always leads to vehicle stop condition represented by the variable state `vehiclestop=true`. The screen shot of analysis result is as shown in Figure 20.

Property 6 verifies another safety condition where non availability of sensor data leads to a temporary machine stopped state. Based on the model, the specified LTL property verifies that a condition in which variable `sendlidardata` when set to false finally leads to another state where variable `vehiclestop = true`. This property shall hold for the model in all execution trace. The screen shot of analysis result is as shown in Figure 21.



Attribute	Value
▼ SystemInfo	
Total Spent Time	0
Number of Reached States	6
Number of Reached Transitions	7
Consumed Memory	384
▼ CheckedProperty	
Property Name	Property5
Property Type	LTL
Analysis Result	satisfied

Figure 20: Verification Result of Property 5



Attribute	Value
▼ SystemInfo	
Total Spent Time	1
Number of Reached States	6
Number of Reached Transitions	7
Consumed Memory	384
▼ CheckedProperty	
Property Name	Property6
Property Type	LTL
Analysis Result	satisfied

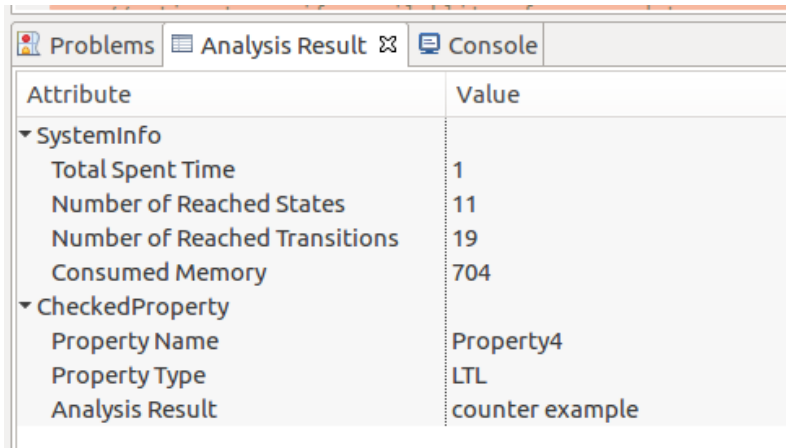
Figure 21: Verification Result of Property 6

3.4.6 Discussion on Machine Properties

Similar to the validity check done for server model, we also verified machine model by injecting a fault in the Rebeca code and then verified it against specified properties. Thus, a faulty model can result in failure of properties which did satisfied for the correct model. The result of a property check gives whether the property was satisfied or not, in second case a counter example is shown.

Property 4 verified that non availability of sitepathfile to machine control leads to a condition in motion control node where vehicle is set to pause state. The fault injection was made such that the variable representing vehicle pausing was set to false when sitepathfile is non available. This faulty model when verified against property 4 resulted in failed result with a counter example. The screenshot for failed verification result of property4 is shown in Figure 22.

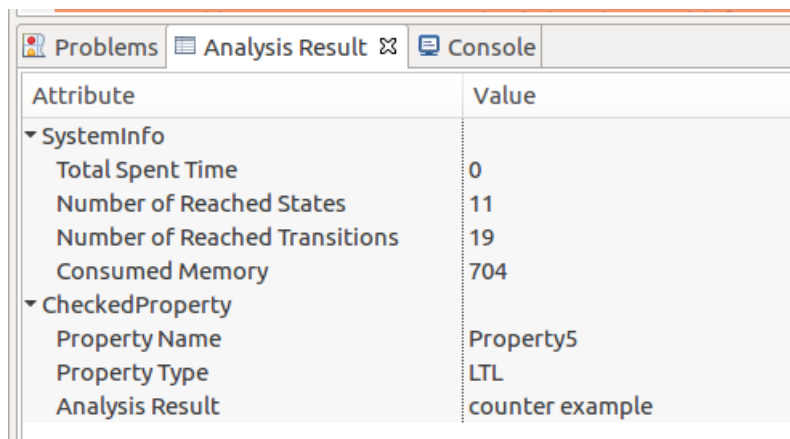
Property 5 verifies that personal safety with machine-wheel loader interaction. This property verifies that if server emergency brake is applied, it is of high priority and vehicle to which this brake signal was sent shall be stopped immediately. In model, server emergency brake signal is modelled using non- deterministic expression and in faulty model, the variable representing vehicle stop is set to false when ever emergency brake signal is set to true. Verification result for this faulty model against property 5 gives a failure result with counter example trace. The screen shot



Attribute	Value
▼ SystemInfo	
Total Spent Time	1
Number of Reached States	11
Number of Reached Transitions	19
Consumed Memory	704
▼ CheckedProperty	
Property Name	Property4
Property Type	LTL
Analysis Result	counter example

Figure 22: Verification Result of Property 4 against fault Injection model

of analysis result is as shown in Figure 23.



Attribute	Value
▼ SystemInfo	
Total Spent Time	0
Number of Reached States	11
Number of Reached Transitions	19
Consumed Memory	704
▼ CheckedProperty	
Property Name	Property5
Property Type	LTL
Analysis Result	counter example

Figure 23: Verification Result of Property 5 against fault Injection model

Property 6 is similar to property 4; instead of site path file, we verify that if sensorlidardata is unavailable vehicle shall be paused until it is available. Similar to other fault injections, here also we hard coded the value of vehiclepause variable to false. Thus, the model represents that even if sensor data is not available, vehicle shall not be paused. Property 6 result shows failed property check with a generated counter example. The result obtained from Afra is shown in Figure 24.

As it can be seen from every verification result report, the number of transitions and number of reached states are huge during model checking process. This number is the maximum possible states and their transition which the system can undergo during execution. It is clear that verification of a system requirement on such a complex system using functional testing can be a tedious and time consuming job. This is the significance of model checking technique.

Attribute	Value
▼ SystemInfo	
Total Spent Time	1
Number of Reached States	11
Number of Reached Transitions	19
Consumed Memory	704
▼ CheckedProperty	
Property Name	Property6
Property Type	LTL
Analysis Result	counter example

Figure 24: Verification Result of Property 6 against fault Injection model

3.5 Pattern Mapping from ROS to Rebeca

In this section, we present the patterns identified when a ROS based system is mapped to Rebeca model. This also includes certain patterns which were identified as not possible to be mapped. Following subsections describe each of these patterns with the titles representing ROS terminologies. A table representing ROS-Rebeca mapping pattern list is also included at the end of this section. The identified patterns are general to any ROS code and is not specific to the ROS code of case study considered for this work. The case study specific ROS code is not included due to confidentiality reasons.

3.5.1 ROS Nodes

Node is a basic processing unit in ROS that performs simple to complex computation. Various nodes communicate with each other using different communication schemes. A robotic system shall consist of several number of nodes. For example in an autonomous guided vehicle each node performs different functionalities like path planning, obstacle detection, fleet management, traffic control etc. In other words, main communication in ROS is achieved by message passing across nodes.

A ROS node can be mapped to an 'actor' or 'Rebec' in Rebeca model. Similar to a node in ROS, actors are the basic processing units in an actor model. By representing communication and co ordination across the actors, system computation can be modelled in Rebeca. However, the computation logic and details shall be properly abstracted and model development is based on the response of an actor for its received input. Message passing between ROS nodes corresponds to message sending between actors in Rebeca. Similarly computation or event triggering in ROS corresponds to response behaviour or control flow representation in Rebeca.

Figure 25 shows mapping of a ROS node "talker" which performs simple sending of messages to its Actor Model. Here "Talker" is the Node name and it sends messages at a rate of 10 to another node. In Rebeca, action and reaction of sending and receiving of messages are modelled. The code snippet is taken from the licensed official internet page of Open Source Robotics Foundation[50]

3.5.2 ROS Message Passing

As mentioned in Section 3.5.1, nodes communicate by passing messages to each other. This means when a ROS node is created and initialized, it is capable of sending data to its intended recipients. The node which shall receive this data has to be separately created and initialized. One node can send messages to multiple nodes based on the requirement. However, in Rebeca there is a neces-

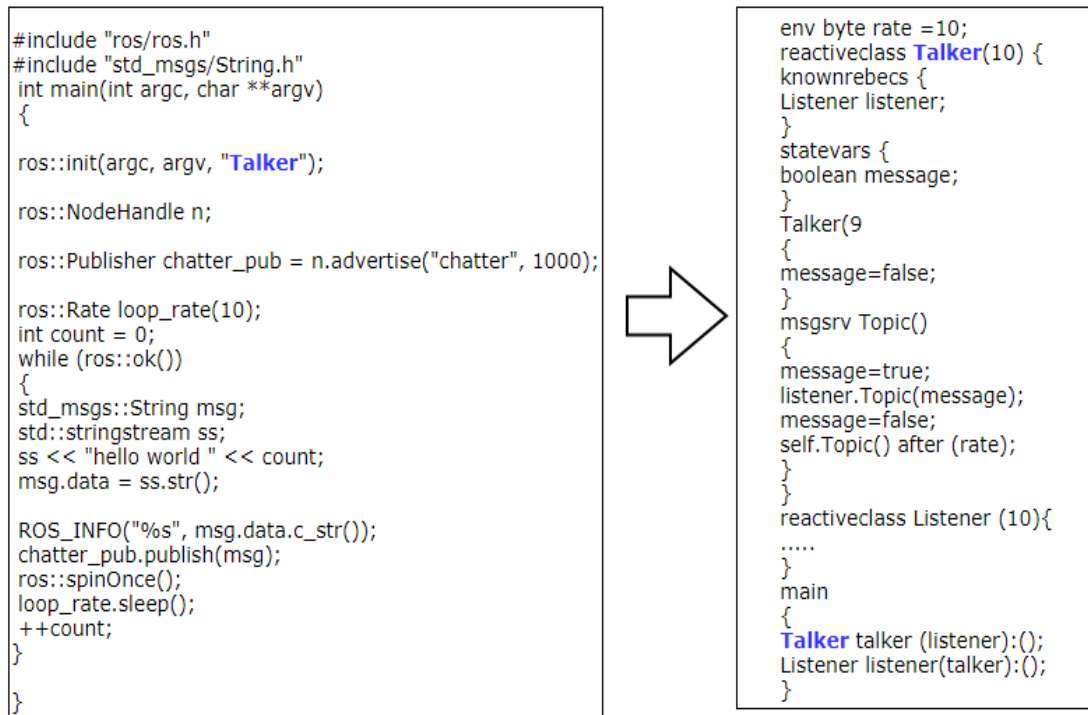


Figure 25: Mapping of ROS Node to Rebeca Actor

sity of knowing the recipient node(s) beforehand and they has to be declared as 'Knownrebecs'. Thus, 'message passing' pattern in ROS can be mapped to a combined action of 'knownrebecs' and 'message servers'in Rebeca.

As seen in Figure 26, message passing in a ROS node is done as a callback function 'chattercallback' that shall be called whenever a message is received at the node. A pointer 'msg' is passed which indicates that the arrived message could be stored if required. In Rebeca, the receiving message is stored in the queue. When the message is extracted for execution, its corresponding message server is executed. This is achieved by defining knownrebecs for sending and receiving actors and message sending is modelled with the help of message servers defined in rebecs and send between known rebecs. In Figure 26 line 8 in block A defines the publisher function "chatter_pub", with topic name as 'chatter'. Line 11 in block B defines the subscriber function "sub" which invokes a call back function. The call back function is defined in line 3 of block B. Here messages from node talker(block A) is send to node receiver(block B) through topic chatter. In block C, the rebeca model corresponding to this pattern is represented. Line 3 models the subscriber and topic as "knownrebecs" to the talker node. Line 9 defines the functions of talker node as message server. Line 15 models the functions of topic. Line 23 models the actual message sending using message servers. Line 27 models the subscriber node as Listener rebec. This rebec is defined with talker and topic as its knownrebecs and finally line 34 models that messages are received at subscriber rebec.

3.5.3 ROS Variables

We use variables in ROS, both local and shared ones, like any other programming language. Local variables are those which are defined to be inside one class. They cannot be accessed from outside the class, whereas shared variables are those which are defined outside any particular class such that they can be accessed from across any class or function of the program. Similar to this, we tried to identify patterns in Rebeca. Local variables could be mapped to Rebeca state variables. They denote the current state of that actor. However, Rebeca does not have a concept of shared

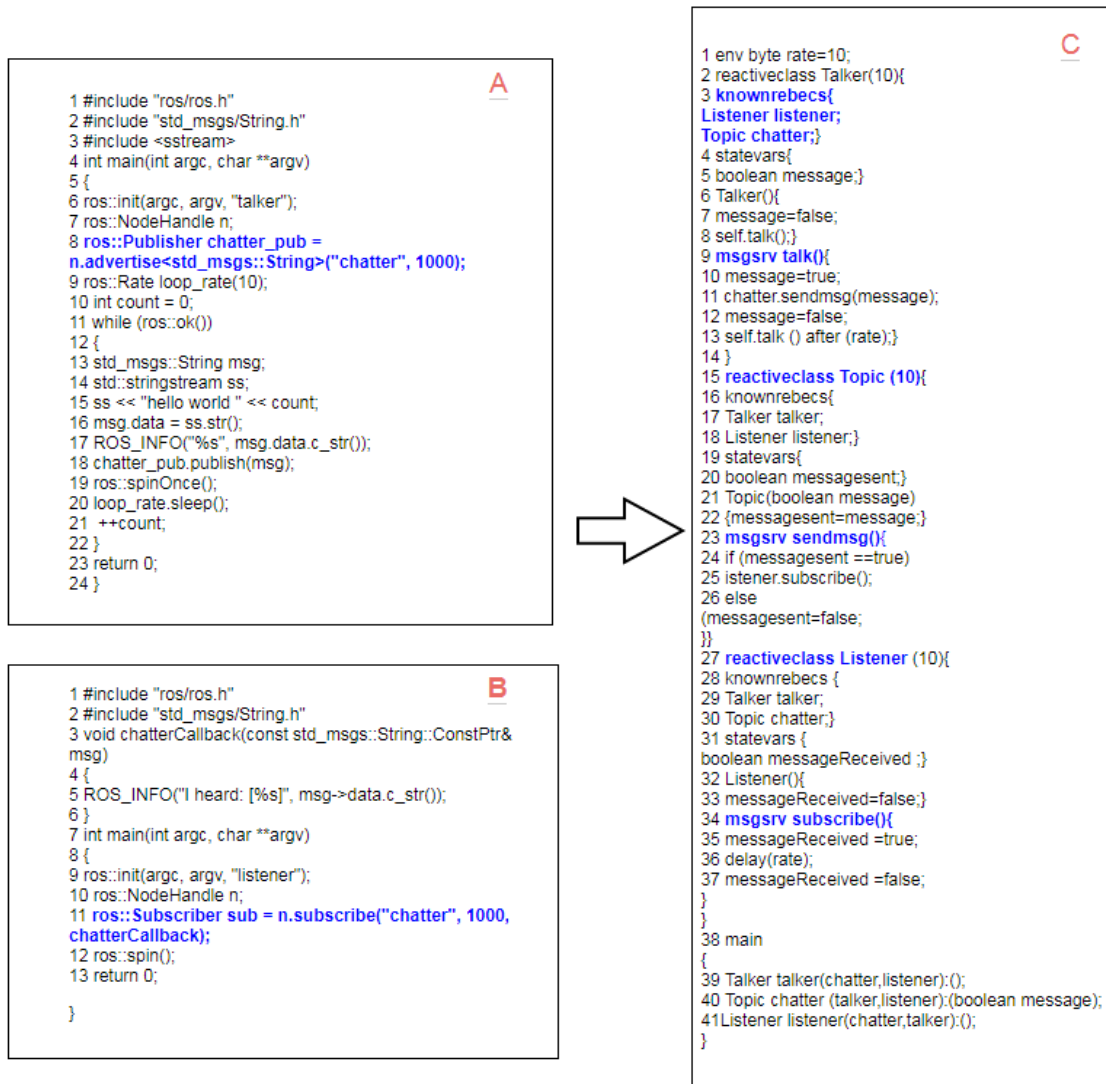


Figure 26: Mapping of ROS Message Passing to Rebeca KnownRebecs & Message Servers

variables. A variable can only be used within a reactiveclass and cannot be shared

Mapping of ROS local variable pattern is shown in Figure 27. Here count is a local variable in ROS node 'talker', it has been initialized after defining the node. In rebeca, all the local variables to be used in a class shall be defined within the 'statevars' section.

3.5.4 ROS Distributed Parameter System

Another mapping pattern which was derived from ROS framework to Rebeca model is 'ROS distributed parameter system'. Configuration information could be shared in ROS using global key values. In other words tasks could be easily modified using parameter system. We consider an example for this from ROS tutorials[50]. We define certain parameters as below :

```

camera/left/name: leftcamera
/camera/left/exposure: 1
/camera/right/name: rightcamera

```

Thus, the parameter '/camera/left/name' can get the value of 'leftcamera' wherever defined. It

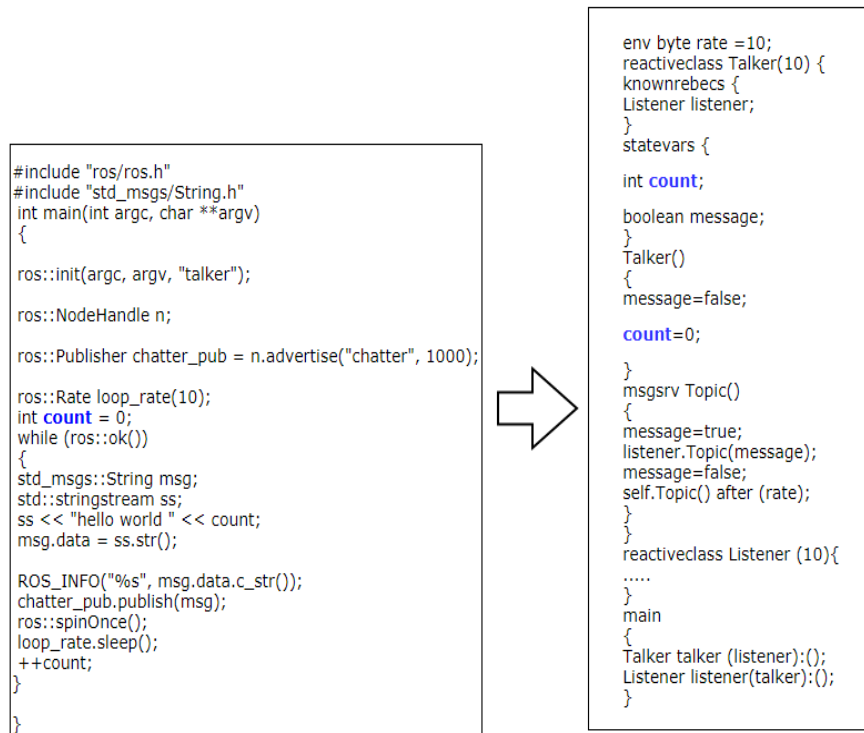


Figure 27: Mapping of ROS local variables to Rebeca statevariables

can also act as an dictionary, however we will not sneak into details of that. A mapping pattern similar to the above ROS terminology would be 'environment variables' in Rebeca. But we cannot achieve the same result of using a parameter system in ROS. This is because ROS uses XMLRPC data types for parameter values, which includes integers, boolean, doubles, strings etc. Meanwhile in Rebeca string data types cannot be defined. Figure 28 shows mapping of a ROS code to Rebeca model in which definition of an environment variable "rate" are shown.

In Figure 28 'env byte rate=10' is the syntax for declaring an environmental variable. The variable 'rate' could be mapped to a ROS parameter which carries the same value wherever defined in the Rebeca model.

3.5.5 ROS member Functions

A member function defined in ROS class can access all the objects of that class and also all the members of a class for that object. Computations as well as other functional manipulations shall be performed using the member functions. In Rebeca, we abstract these computational and manipulation details and model the functionality of nodes.

In Figure 29, pattern mapped from ROS to Rebeca is shown. 'Advertise' is a member function which shall publish string messages. In Rebeca, we model this as a message server as highlighted in the figure. Most of the member functions can be modelled using boolean statements, which can be either true or false values. These statements shall be defined in corresponding message servers.

3.5.6 ROS shared Variables

In ROS, Shared Variables carry value within a class or they can be accessed from outside a class. However, this pattern was not directly found in Rebeca language. In Rebeca, a variable can be only local to a class and they shall not be accessed outside of it. A Rebeca model can bypass usage of shared variables by its design. This is because modelling technique is different in Rebeca even

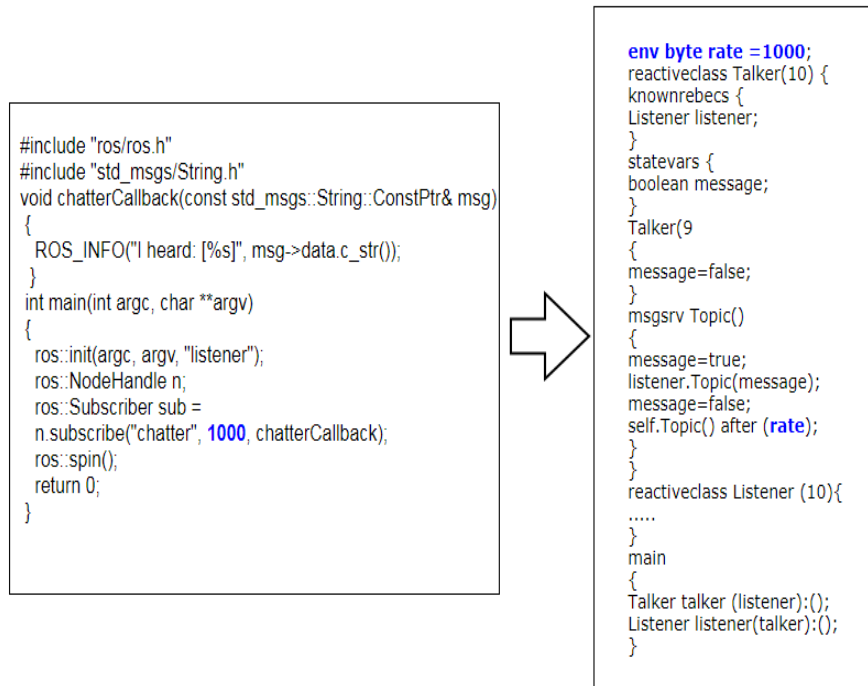


Figure 28: Mapping of ROS parameter system to Rebeca Environmental Variable

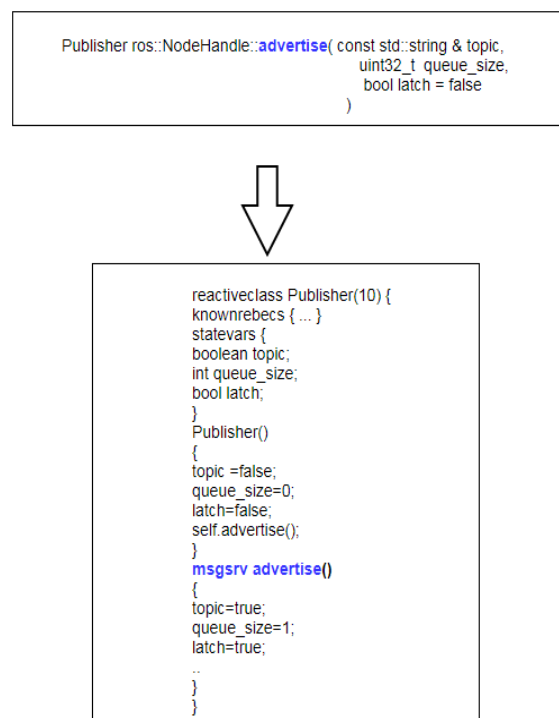


Figure 29: Mapping of ROS member function to Rebeca message server

though a major part of communication in ROS and Rebeca is via asynchronous message passing.

3.5.7 ROS Bags

A ROS bag is a format to store ROS messages. These messages are stored and later it can be processed, analyzed and visualized. Thus, they serve as a template of storing-playback scheme. However, such a pattern could not be traced in Rebeca. A possible reason for this can be rebeca models focus at model checking and verification, whereas ROS allows ROS bags feature which are useful to perform diagnostics and visualization.

3.5.8 ROS Display functions

Another ROS pattern which is not possible to be mapped to Rebeca are Display functions, which are for printing out messages. These functions are similar to Cout /Cin functions used in C++. Since displaying a message is not in the scope of a model, this kind of patterns are not possible to be modelled in Rebeca.

3.5.9 ROS Services

In ROS, the services are communication scheme which are of the request-response scheme, which are often required in a distributed system. This means that a ROS node can request for a service from another ROS node. The node requesting the service is called 'client node' and the node serving the service is called 'server node'. The services usually have an unique name and they are defined using a pair of messages- 'request' and 'response'. The server node offers a service and client node requests for this service by sending a request and waits until it receives. In system, this waiting time denotes that a process is getting blocked until its request is served. Thus, in ROS service communication is synchronous in nature. This means that, in Rebeca if the service is not available immediately, the next process shall be executed and there is no blocking for the initial task. Hence, in Rebeca task execution occurs via non-preemptive manner. While mapping to Rebeca, the challenge was to model this synchronous communication of ROS services since Rebeca communication are through asynchronous message passing. With this study, we realized that ROS services cannot be directly be mapped to Rebeca semantics. However, it can be mapped via Rebeca modelling logic. This means that we define a flag (variable), which can take values of boolean 'true' or 'false'. Until the service is available for the client node, the flag status shall be set as false and with flag status being false, next process is not allowed to be executed. Thus, in effect the initial process which requested for service is getting blocked until it gets the response.

In the Listing 1, Node "add_two_ints_Server" is the server node and "add_two_ints_client" is the client node. This particular server node serves with adding two integers and sending it to the client node. Once the sum is received, the value is incremented once. In ROS, this node implementation can be explained as follows. The first four lines of the code includes the header files which are included from .srv file. Header file contains code layout, pieces of procedural code and forward declarations. This step is required in ROS since header file contains code layout, pieces of procedural code and forward declarations. Line number 3 and 4 defines a function 'add', whose input is a request and response type is defined in .srv file. The response type in this case is boolean. Further, line number 7 stores the sum of two integers a and b, which have been requested to be added and the added value is stored as response. Line number 8 and 9 are for displaying the request and response messages respectively. The next step is to initialize the service node, line number 13 is the syntax for the same. Initialization step includes declaring the node name, creation of node handle and creating service and advertise it over ROS (line number 18).

```
1 #include "ros/ros.h"
2 #include "beginner_tutorials/AddTwoInts.h"
3 bool add(beginner_tutorials::AddTwoInts::Request &req,
4 beginner_tutorials::AddTwoInts::Response &res)
5
6 {
7     res.sum = req.a + req.b;
8     ROS_INFO("request: x=%ld, y=%ld", (long int)req.a, (long int)req.b);
9     ROS_INFO("sending back response: [%ld]", (long int)res.sum);
10    return true;
```

```

11 }
12
13 int main(int argc, char **argv)
14 {
15     ros::init(argc, argv, "add_two_ints_server");
16     ros::NodeHandle n;
17     ros::ServiceServer service = n.advertiseService("add_two_ints", add);
18     ROS_INFO("Ready to add two ints.");
19     ros::spin();
20     return 0;
21 }

```

Listing 1: A example for Service Server

The client node requests for the service of sending the 'sum of integers', ROS implementation for client node shown in Listing 2 can be explained as follows. Line number 1 to 3 is statements for including header files. This is followed by main function where the client node is initialized (line number 6), a validity check statement (line number 7) and a display statement (line number 9). A node handle is created in line number 12 and a client is created for the `add_two_ints` service (line number 13). The `ros::ServiceClient` object is used to call the service when required. In line number 14, 15, and 16 an auto generated service class is instantiated and assign values into its request member. A service class contains two members, request and response. It also contains two class definitions which are request and response. The line number 17 calls the service. Since service calls are blocking in nature, it shall be served once the call is done. If the service call succeeded, `call()` will return true and the value in `srv.response` will be valid. If the call did not succeed, `call()` will return false and the value in `srv.response` will be invalid. Once the client node received the service, line number 26 executes, that is to increment sum value once. Line 27 is blocked until this node receives the service 'sum'.

```

1 #include "ros/ros.h"
2 #include "beginner_tutorials/AddTwoInts.h"
3 #include <cstdlib>
4 int main(int argc, char **argv)
5 {
6     ros::init(argc, argv, "add_two_ints_client");
7     if (argc != 3)
8     {
9         ROS_INFO("usage: add_two_ints_client X Y");
10        return 1;
11    }
12    ros::NodeHandle n;
13    ros::ServiceClient client =
14        n.serviceClient<beginner_tutorials::AddTwoInts>("add_two_ints");
15    beginner_tutorials::AddTwoInts srv;
16    srv.request.a = atoll(argv[1]);
17    srv.request.b = atoll(argv[2]);
18    if (client.call(srv))
19    {
20        ROS_INFO("Sum: %ld", (long int) srv.response.sum);
21    }
22    else
23    {
24        ROS_ERROR("Failed to call service add_two_ints");
25        return 1;
26    }
27    sum = sum ++;
28    return 0;
29 }

```

Listing 2: A example for Client Server

In Listing 3, Rebeca model for service communication represented. Here, from the initialization part of ROS code, we derive node names for service server node and client server node and map them as actors. These two actors communicates each other to model the request - response communication process. Thus, client server shall be the known rebec for service server and vice versa. `add_two_ints_client` and `add_two_ints_server` are the client and server nodes respectively. Library functions and display functions used in ROS code are abstracted in Rebeca model.

Model represents the information that a service request has been sent from client node to server node. The service request is modelled using message servers in Rebeca, and serving of service is also modelled using message servers with value of sum send as input parameter from service message server (`msgsrv add()`) to client message server (`msgsrv increment()`). However, in Rebeca message sending cannot be blocked. This means that if there is a delay in serving the client node, the process will not wait until the service event occurs. It will immediately execute the next process. Hence, to model the synchronous communication of ROS services, we have derived a logic for modelling ROS services. When the service is not available, the client server shall send messages to itself to the point until when the service is available. Thus blocking nature of services is modelled without doing anything except checking the availability of service. The while loop (line 21) represents various tasks in the client server. The execution of ‘while loop’ is allowed only until the service is available. In Rebeca model, significant functionalities of the real system is modelled and other details are abstracted.

```

1 reactiveclass add_two_ints_client (10){
2   knownrebecs
3   {add_two_ints_server addserver ;}
4   statevars {
5     boolean isBlocking;}
6   add_two_ints_client ( )
7   { isBlocking = true;
8     self.checkServiceAvailability(); }
9   msgsrv checkServiceAvailability(){
10    addserver.isServiceAvailable();
11  }
12  msgsrv serviceIsAvailable()
13  {
14    isBlocking = false;
15    self.addService();
16  }
17  msgsrv serviceIsNotAvailable(){
18    checkServiceAvailability();
19  }
20  msgsrv addService( )
21  {while(isBlocking){ }
22    addserver.add();}
23  msgsrv increment ( )
24  {while(isBlocking){}
25    sum =sum++;
26  }}
27 reactiveclass add_two_ints_server (10) {
28   knownrebecs {
29     add_two_ints_client addclient;}
30   statevars {
31     int a ;
32     int b ;
33     int sum ;}
34   add_two_ints_server ( )
35   { a=0;
36     b=0;
37     sum=0;
38   }
39   msgsrv add ( ) {
40     sum =a+b ;
41     addserver.increment (sum) ;}
42   msgsrv isServiceAvailable(){
43     addclient.serviceIsAvailable();}
44   main {
45     add_two_ints_server addserver ( addclient ) : ( ) ;
46     add_two_ints_client addclient ( addserver ) : ( ) ;
47   }

```

Listing 3: Rebeca model for ROS service communication

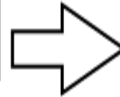
Thus, a ROS service can be mapped to Rebeca model as a derived pattern with following steps. Identifying the nodes participating in the communication, the service functions performed by nodes, modelling of blocking tasks in ROS using boolean variables in Rebeca. Fig30 shows the mapping

ROS Service Mechanism**Node "add_two_ints_server"**

```
#include "ros/ros.h"
#include "beginner_tutorials/AddTwoInts.h"
bool add(beginner_tutorials::AddTwoInts::Request &req,
beginner_tutorials::AddTwoInts::Response &res)
{ res.sum = req.a + req.b;
  ROS_INFO("request: x=%ld,y=%ld",
  (long int)req.a, (long int)req.b);
  ROS_INFO("sending back response: [%ld]", (long int)res.sum);
  return true;
} int main(int argc, char **argv)
{ ros::init(argc, argv, "add_two_ints_server");
  ros::NodeHandle n;
  ros::ServiceServer service =
  n.advertiseService("add_two_ints", add);
  ROS_INFO("Ready to add two ints.");
  ros::spin();
  return 0;
}
```

Node "add_two_ints_client"

```
#include "ros/ros.h"
#include "beginner_tutorials/AddTwoInts.h"
#include <cstdlib>
int main(int argc, char **argv)
{ros::init(argc, argv, "add_two_ints_client");
  if (argc != 3)
  {ROS_INFO("usage: add_two_ints_client X Y");
  return 1;
  } ros::NodeHandle n;
  ros::ServiceClient client = n.serviceClient<beginner_tutorials
  ::AddTwoInts>("add_two_ints");
  beginner_tutorials::AddTwoInts srv;
  srv.request.a = atoll(argv[1]);
  srv.request.b = atoll(argv[2]);
  if (client.call(srv))
  {ROS_INFO("Sum: %ld", (long int)srv.response.sum);
  }else
  {ROS_ERROR("Failed to call service add_two_ints");
  return 1;
  }
  sum =sum ++;
  return 0;
}
```

**Rebeca Model for ROS Service Mechanism**

```
reactiveclass add_two_ints_client (10){
  knownrebecs {add_two_ints_server addserver;}
  statevars {boolean isBlocking;}
  add_two_ints_client ( )
  { isBlocking = true;
    self.checkServiceAvailability();
  }
  msgsrv checkServiceAvailability(){
    addserver.isServiceAvailable();}
  msgsrv serviceIsAvailable()
  { isBlocking = false;
    self.addService();
  }
  msgsrv serviceIsNotAvailable(){
    isBlocking = true;
    checkServiceAvailability();}
  msgsrv addService ( )
  { while(isBlocking){
    addserver.add();
  }
  msgsrv increment ( )
  { while(isBlocking)
    }sum =sum++;}
  reactiveclass add_two_ints_server (10) {
  knownrebecs {
  add_two_ints_client addclient;}
  statevars {
    int a ;
    int b ;
    int sum ;}
  add_two_ints_server ( )
  { a=0;
    b=0;
    sum=0;}
  msgsrv add ( )
  { sum =a+b ;
  addserver.increment (sum) ;}
  msgsrv isServiceAvailable(){
  addclient.serviceIsAvailable();}
  main {add_two_ints_server addserver
  ( addclient ) : ( ) ;
  add_two_ints_client addclient
  ( addserver ) : ( ) ;
  }
}
```

Figure 30: Pattern Mapping for ROS Service Mechanism

pattern representation for ROS Service to Rebeca Service mechanism.

3.5.10 ROS Topics

ROS Topics are communication channels through which messages are sent from one node to another via publish- subscriber format. A topic act as a bus through which messages flow between nodes. Figure 31 shows the pattern mapping from ROS to Rebeca for publish-subscribe communication scheme. The ROS code given in this example represents two nodes- “talker” and “listener” and a topic chatter through which messages are published from node talker to node listener. ROS implementation of talker node can be explained as shown in Listing 4. Line number 6, is an initialization function initialized the input arguments and name of the node. Line 7 defines the node handle which is the main access point of node to communicate with ROS system. Line 8 defines advertise() function which helps to publish messages on a topic. This function registers at master node and the subscriber node who is subscribed to this topic. The second parameter of advertise function is the message queue size of the publishing messages through this topic. Line 10 defines the variable count to track the number of messages sent by this node. Line 13 defines message object ‘msg’, in which data is stuffed and then it is published. Line 18 is the publish function, using which messages are sent.

```

1  #include "ros/ros.h"
2  #include "std_msgs/String.h"
3  #include <sstream>
4  int main(int argc, char **argv)
5  {
6  ros::init(argc, argv, "talker");
7  ros::NodeHandle n;
8  ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);
9  ros::Rate loop_rate(10);
10 int count = 0;
11 while (ros::ok())
12 {
13 std_msgs::String msg;
14 std::stringstream ss;
15 ss << "hello world " << count;
16 msg.data = ss.str();
17 ROS_INFO("%s", msg.data.c_str());
18 chatter_pub.publish(msg);
19 ros::spinOnce();
20 loop_rate.sleep();
21 ++count;
22 }
23 return 0;
24 }

```

Listing 4: An example for ROS publisher node

Similarly ROS implementation for Listener Node can be defined as shown in Listing 5. Line 1 and 2 include header files which is required to use in the code so as to access general ROS system components. Line 3 defines the call back function, which is called whenever messages are received on subscribed topic. Line 11 defines subscribe function with topic name and call back function name. Line12 is a ROS loop function which is used for calling message callbacks as fast as possible.

```

1  #include "ros/ros.h"
2  #include "std_msgs/String.h"
3  void chatterCallback(const std_msgs::String::ConstPtr& msg)
4  {
5  ROS_INFO("I heard: [%s]", msg->data.c_str());
6  }
7  int main(int argc, char **argv)
8  {
9  ros::init(argc, argv, "listener");
10 ros::NodeHandle n;
11 ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);
12 ros::spin();
13 return 0;

```


Listing 5: An example for ROS subscriber node

Figure 31 shows the pattern mapping from ROS to Rebeca for publish-subscribe communication scheme. The model is given in Listing 6. In Rebeca, the two nodes 'talker' and 'listener' are modelled as actors. Now modelling of topic were initially designed to be modelled as message servers. However, in case of one node being subscribed to more than one topic or different type of messages are published through the topic; this design may not serve the purpose. During the study, a better design was derived that topic is also modelled as an actor and this helps to establish communication between talker, listener and the topic by asynchronous message sending. Rebeca model can be explained as follows. In the model, 'Talker' node will be publishing messages continuously at a rate of 10 time units and this rate is set as environment variable, 'rate'. Since the 'Listener' had subscribed to this message publishing channel called 'Topic', it receives all those information which are getting published.

In Rebeca model, every time messages are published from talker, the message server 'chatter' is invoked which will set a variable 'messagesent' to be true. Once this variable is true the actor corresponding to listener node is called. At listener, we set variable messageReceived = true, which indicates that messages published from talker has finally received at listener end.

```

env byte rate=10;
reactiveclass Talker(10)
{
knownrebecs{

Listener listener;
Topic chatter;}

statevars{
boolean message;}

Talker(){
message=false;
self.talk();}
msgsrv talk(){
message=true;
chatter.sendmsg(message);
message=false;
self.talk () after (rate);}
}

reactiveclass Topic (10){
knownrebecs{
Talker talker;
Listener listener;}

statevars{
boolean messagesent;}

Topic(boolean message)
{messagesent=message;
}
msgsrv sendmsg()
{
if (messagesent ==true)
listener.subscribe();
else
(messagesent=false;
}
}

reactiveclass Listener (10){
knownrebecs {
Talker talker;
Topic chatter;}

```

```
statevars {
boolean messageReceived ;}

Listener(){
messageReceived=false;
}
msgsrv subscribe()
{
messageReceived =true;
delay(rate);
messageReceived =false;
}
}

main
{
Talker talker(chatter,listener):();
Topic chatter (talker,listener):(boolean message);
Listener listener(chatter,talker):();
}
```

Listing 6: Rebeca model for ROS Topic communication

3.5.11 ROS Parameter Server

In ROS parameter server is used to store and retrieve parameters during run-time. This facility in ROS for usage of static data such as configuration parameter management. It can be defined globally by all the nodes, tools and libraries and thus configuration state of the system can access it. In Rebeca, we are not modelling any run-time behaviour of the system and there is no configuration file to be updated for Rebeca models. Hence mapping of this ROS pattern is not applicable in Rebeca.

ROS Topic Mechanism**Node "Talker"**

```

#include "ros/ros.h"
#include "std_msgs/String.h"
#include <sstream>
int main(int argc, char **argv)
{
  ros::init(argc, argv, "talker");
  ros::NodeHandle n;
  ros::Publisher chatter_pub =
  n.advertise<std_msgs::String>("chatter", 1000);
  ros::Rate loop_rate(10);
  int count = 0;
  while (ros::ok())
  {
    std_msgs::String msg;
    std::stringstream ss;
    ss << "hello world " << count;
    msg.data = ss.str();
    ROS_INFO("%s", msg.data.c_str());
    chatter_pub.publish(msg);
    ros::spinOnce();
    loop_rate.sleep();
    ++count;
  }
  return 0;
}

```

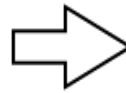
Node "Listener"

```

#include "ros/ros.h"
#include "std_msgs/String.h"
void chatterCallback(const std_msgs::String::ConstPtr&
msg)
{
  ROS_INFO("I heard: [%s]", msg->data.c_str());
}
int main(int argc, char **argv)
{
  ros::init(argc, argv, "listener");
  ros::NodeHandle n;
  ros::Subscriber sub = n.subscribe("chatter", 1000,
chatterCallback);
  ros::spin();

  return 0;
}

```

**Rebeca Model for ROS Topic Mechanism**

```

env byte rate=10;
reactiveclass Talker(10){
  knownrebecs{
    Listener listener;
    Topic chatter;}
  statevars{
    boolean message;}
  Talker(){
    message=false;
    self.talk();}
  msgsrv talk(){
    message=true;
    chatter.sendmsg(message);
    message=false;
    self.talk () after (rate);}
}
reactiveclass Topic (10){
  knownrebecs{
    Talker talker;
    Listener listener;}
  statevars{
    boolean messagesent;}
  Topic(boolean message)
  {messagesent=message;}
  msgsrv sendmsg(){
    if (messagesent ==true)
    listener.subscribe();
    else
    (messagesent=false;
    )}
}
reactiveclass Listener (10){
  knownrebecs {
    Talker talker;
    Topic chatter;}
  statevars {
    boolean messageReceived ;}
  Listener(){
    messageReceived=false;}
  msgsrv subscribe(){
    messageReceived =true;
    delay(rate);
    messageReceived =false;
  }
}
main
{
  Talker talker(chatter,listener):();
  Topic chatter (talker,listener):(boolean message);
  Listener listener(chatter,talker):();
}

```

Figure 31: Pattern Mapping for ROS Publish-Subscribe Mechanism

3.6 A Pseudo Code for transforming ROS code to Rebeca Model

In this section, we present a pseudo code or an algorithm to transform any ROS code to Rebeca model. It can be considered as a reference procedure for this transformation. This algorithm aims to extract relevant information from ROS code such that it can be used as input information for

developing the Rebeca model.

Algorithm: *Mapping Pattern Generation*

```

Input: ROS code of the application
Begin :

1:   If (Identify src , srv and msg folders in the source code)
2:     Open .cpp , .srv and .msg files
3:     If (Identify interface message folder /application folders)
4:       Open associated files to understand implementation
5:     Else do nothing

6:     For (i =1 to i, where i is the number of .cpp files in src folder)

7:       a. Identify the nodes from .cpp files
8:       b. Identify the variables used, its data types and their values
9:       c. Identify member functions and their functionalities
10:      d. Identify those nodes which are communicating each other

11:   Define & Initialize Rebeca main class with actors identified from step 3.a
12:   Define message queue size , statevariables and known rebecs for each actor
13:   Define message servers for Member functions identified from step 3

14:   If (distributed parameters identified in step 2)
15:     Define environment variables in Rebeca corresponding to parameters.
16:   Else do nothing

17:   For (i =1 to j, where j is the number of .srv files in src folder)

18:     a. Identify the services defined with each node
19:     b. Identify the purpose of the service and nodes involved
20:     c. Identify any blocking of processes due to the service

21:   Define each service as message servers

22:   If (services are blocking any process)
23:     Define and initialize flag variables and message servers
24:   Else
25:     Define services using message servers

26:   For (i =1 to k, where k is the number of messages in msg folder)

27:     a. Identify the messages used by each nodes
28:     b. Identify the type of messages sharing through each topic

29:   If ((m > 1 for a node, where m is number of topic)|| (msg data type > 1)]
30:     Design Topic as an actor .
31:   Else
32:     Design the Topic message as argument passing between the actors
33:   Else
34:     ROS source code not found , mapping cannot be performed

: End

```

3.6.1 Pseudo code Description

We derived a general procedure for transforming ROS code to Rebeca models and this procedure is reusable for mapping of any ROS code to Rebeca. The details of our reverse mapping procedure is shown in Algorithm 'Mapping Pattern Generation'. The main loop if algorithm, which is from Line 1 to Line 5 searches for appropriate folders in the ROS application package. In ROS, elements are modularised in folders. The relevant ROS code required for developing its model are application source folder, service folder and messages folder. Once this is found, application source files (.cpp), services used (as .srv files), messages used (as .msg files) are searched. In addition to this, source code for any interface messages (specific to application) are also searched if there exist any. From Line 6, an iteration loop starts to identify nodes from source code (.cpp files). During this itera-

tion, specification for each node is also identified such as variables used, their data types, value of variables, member functions used in the node, their functionalities, connected nodes etc. At the end of this iteration first level of Rebeca model development can be done. As given from Line 11 to 13, Rebeca actors/ Rebecs are defined corresponding to each identified nodes. The connected nodes in ROS shall be defined as known Rebecs. Message queue size and state variables for each actors is defined by model developer. Variables can also be defined for an actor by passing them as arguments from main class. Similarly, identified member functions are modelled as message servers in Rebeca. Message servers are designed as abstracted functions without any computation or display functions.

However, the model developer can make changes in model design so as to suite the real system. For example, not all ROS nodes needs to be a Rebec or not all member functions has to be modelled as message server. There is no exact procedure for this mapping process. The developer needs to review and ensure that the model reflects abstract representation of real system. Line 14 to 16 states that distributed parameters used in ROS system can be modelled with environment variables. Now the focus is to establish communication in the model. Line 17 to 20 shows an iteration to identify services from each .srv file. Also any service which can result in blocking a process is identified during this iteration. Now, corresponding to each service, message servers are defined in Rebeca model. If a service is blocking any process, this can be modelled using a boolean flag variable whose status can be set to represent this blocking status. This is given from Line 22-25. Communication in Rebeca is non preemptive whereas, communication via services can be preemptive in ROS. Similar to .srv files, messages defined in each .msg file is iterated to understand their types and topics used to send these messages. Line 26 -28 describes this iteration. Line 29-32 states that if any node uses more than one topic or if several message types are sent, each topic shall be defined as an actor. Otherwise the messages published through topics can be mapped as passing arguments between actors. Line 33 is the else condition for *if* loop given in line 1. I.e Rebeca model cannot be developed if src folder, srv folder and msg folders are not found for the application package.

SI No	ROS Terminology	Mappable	Direct Mappable or Not (Y/N)	Rebeca Terminology
1	Nodes	Yes	Yes	Rebecs (Actors)
2	Local Variables	Yes	Yes	State Variables
3	Shared Variables	No	--	--
4	Message Passing	Yes	No	Known Rebecs
5	Member Functions	Yes	Yes	Message Servers
6	Display/Print Functions	No	--	--
7	Topic	Yes	No	Derived Pattern
8	Service	Yes	No	Derived Pattern
9	ROS Bags	No	--	--
10	Distributed Parameter System	Yes	No	Environmental variables

Figure 32: An overview of Mapping Patterns between ROS and Rebeca

4 Electric Site Fleet Management Operation

In this section, modelling and verification of electric site operational behaviour level is presented. Electric site system can be considered as a concurrent system with several autonomous machines operating at the same time. The system is designed in such way that these machines are able to carry out their tasks without colliding with each other and meeting their deadlines. This system is modelled in Timed Rebeca, since the fleet management operation design requirement includes timing constraints. This model was developed based on the electric site fleet design. Timing constraints and real measurement values used for this model were extracted from expert knowledge. In other words, this model is not evolved out from a ROS to Rebeca transformation process.

This process consists of several steps as given in Figure 33. In this section, each of these steps are described as subsections.

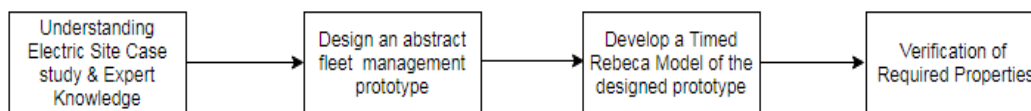


Figure 33: An Overview of Fleet Management Model Development

4.1 Understanding Electric Site Case study

In this section, we describe the case study of Electric Site fleet management. Electric Site project involves operating an electrified quarry site where materials are transported from one place to another. Electric Site fleet management is a prototype for managing a fleet of HX machines navigating autonomously and carrying out predefined tasks at the quarry site. This prototype gives emphasis for a non colliding movement of HX machines, deadlock freedom during site operation, no deadline miss scenario and in general schedulability of assigned task execution. In this prototype, HX machines are intended to work in a fleet manner for performing tasks such as material transport, loading unloading, charging etc. in a cyclic manner. The entire process can be described as follows: The materials primarily demolished at a quarry site are loaded into a Primary Crusher (PC) where they are crushed and remnants are loaded. The materials are then transported and unloaded to Secondary Crusher (SC). Machines are also loaded using a Wheel Loader from previously stocked pile of materials. HX machines are used for transporting activities and unloading. In this particular scenario, individual machines are assigned for unique task like getting loaded at the wheel loader or at PC, unloading at SC, charging at charge stations, navigating or in queue up state. The machines are unloaded at secondary crusher point. There is a charge station consisting of two chargers for fueling up the machine during operation and every returning machines from unloading point shall be charged to full battery. Figure 34 shows a pictorial representation of this process.

Details obtained from design diagram of fleet management prototype are as follows. The Beginning of the operation can be considered from that point where machines are queued up at 'Decision Point'. At this point, a decision is taken whether the first machine in the queue is moving to PC or Wheel loader. This decision is based on two conditions - 1) PC is in operation and 2) There shall be two machines assigned for task execution at PC at any time. Thus, condition for machines to move towards wheel loader are 1) PC is not in operation or 2) There is already two machines assigned for task execution at PC. Note that the HX machines and Wheel loader are mobile points. We verified in design architecture model that each HX machine has a right knowledge of wheel loader's current position. The materials loaded by HX machines at PC or at wheel loader will be transported to SC. The path segment through which this transportation takes place is a collision prone area, since machines from PC and wheel loader locations tends to meet together. Thus,

a critical section is designed at this path and only one machine is allowed to pass through this section. Also there is control on machine speed and also a safe distance is maintained by each HX machines during the navigation. At SC, materials are unloaded and further machines are designed to move towards charge stations. There are two charge stations, where a single machine could be charged at a time. Thus until a charge station is available, HX machines queue up waiting for its turn.

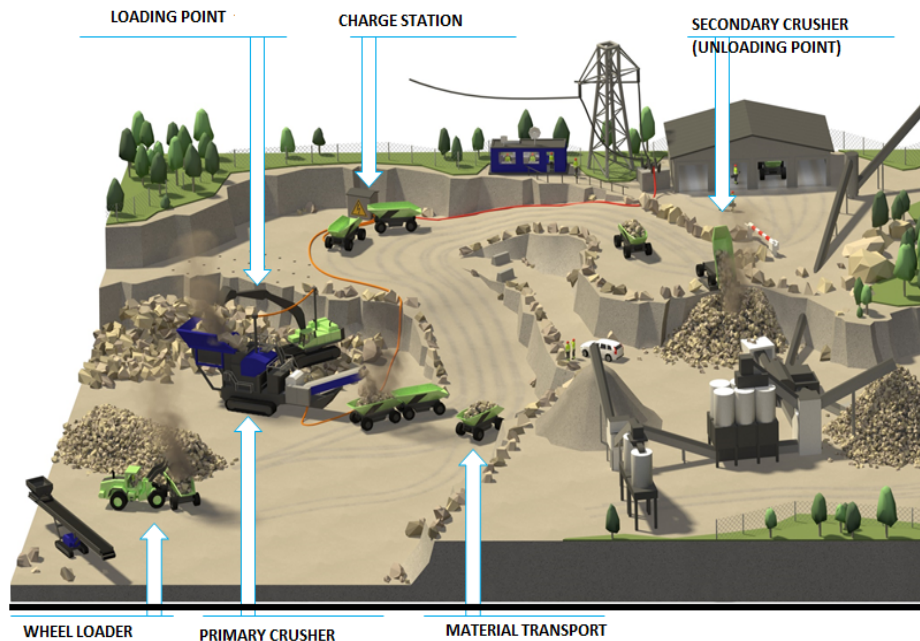


Figure 34: Fleet Operation Prototype

4.2 Abstracted Fleet Management Prototype

An abstracted and generic representation of fleet management prototype was developed based on design diagram as shown in Figure 35. The fleet path is divided into seven segments from S1 to S7. Each segment has predefined lengths and speed profiles assigned to it. Such fleet management of machines can be found under several applications such as aviation machinery, large scale mining operations, vehicle telematics etc. where efficiency and productivity can be increased. This is a generic design design of fleet management and it can be scaled up for various configurations or specific requirements. The major points to be considered for modelling of this design are as below:

- Number of machines participating in fleet operation - There shall be an optimum number of machines with which the entire operation can be scheduled within the designed area.
- Critical sections of the path - These are the intersecting points of two path segments during operation which is one of the collision prone areas. Thus, only one machine shall be allowed within this critical area.
- Decision points at fleet path - There are predefined points in the fleet path where certain decision are taken to proceed with the operation. These decisions are based on specific conditions of fleet requirement.
- Queue up Points - These are the points where machines queue up due to blocked processes such as crossing of critical sections, unavailability of charge stations etc. One design criteria shall be to minimize the queue up frequency.

In Figure 35, a generic example of fleet management operation is shown including the above design characteristics. The events mentioned in the figure can represent various resources such as primary crusher, secondary crusher, charge stations etc. as shown in Figure 34. We assume that the operation starts at Event 4, from which segment S4 leads to a decision point. From this decision point, machines can move towards Event 1 or Event 2 through segment S5 and S6 respectively. Once the process is executed at Event 1 and Event 2, machines shall proceed to a queue up point at the start of segment S2. However, the machines reaching to this point from Event 1 shall cross a critical section where segments S6 and S7 crosses each other. At the queue up point, machines wait for their turn to process at Event 3 and returns to the starting position (Event 4).

In this scenario, Event 4 can be considered as a charge station from which machines are moving out with full battery charge. Segment S4 can be defined as a high speed profile since the chances of collision and queue up is less. S4 leads to a decision point, from which machines can move towards two different loading points. Thus Event 1 and Event 2 could be loading process events. Event 3 could be an unloading spot where materials loaded from Event 1 and 2 are dumped. Thus segments connecting Event 1, Event 2 and Event 3 are high traffic area and hence the corresponding segments has to be defined with reduced speed along with maintaining safe distances between machines. Similar to S4, Segment S3 can also be defined as a high speed profile segment.

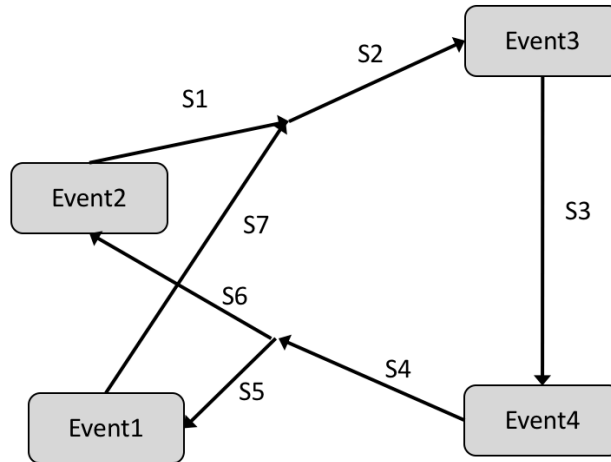


Figure 35: Fleet Path for Operation Procedure

4.3 Timed Rebeca Model of Fleet management Prototype

In this Timed Rebeca model, we verify the collision avoidance property of HX machines. The modelling logic can be explained as follows: Every segment in the fleet is divided into few sub-segments. Each sub-segment has predefined length and speed profiles. Each navigating machine arriving at a sub-segment shall request for permission acknowledgement in order to proceed to the next sub-segment. If the preceding sub-segment is free, i.e. when it is available without a machine occupancy, the requested machine gets permission signal and is able to exit the current sub-segment and enters the new one. Otherwise, the machine stops at a “safe distance” from the next sub-segment until it gets the permission. Since we assure the existence of one machine inside a sub-segment, there is no need to check the collision while a machine is traveling inside a sub-segment. There are a few parameters in the model that are used in this model development, which are as given below:

- **RESENDING PERIOD:** The machine which is waiting for entering into new sub-segment, re-sends the request after elapsing a specified time, which is termed as ‘resending period’.

- **NORMAL SPEED:** Machines are designed to be navigate with specific speed profiles. Normal speed is the speed profile for most of the times.
- **REDUCED SPEED:** In a few segments, machines have navigate with a lesser speed profile, which is termed as 'reduced speed'.
- **SAFE DISTANCE:** Vehicles must be in a safe distance from each other. When a vehicle wants to exit its current sub-segment, it must stop in a safe distance from the next sub-segment as there may be another vehicle at the beginning of the next sub-segment.
- **SEGMENT LENGTH:** the length of each sub-segment
- **PROCESS TIME 1 :** The process time at Event 1
- **PROCESS TIME 2 :** The process time at Event 2
- **PROCESS TIME 3:** The process time at Event 3
- **PROCESS TIME 4:** The process time at Event
- **NUMBER VEHICLES:** The number of vehicles in the system

For developing Timed Rebeca model, the above variables has been specified with certain samples values which are as follows. Normal speed and reduced speed are defined to be as 30 km/hr and 10 km/hr respectively. The entire fleet distance is 1000 meters with each connecting segments to be 200 meters. Process time at Event 1 and 2 are 60 seconds each and that at Event 3 is 30 seconds. Event 4 has a processing time for 60 seconds. The number of vehicles has been defined as four. The model aims to verify the schedulability of fleet management operation and to verify collision free property of the fleet design with above characteristics.

Figure 36 shows the modelling logic representation for fleet management timed Rebeca model. Each resource points and the segments are represented as an actor in the model. I.e each Event and each segment shall be mapped as a Rebec and number of machines are modelled as messages which are being sent between Rebecs. Actor segment contains several sub-segments defined in its class which sends messages to other actors such as Event 1, Event 2, Event 3 and Event 4. These messages are permission requesting or permission grant acknowledgements and they are modelled as boolean variables. Path segment representing critical section allows the navigation of only one machine at a time. Speed values for each segment and safe distance are defined in the model.

In the model, all the predefined times and values such as resending period, normal speed, reduced speed, safe distance, segment length, loading-unloading duration, charging time and number of machines are defined as environment variables. Every machine has a machineID and permission acknowledgement is send with the reference of this ID. Status of each task execution is modelled using boolean variables in corresponding actor class. If a machine is not allowed to move forward, it is modelled as resending permission for that machine to move forward after a wait time. A assertion (false) statement is included to verify the schedulability and non collision property of model execution after one cycle of operation. Once the model checking reaches assertion (false) statement, the process stops and state space is generated for the performed model checking cycle.

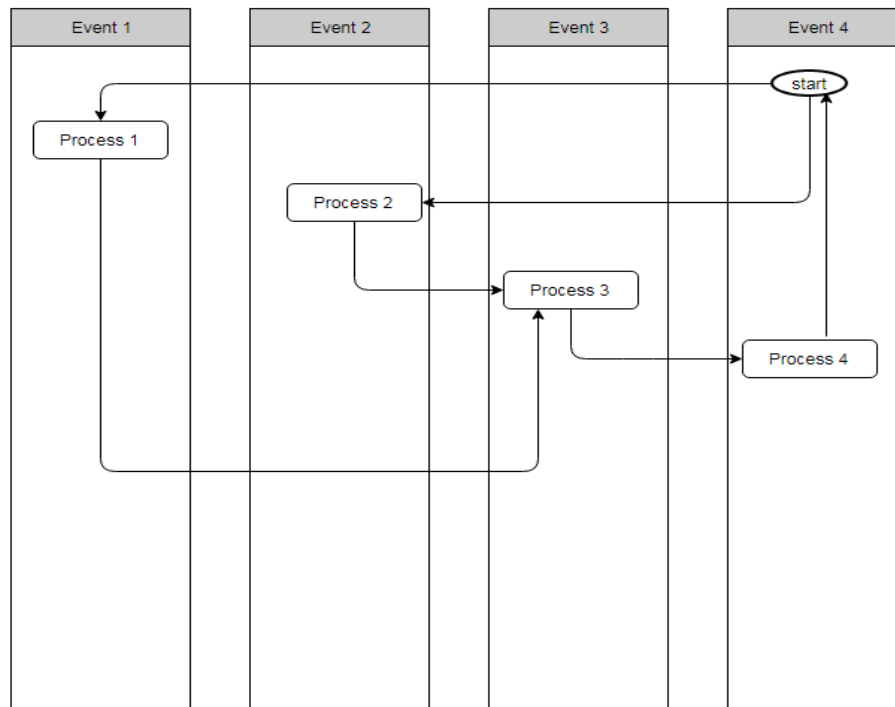


Figure 36: Representation fleet management model

4.4 Verification of Fleet Management Model and Results

In Fleet Management Model, we have modelled the scenario of fleet operation of HX system with timing constraints, distance parameters, fleet size and segment speed profiles of real system. We are modelling and verifying the concurrent behaviour of fleet operation prototype. Below are the properties which shall be evaluated with fleet model.

- **Schedulability**

Schedulability can be explained as a state where all the assigned tasks and missions were executed successfully by every machine in the fleet while satisfying the specified constraints. Since we are modelling the prototype with real system parameters such as segment distance, time duration for resource operation, speed profiles, number of machines etc. a satisfied verification result gives an assurance that the real system shall also be schedulable.

- **Reachability**

Reachability is a condition where all the states in the system are reachable or there are no states which are held as blocked or unavailable. Reachability problem is a common issue in concurrent system. This is because when a waiting state is held off due to a blocking resource. Reachability is a default property verification in Rebeca.

- **No-Deadlock Detection**

Deadlock is not a preferred condition in any concurrent system, where the system execution does not progress beyond a locked state. While modelling the system with non deterministic logic, the model shall be carefully developed. Since the HX systems are autonomous, deadlock detection is one criteria to be evaluated.

- **No-Deadline miss Scenario**

Even though this system doesnot specify any hard deadlines like in real time systems, deadline can be considered as meeting the specified conditions at each resource point. Using routing algorithms in asynchronous system can potentially introduce deadlocks, however the system takes care of it by restricting path diversity or by using extra resources. This shall be verified.

- **Collision free Navigation**

Generally model checking verifies static scenarios. However, by abstract interpretation we are verifying the operational system behaviour. I.e to check if there is any possibility for a collision in the system prototype. The collision prone areas are guarded with critical sections and gateways in the system. Also safe distance is maintained between machines to avoid hitting each other. This property shall also be verified.

Schedulability and collision free properties are verified in the model by its design. Dead lock freedom and Non Deadline miss are verified in Afra model checker by default. Once the complete model of the fleet management has been created, the Afra model checking tool verifies whether the schedulability properties hold in all reachable states of the system. A counterexample will be produced if there are any deadline violations which denotes the trace of states that resulted in violation. In the model, state space generated at the end of each cycle of operation is analyzed to see the current state and transitions undergone by each resource. Also the total time taken to complete one cycle of operation is included in the result. In the verified result, checked properties lists as below:

```
< model - checking -report>
< system - info>
  <total-spent-time>1</total-spent-time>
  <reached-states>572</reached-states>
  <reached-transitions>680</reached-transitions>
  <consumed-mem>125840</consumed-mem>
</system-info>
<checked-property>
  1  <type> Reachability</type>
  2  <name> Deadlock-Freedom and No Deadline Missed</name>
  3  <result> assertion Failed </result>
  4  <message> no message </message>
```

Line 1 denotes that the reachability properties are satisfied by the model. Line 2 of the result states that there is no deadlock and no deadline miss possibilities for this model operation. We have specified 'assertion(false)' statements in the model so that the model execution stops and exits out from model execution process. Hence, Line 3 of the result shows that model checking has been stopped and exited after 'assertion(false)' statement which shows that schedulability and non-collision property were satisfied for this one cycle of model execution.

The model checking report also shows that there are a total of 572 states and 680 state transitions possible with this model. The verified properties are assured to be satisfied in all these states. Obtaining such a verification assurance is a tedious and time consuming using functional testing.

5 Conclusion and Future Work

5.1 Summary

In this thesis, we explored the potential of using Rebeca Modelling language to analyze a distributed concurrent complex system. Rebeca is an actor based language supported by formal verification methods. It bridges the gap between real applications and formal verification. Rebeca models for this study were developed in Afra, which is the model checker tool supported by Rebeca. In this study, we analyzed concurrent operations of autonomous machines at an electrified quarry site using Rebeca models. Schedulability and non collision property of fleet management prototype was verified using these models. Since the operation involved certain timing parameters, the model was developed as timed rebeca model. Timed Rebeca model is an timed extension of Rebeca. Deadlock freedom and No-deadline miss properties are automatically verified in Afra. We also analyzed distributed architecture of the autonomous machines and verified its properties. The models for verifying distributed architecture were developed as Core Rebeca models, since we did not focus on timing constraints of it. This architecture was designed in a robotic framework termed 'Robot Operating System'(ROS) and hence the analysis involved a process of transformation from ROS to Rebeca. This process leads to derive certain mapping patterns between ROS and Rebeca which can be used for any ROS code. The transformation process has been structured into an algorithm which can be used for future scope.

The verification results obtained from the model checking process shows the number of state transitions which a complex system can undergo during operation. The properties that we verified were selected by referring to similar works done and based on opinions from experts. Properties of core rebeca models uses LTL specifications and their results assure safety during machine navigation. However, we have not considered all the aspects for covering safety assurance, but the properties verifies basic system functional requirements. Properties of timed rebeca model assures schedulability and collision free navigation of fleet management prototype. Nevertheless, there may be several factors for a complex system for resulting in an emergent behaviour. We are not looking into those areas in this study. This study primarily shows the ability of Rebeca modelling language in analyzing a ROS based distributed concurrent system and in verifying their properties.

5.2 Discussion

Merits of formal verification(FV) is popular and FV is a growing field for more than thirty years[30]. Using FV, we can prove or disprove a system design conformation to its requirements. Distinctly, an actor model is a concept to represent concurrent computation of a system. A combination of these two technologies can be a real benefit in validating real applications which are difficult to verify using other methods. Rebeca is an actor based modelling language which is supported by formal verification semantics and is suitable for verifying real time concurrent systems. Such modelling has several advantages. Firstly, the model can specify formal properties. A model can be true copy of the real system, however a modelled system is always deterministic. This means that for the same input given to a model, we get same outputs all the time. It cannot produce emergent characterizes which is possible in real system. In other words, a complete assertion is not possible with physical realization of a system. Hence a good model shall abstract relevant and significant features of the real system without increasing its complexity. Timed Rebeca helps to develop lower level of abstraction. A Timed Rebeca model developer can abstract the relevant functional features of a system together with the timing constraints for both computation and network latencies, and analyze the model from various points of view[51].

At this point of study, we are in a position to answer the research questions defined at the initial phase of the study. The major task of this thesis was to understand the ROS code, relevant abstraction of its features and transform it into Rebeca models. This process involves identification of executable features of ROS such as ROS nodes, ROS services, ROS topics and so on. The functionality and relation between these features needs to understood and to be abstracted to transform it into rebeca semantics. We found that most of the semantics in ROS can be accomplished in Rebeca model. However, certain features could not be transformed. Also computational details

and other implementation specific functionalities cannot be mapped towards Rebeca. Nevertheless, once this process is done, a procedure to perform this transformation can be derived. In other words, this process can be automated. Here automation refers to a standard algorithm which can be used for any ROS code to be transformed to Rebeca models. Another research question aimed at understanding the challenges involved in ROS to Rebeca mapping process. When mapping from a robotic framework to a formal model, there can be a lot of inconsistencies which needs to be solved. However, it is not an easy task to solve all of them. Thus the idea is to overcome these factors in a harmless way such that it won't affect any other part of the system/model. One major challenge was to find a balance in abstracting the details of real system when mapping to its model. The computations or calculations involved in the real system could be abstracted, but their outcome has to be included in the model. Also an extensive compositional verification can lead to huge state space generation and can lead to state space explosion.

Property verification is the core of model checking process. The properties to be verified are usually the desired system requirements. In this thesis, there was a challenge in identifying these properties. This is because desired system requirements were not known at the time of conducting the study. Hence we had to get inspiration from similar studies, where formal verification methods were used for analyzing a distributed system. Also we took a few hints from expert knowledge in identifying certain properties. The identified properties include safety and liveness related properties. Anyhow, in Rebeca we specify these properties using LTL statements and the verification result includes the state transitions undergone while model checking, total time taken for verifying the entire model and the verification result whether the property is satisfied or not. However, there are some limitations for this process. The success of model checking solely lies in quality of the developed model. We have not set any criteria for measuring the quality of the developed model. The only focus was on not abstracting any relevant feature of the real system.

5.3 Future Work

The approach taken in this study is 'Actor based formal modelling and verification'. It is one of the option to analyze a distributed concurrent system. A predefined extension for this work is to use another modelling tool called Ptolemy which is a powerful visual simulation tool. Thus a simulated environment of the case study could be obtained using Ptolemy modelling. Further, this study was a stepping stone to explore the potential of Rebeca in analyzing a distributed robotic system architecture. The results shows that Rebeca is a good reference tool for modelling systems having concurrent computation. This study could be extended to a step higher to define and investigate more design properties by manipulating the system architecture to generate more input data for enhancing the analysis.

Two different models were used for design architecture modelling in this study. This analysis can be made extensive by developing several models with further details. Similarly for modelling the fleet management operation, the fleet configuration can be modified by adding or removing resources. Also for the existing configuration, optimization of a significant parameter can be evaluated.

5.4 Validity Threat

For the model development, ROS to Rebeca transformation is done in this study. However, there is no performance criteria set to measure the quality of the models developed. In order to avoid a state space explosion, models are developed with balance of maintaining complexity of system state without abstracting significant features of real system. The verified properties are based on the model behaviour, however there may be several unknown factors which could affect the real system to show emergent behaviour. This is not covered in the models and also the results obtained from this study are not validated against the real system. Similarly, the amount of information derived from design architecture and level of abstraction made to the model is of utmost importance and can affect the results dramatically if administered in a different way.

References

- [1] “*Rebeca Formal Modeling Language*,” 2015, online visited = 2017-08-28. [Online]. Available: <http://www.rebeca-lang.org/wiki/pmwiki.php/Rebeca/Rebeca>
- [2] E. A. Lee, “*Cyber Physical Systems: Design Challenges*,” *11th IEEE Symposium on Object Oriented Real-Time Distributed Computing (ISORC)*, pp. 363–369, 2008.
- [3] P. Pourbeik, P. S. Kundur, and C. W. Taylor, “*The anatomy of a power grid blackout-Root causes and dynamics of recent major blackouts*,” *IEEE Power and Energy Magazine*, vol. 4, no. 5, pp. 22–29, 2006.
- [4] S. Ghosh, “*Trustguard: A containment architecture with verified output*,” Ph.D. dissertation, Princeton University, 2017.
- [5] T. Wongpiromsarn and R. M. Murray, “*Formal verification of an autonomous vehicle system*,” in *Conference on Decision and Control*, 2008.
- [6] S. Tasharofi, P. Dinges, and R. E. Johnson, “*Why do scala developers mix the actor model with other concurrency models?*” in *European Conference on Object-Oriented Programming*. Springer, 2013, pp. 302–326.
- [7] M. Sirjani and M. M. Jaghoori, “*Ten years of analyzing actors: Rebeca experience*,” in *Formal modeling*. Springer-Verlag, 2011, pp. 20–56.
- [8] J. C. Knight, “*Safety critical systems: challenges and directions*,” in *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*. IEEE, 2002, pp. 547–550.
- [9] 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*, vol. 3, no. 3.2. Kobe, 2009, p. 5.
- [10] J. F. Nunamaker Jr, M. Chen, and T. D. Purdin, “*Systems development in information systems research*,” *Journal of management information systems*, vol. 7, no. 3, pp. 89–106, 1990.
- [11] A. Böckenkamp, F. Weichert, J. Stenzel, and D. Lünsch, “*Towards autonomously navigating and cooperating vehicles in cyber-physical production systems*,” in *Machine Learning for Cyber Physical Systems: Selected papers from the International Conference ML4CPS 2015*. Springer, 2016, p. 111.
- [12] M. Bader, A. Richtsfeld, M. Suchi, G. Todoran, W. Holl, and M. Vincze, “*Balancing Centralised Control with Vehicle Autonomy in AGV Systems for Industrial Acceptance*,” in *11th Int. Conf. on Autonomic and Autonom. Sys*, 2015.
- [13] D. Weyns and T. Holvoet, “*Architectural design of a situated multiagent system for controlling automatic guided vehicles*,” *International Journal of Agent-Oriented Software Engineering*, vol. 2, no. 1, pp. 90–128, 2008.
- [14] D. González, J. Pérez, V. Milanés, and F. Nashashibi, “*A review of motion planning techniques for automated vehicles*,” *IEEE Transactions on Intelligent Transportation Systems*, vol. 17, no. 4, pp. 1135–1145, 2016.
- [15] P. Farahvash and T. O. Boucher, “*A multi-agent architecture for control of AGV systems*,” *Robotics and computer-Integrated manufacturing*, vol. 20, no. 6, pp. 473–483, 2004.
- [16] D. Weyns, T. Holvoet, K. Schelfhout, and J. Wielemans, “*Decentralized control of automatic guided vehicles: applying multi-agent systems in practice*,” in *Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*. ACM, 2008, pp. 663–674.

- [17] G. Antonini and M. Bierlaire, “*Capturing interactions in pedestrian walking behavior in a discrete choice framework*,” Tech. Rep., 2005.
- [18] S. Glaser, B. Vanholme, S. Mammari, D. Gruyer, and L. Nouveliere, “*Maneuver-based trajectory planning for highly autonomous vehicles on real road with traffic and driver interaction*,” *IEEE Transactions on Intelligent Transportation Systems*, vol. 11, no. 3, pp. 589–606, 2010.
- [19] J. Huang, C. Erdogan, Y. Zhang, B. Moore, Q. Luo, A. Sundaresan, and G. Rosu, “*ROSRV: Runtime verification for robots*,” in *International Conference on Runtime Verification*. Springer, 2014, pp. 247–254.
- [20] Z. Kootbally, S. Balakirsky, and A. Visser, “*Enabling codesharing in rescue simulation with usarsim/ros*,” in *Robot Soccer World Cup*. Springer, 2013, pp. 592–599.
- [21] B. B. Rhoades, J. P. Sabo, and J. M. Conrad, “*Enabling a National Instruments DaNI 2.0 robotic development platform for the Robot Operating System*,” in *SoutheastCon, 2017*. IEEE, 2017, pp. 1–5.
- [22] S. Cousins, “*Is ROS Good for Robotics?*” in *IEEE Robotics & Automation Magazine*, vol. 19, June 2012. [Online]. Available: <http://ieeexplore.ieee.org.ep.bib.mdh.se/document/6213236/>
- [23] S.-Y. Jeong, I.-J. Choi, Y.-J. Kim, Y.-M. Shin, J.-H. Han, G.-H. Jung, and K.-G. Kim, “*A Study on ROS Vulnerabilities and Countermeasure*,” in *Proceedings of the Companion of the 2017 ACM/IEEE International Conference on Human-Robot Interaction*. ACM, 2017, pp. 147–148.
- [24] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald, “*Formal methods: Practice and experience*,” *ACM computing surveys (CSUR)*, vol. 41, no. 4, p. 19, 2009.
- [25] K. Y. Rozier, “*Specification: The biggest bottleneck in formal methods and autonomy*,” in *Verified Software. Theories, Tools, and Experiments: 8th International Conference, VSTTE 2016, Toronto, ON, Canada, July 17–18, 2016, Revised Selected Papers 8*. Springer, 2016, pp. 8–26.
- [26] E. M. Clarke and J. M. Wing, “*Formal methods: State of the art and future directions*,” *ACM Computing Surveys (CSUR)*, vol. 28, no. 4, pp. 626–643, 1996.
- [27] P. Fonseca, K. Zhang, X. Wang, and A. Krishnamurthy, “*An Empirical Study on the Correctness of Formally Verified Distributed Systems*,” in *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, 2017, pp. 328–343.
- [28] M. Imran, N. A. Zafar, M. A. Alnuem, M. S. Aksoy, and A. V. Vasilakos, “*Formal verification and validation of a movement control actor relocation algorithm for safety-critical applications*,” *Wireless Networks*, vol. 22, no. 1, pp. 247–265, 2016.
- [29] J. Trevor, “*A Method and Tool for Automated Analysis of Heavy Vehicle Requirements*,” *Student thesis*, Available from: <http://urn.kb.se/resolve?urn=urn:nbn:se:mdh:diva-29620>.
- [30] H. Shi-Yu and C. Kwang-Ting, “*Formal equivalence checking and design debugging*,” *Boston: Kluz~ er Academic Publishers*, vol. 14, 1998.
- [31] T. Wongpiromsarn and R. M. Murray, “*Formal verification of an autonomous vehicle system*,” in *Conference on Decision and Control*, 2008.
- [32] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and P. Schnoebelen, *Systems and software verification: model-checking techniques and tools*. Springer Science & Business Media, 2013.
- [33] G. Fraser, F. Wotawa, and P. E. Ammann, “*Testing with model checkers: a survey*,” *Software Testing, Verification and Reliability*, vol. 19, no. 3, pp. 215–261, 2009.

- [34] K. G. Larsen, P. Pettersson, and W. Yi, “UPPAAL in a nutshell,” *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 1, no. 1, pp. 134–152, 1997.
- [35] M. S. Nawaz, H. Ali, and M. I. U. Lali, “Concurrent Algorithms in SPIN Model Checker,” in *Frontiers of Information Technology (FIT), 2016 International Conference on*. IEEE, 2016, pp. 193–198.
- [36] M. M. Jaghoori, F. de Boer, D. Longuet, T. Chothia, and M. Sirjani, “Compositional schedulability analysis of real-time actor-based systems,” *Acta Informatica*, vol. 54, no. 4, pp. 343–378, 2017.
- [37] A. Jafari, E. Khamespanah, H. Kristinsson, M. Sirjani, and B. Magnusson, “Statistical model checking of Timed Rebeca models,” *Computer Languages, Systems & Structures*, vol. 45, pp. 53–79, 2016.
- [38] M. Sirjani, F. S. De Boer, and A. Movaghar-Rahimabadi, “Modular Verification of a Component-Based Actor Language.” *J. UCS*, vol. 11, no. 10, pp. 1695–1717, 2005.
- [39] E. Khamespanah, Z. Sabahi Kaviani, R. Khosravi, M. Sirjani, and M.-J. Izadi, “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.
- [40] R. Halder, J. Proença, N. Macedo, and A. Santos, “Formal verification of ROS-based robotic applications using timed-automata,” in *Proceedings of the 5th International FME Workshop on Formal Methods in Software Engineering*. IEEE Press, 2017, pp. 44–50.
- [41] T. Wongpiromsarn and R. M. Murray, “Formal verification of an autonomous vehicle system,” in *Conference on Decision and Control*, 2008.
- [42] M. Weißmann, S. Bedenk, C. Buckl, and A. Knoll, “Model checking industrial robot systems,” in *International SPIN Workshop on Model Checking of Software*. Springer, 2011, pp. 161–176.
- [43] M. Webster, C. Dixon, M. Fisher, M. Salem, J. Saunders, K. L. Koay, K. Dautenhahn, and J. Saez-Pons, “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.
- [44] A. Mohammed, U. Furbach, and F. Stolzenburg, “Multi-robot systems: Modeling, specification, and model checking,” in *Robot Soccer*. InTech, 2010.
- [45] A. Cowley and C. J. Taylor, “Towards language-based verification of robot behaviors,” in *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*. IEEE, 2011, pp. 4776–4782.
- [46] E. Khamespanah, K. Mechitov, M. Sirjani, and G. A. Agha, “Schedulability Analysis of Distributed Real-Time Sensor Network Applications Using Actor-Based Model Checking,” in *Model Checking Software - 23rd International Symposium, SPIN 2016, Co-located with ETAPS 2016, Eindhoven, The Netherlands, April 7-8, 2016, Proceedings*, 2016, pp. 165–181. [Online]. Available: https://doi.org/10.1007/978-3-319-32582-8_11
- [47] B. Magnusson, E. Khamespanah, and R. Khosravi, “Event-based Analysis of Timed Rebeca Models using SQL,” in *ACM 978-1-4503-2189-1/14/10. . .*, 2014. [Online]. Available: <http://dx.doi.org/10.1145/2687357.2687365>
- [48] A. Jafari, E. Khamespanah, M. Sirjani, H. Hermanns, and M. Cimini, “PTRebeca: Modeling and analysis of distributed and asynchronous systems,” *Sci. Comput. Program.*, vol. 128, pp. 22–50, 2016. [Online]. Available: <https://doi.org/10.1016/j.scico.2016.03.004>
- [49] A. Jafari, E. Khamespanah, H. Kristinsson, M. Sirjani, and B. Magnusson, “Statistical model checking of Timed Rebeca models,” *Computer Languages, Systems & Structures*, vol. 45, pp. 53–79, 2016. [Online]. Available: <https://doi.org/10.1016/j.cl.2016.01.004>

- [50] L. Joseph, *Learning ROS for Robotics Programming*, second edition, Ed. PACKT, 2015.
- [51] M. Sirjani and E. Khamespanah, “On Time Actors,” in *Theory and Practice of Formal Methods*. Springer, 2016, pp. 373–392.