Article

Safety Verification of Multiple Industrial Robot Manipulators with Path Conflicts Using Model Checking

Metin Ozkan [1,*], Zekeriyya Demirci [1], Özge Aslan [2] and Ahmet Yazıcı [1]

[1] Department of Computer Engineering, Eskisehir Osmangazi University, 26480 Eskisehir, Turkey
[2] Department of Computer Engineering, Erzincan Binali Yıldırım University, 24002 Erzincan, Turkey
* Correspondence: meozkan@ogu.edu.tr

Abstract: Software development for robotic systems is traditionally performed based on simulations, manual code implementation, and testing. However, this software development approach can cause safety issues in some scenarios, including multiple robots sharing a workspace. When different robots are executing individual planned tasks, they may collide when not adequately coordinated. Safety problems related to coordination between robots may not be encountered during testing, depending on timing, but may occur during the system's operation. In this case, formal verification methods can provide a more reliable means to ensure the safety of robotic systems. This paper uses the formal method of model checking for the safety verification of multiple industrial robot manipulators with path conflicts. We give comparative results of two model-checking tools applied to a system with two robot manipulators. Whole workflows, from requirement specification to testing, are presented.

Keywords: industrial robot; formal verification; model checking; model-driven engineering

check for updates

Citation: Ozkan, M.; Demirci, Z.; Aslan, Ö.; Yazıcı, A. Safety Verification of Multiple Industrial Robot Manipulators with Path Conflicts Using Model Checking. *Machines* **2023**, *11*, 282. https://doi.org/10.3390/machines11020282

Academic Editor: Huosheng Hu

Received: 4 January 2023
Revised: 26 January 2023
Accepted: 5 February 2023
Published: 13 February 2023

1. Introduction

Multiple robot manipulators that carry out independent tasks in a shared workspace need a solution to avoid mutual collisions. The risk of collision is an issue in various robotic fields, including industrial and service applications. However, robotic system software, which requires the evaluation of such issues, is traditionally developed by robotic system experts. Moreover, manual tests are carried out both in the real system and in the simulation environment. Therefore, the verification and validation (V&V) of the system software depend entirely on the experience and attention of the experts. If the V&V of the system software is not conducted adequately, overlooked system errors can lead to downtime for extremely expensive production plants.

Although software plays an ever-increasing role in robotics, current software engineering practices are perceived as insufficient, often leading to error-prone software that is hard to maintain and cannot easily evolve [1]. Moreover, research and industry have tried to propose many model-driven solutions to engineer the software of robotics systems. Casalaro et al. [2] provide a map of software engineering research in MDE for robotic systems since there is no systematic study of state-of-the-art developments in model-driven engineering (MDE) for robotic systems. Robotic systems are advanced cyber-physical systems (CPS) composed of an intricate blend of hardware, software, and environmental components. Software engineering can be very beneficial in the CPS domain; however, it has traditionally been considered an auxiliary concern of robotic system construction. On the other hand, software engineering technologies and methodologies can facilitate the development of robotic systems by adopting a systematic, disciplined, quantifiable approach in each phase of a software application's lifespan, from requirements analysis, system architecture definition, and component design to code implementation, testing, and deployment [3]. Some studies provide roboticists with methods and tools to easily create and validate software for robotic systems. Miyazawa et al. [4] and Ye et al. [5] propose a set of constructs suitable for modeling robotic applications and supporting verification via model checking and theorem proving. Their goal is to support roboticists in writing models and applying modern verification techniques using a language familiar to them. With the use of formal verification methods together with software engineering approaches, robotic systems can be less error-prone and more reliable, and the dependency on human skills can be eliminated.

Although formal verification methods have been discussed in the software domain for many years, they are relatively new for robotic systems. Therefore, the demonstration of workflows and application models for formal verification of robotic systems is essential for dissemination. Some survey papers are investigating formal verification methods for robotic systems. Sinha et al. [6] survey formal methods for the dependability of industrial automation systems. They focus on offline or static approaches, leaving out online or runtime approaches like monitoring, diagnosis, and fault tolerance. Luckcuck et al. [7] systematically survey the state of the art in formal specification and verification for autonomous robotics. The main emphasis is that the commonly used both testing methods and simulations are insufficient to ensure correctness, and there is a need to employ formal methods for autonomous robotic systems. Zheng et al. [8] present a literature survey, an online survey of CPS researchers, and interviews in their paper, Perceptions on the State of the Art in Verification and Validation in Cyber-Physical Systems. Ingrand [9] deals with recent trends in the formal V&V of autonomous robot software. Autonomous robot software needs to be organized along with a particular architecture. Some architectures make easier the V&V of the overall system.

Some studies present architectures to implement V&V for robotic systems. Kanter and Vain [10] present a testing toolkit named TestIt. It provides tools for the automated model-based testing of autonomous multi-robot systems. They use UPPAAL (Ver.4.1.26-2) [11] for modeling and model checking. Wang et al. [12] present a model-based approach, which has the phases of modeling, verification, and automatic code generation. In their framework, UPPAAL is employed for verification, and the code generator produces ROS code. An industrial robot application of grasping a cup is used to demonstrate and evaluate the proposed framework.

The use of various verification methods like model checking, runtime verification, and simulation-based testing, in combination, increases the verification coverage and provide a more reliable system. Webster et al. [13] present an approach for the V&V of a robot assistant in the context of human–robot interaction. They use model checking, simulation-based testing, and user validation in experiments. They entitle their systematic approach corroborative V&V. They employ PRISM (Ver.4.7) [14] for model checking and ROS–Gazebo [15] for simulation-based testing. Villani et al. [16] propose the combined application of two verification techniques: model checking with UPPAAL and CoFI (Conformance and fault injection) model-based testing with ConData [17]. They present the tool and discuss its usage in industrial software development cases. Kejstova et al. [18] describe a novel approach for adapting an existing software model checker to perform precise runtime verification. They assert that the runtime verification after model checking improves coverage. Desai et al. [19] also promote a similar approach. They present an approach for validating the end-to-end correctness of a robotic system by combining model checking and runtime verification. They claim that combining model checking with runtime verification provides a bridge between software verification and the actual execution of the software on a real robotic platform in the physical world.

On the other hand, demonstrating formal verification methods in robotic system applications is of great importance. These applications serve as a model for robotic system developers. These demonstrations will remain valuable until formal verification methods become standard processes in robotic systems and tools are developed to facilitate this process. Halder et al. [20] propose an approach to model and verify ROS systems by using the UPPAAL model checker, focusing on one of the main features of ROS, the communication between nodes. Webster et al. [21] present a case study on formal verification for a high-level planner for the robot named Care-O-bot, an autonomous personal robot assistant. The robot can implement many service tasks in a structured robot house by operating close to its human operators. In this study, the robot models and their environment are formally verified by using model checking. Konur et al. [22] develop and apply automated probabilistic formal verification techniques to robot swarms. They used the probabilistic model checker PRISM and analyzed the probabilistic properties of the swarm. They targeted an existing swarm algorithm for foraging robots. Gjondrekaj et al. [23] present a formal verification approach for collective robotic systems. They used the formal language KLAIM and related analysis tools. They claim that while existing approaches focus on microscopic or macroscopic views of the system, they model aspects of both the robot hardware and behavior, and the related environment. They model a scenario of collective transport by robots. Dixon et al. [24] use formal verification techniques for analyzing the emergent behaviors of robotic swarms. They apply model checking to check whether temporal properties are satisfied for all the possible behaviors of the system. Weismann et al. [25] present a compiler that can transform industrial robot programs into PROMELA models. Then, they implement model checking using SPIN (Ver.6.0) to check collisions and deadlocks in a car-body welding station with nine robots. Quottrup et al. [26] present a study in which they model, analyze, and verify motion-planning problems in a scenario with multiple robotic vehicles. The verification is employed by using the verification software UPPAAL. Gu et al. [27] explore model checking for the automatic generation of mission plans for autonomous vehicles. They propose modeling autonomous vehicles as agents in timed automata with a monitor. Then, they implement a tool called TAMAA (timed-automata-based planner for multiple autonomous agents). The demonstration is conducted in an industrial autonomous wheel-loader use case. Wang et al. [28] present a motion planning method using model checking for reconfigurable robot systems. The verification of the model is implemented by UPPAAL.

Industrial robots are programmable multifunction mechanical devices designed to move material, parts, tools, and specialized devices in order to perform a variety of tasks including assembly, sorting, inspection, and others. Many tasks require the robot to work in coordination with other robots and mechanisms like conveyor bands and CNCs. Studies indicate that many robot accidents occur during non-routine operating conditions, such as programming, maintenance, testing, setup, or adjustment. Rather than directly programming and testing a robotic system, modeling the system software, verifying the model with a formal approach, and subsequently programming in accordance with the model will not only reduce accidents but also increase confidence in the safety of the developed system.

This paper presents a methodology from creating verified models to constructing software designs for robotic systems. The methodology is explained through a case study involving multiple industrial robot manipulators with path conflicts. When there is an overlap in the workspaces of more than one robot, they may collide with each other in their movements in this overlapping space. In order to avoid a safety problem, the robots should not have a path passing through the overlapping space or they should be synchronized. While a solution is proposed for the case study consisting of a safety problem that is encountered with many similar problems in production processes, the methodology for the safety verification of robotic systems is also given. The verified models are constructed by using two prominent model-checking tools: UPPAAL and PRISM. The comparison of the two tools is conducted in terms of the suitability and effectiveness of their features for robotic systems. From the verified model, the design of the software in accordance with the model is explained.

This paper uses the formal method, which is model checking, for the safety verification of multiple industrial robot manipulators with path conflicts. The main contributions are: (i) the presentation of model-checking methods for the conflict resolution of multiple industrial robot manipulators that have a shared workspace; (ii) the creation of suitable models in a sample scenario that can be easily adapted to different similar scenarios; and

(iii) demonstration of the opportunities offered by the model-checking method for the effective programming of robot systems. The paper is organized as follows. In Section 2, we introduce the model-checking technique with two different model constructions. Section 3 describes the case study with modeling. Section 4 presents the verification analysis. The results and discussions are given in Section 5. Then, we provide concluding remarks and directions for future work in Section 6.

2. Model Checking

Model checking is one of the techniques for the formal verification of systems. It verifies the correctness of a system (that is, meeting the defined requirements) by rigorously exploring the behavior of the model, which is an abstraction of the system, expressed in mathematical notation. Thus, all possible behaviors of the systems are described by a finite structure such as a finite-state automaton. Then, the desired properties are specified in a property specification language e.g., temporal logic. Furthermore, all possible paths throughout the automaton correspond to all possible runs of the system. There are many tools for model checking. Two of them, UPPAAL and PRISM, are used in this study. Thus, the first stage of the formal verification of the system using the model-checking technique is to construct the behavioral model of the system. In the following subsections, we introduce the system modeling formalisms utilized by UPPAAL and PRISM, respectively.

2.1. System Modeling Based on Timed Automata

Timed automata were introduced as a formal notation to model the behavior of real-time systems [29,30]. This formalism provides labeled state-transition graphs with timing constraints using real-valued clock variables. The state transition system $\mathcal{S}$ is defined by a tuple $\mathcal{Q}, \mathcal{Q}_0, \Sigma, \rightarrow$, where $\mathcal{Q}$ is a set of states, $\mathcal{Q}_0 \subseteq \mathcal{Q}$ is a set of initial states, Σ is a set of labels (or events), and $\rightarrow \subseteq \mathcal{Q} \times \Sigma \times \mathcal{Q}$ is a set of transitions. The system starts in an initial state, and any transitions between the states are written by $\alpha \xrightarrow{a} \beta$ for $\langle \alpha, a, \beta \rangle \in \rightarrow$ where $\alpha, \beta \in \mathcal{Q}$ and $a \in \Sigma$.

The behaviors of the system may also be expressed as a state transition system $\mathcal{S}^t$ with timing constraints. $\mathcal{S}^t$, also called a timed automaton, is defined by a tuple $\langle \mathcal{L}, \mathcal{L}_0, \mathcal{C}, \mathcal{A}, \mathcal{E}, \mathcal{I} \rangle$. $\mathcal{L}$ is a finite set of locations (representing states of $\mathcal{S}^t$), $\mathcal{L}_0 \subseteq \mathcal{L}$ is the set of start (initial) locations, $\mathcal{C}$ is the finite set of clocks, $\mathcal{A}$ is a set of actions (actions, co-actions, internal τ-action), and $\mathcal{E} \subseteq \mathcal{L} \times \mathcal{A} \times \mathcal{B}(\mathcal{C}) \times 2^{\mathcal{C}} \times \mathcal{L}$ is a set of edges (transitions) between locations with an action, a guard, and a set of clocks. $\mathcal{B}(\mathcal{C})$ is the set of clock constraints involving clocks from $\mathcal{C}$, and $2^{\mathcal{C}}$ is the powerset of $\mathcal{C}$. A clock constraint is simply a term of the form $x \sim c$, where x is a clock, $c \in \mathbb{N}$ is a constant, and $\sim \in \{\langle, \leq, =, \rangle, \geq\}$ is a comparison operator. $\mathcal{I} : \mathcal{L} \rightarrow \mathcal{B}(\mathcal{C})$, which is called an invariant, is a mapping that labels each location in $\mathcal{L}$ with some clock constraint in $\mathcal{B}(\mathcal{C})$.

The states of $\mathcal{S}^t$ are represented by pairs (l, v), where l is a location and v is a clock interpretation for $\mathcal{C}$, such that v satisfies the invariant $\mathcal{I}(l)$. The set of all states of $\mathcal{S}^t$ is denoted by $\mathcal{Q}$. A state (l, v) is the initial state if l is an initial location of $\mathcal{S}^t$ and $v(x) = 0$ for all clocks x.

The transition may occur with elapse of time for a state (l, v) and a real-valued time increment $\delta > 0$, $(l, v) \xrightarrow{\delta} (l, v + \delta)$, if for all $0 < \delta' < \delta$, $v + \delta\prime$ satisfies the invariant $\mathcal{I}(l)$. It may also occur with a location switch for a state (l, v) and a switch l, ε, g, r, l' representing an edge from location l to location l', where $\varepsilon \in \mathcal{E}$ is a label, g is a guard and a clock constraint over $\mathcal{C}$, and $r \subseteq \mathcal{C}$ is a subset of clocks to be reset. The location switch implements $(l, v) \xrightarrow{\varepsilon} (l', v[r \mapsto 0])$. $v[r \mapsto 0]$ means the clock assignment that maps all clocks in r to 0 and agrees with v for other clocks in $\mathcal{C} \setminus r$.

Timed automata are often composed of a network of timed automata over a common set of clocks and actions, consisting of n timed automata $\mathcal{S}^t_i = \langle \mathcal{L}_i, \mathcal{L}_{0i}, \mathcal{C}, \mathcal{A}, \mathcal{E}_i, \rangle \mathcal{I}_i$, $i \leq i \leq n$.

The expressions in timed automata can be addressed in guard, invariant, channel, and *update*. A *guard* is a particular expression evaluating a Boolean function on edges. An *invariant* is an expression that indicates the time that can be spent on a node. A *channel* is considered for synchronizing the progress of two or more automata. An update is an expression set that assigns values to clocks and variables.

2.2. System Modeling Based on Continuous Time Markov Chains (CTMC)

A continuous-time Markov chain (CTMC) is a continuous stochastic process. The process changes state according to an exponential random variable and then moves to a different state as specified by the probabilities of a stochastic matrix. A continuous-time Markov chain $\mathcal{M}$ is a tuple $\mathcal{Q}$, $\mathcal{R}$, $\mathcal{L}$, where $\mathcal{Q}$ is a finite set of states; $\mathcal{R} : \mathcal{Q} \times \mathcal{Q} \to \mathbb{R}_{\geq 0}$ is the transition rate matrix; and $\mathcal{L} : \mathcal{Q} \to 2^{AP}$ is a labeling with atomic propositions [31]. $\mathcal{L}$ assigns to each state $s \in \mathcal{Q}$ the set $\mathcal{L}(s)$ of atomic propositions $a \in AP$ that are valid in s.

A transition between s and s' may occur when $\mathcal{R}(s,s') > 0$, and $1 - e^{-\mathcal{R}(s,s')t}$ is the probability that the transition $s \to s'$ can be triggered within t time units. Thus, the delay of transition $s \to s'$ may occur with the exponential distribution with the rate $\mathcal{R}(s,s')$.

2.3. Relation between Verification Models and Software Design

Software designs generally consist of many modules that interact with each other. If an object-oriented approach is used for the software design, each module becomes an instance of a class. The model of each module may be represented by timed automata or CTMS, and the behavior of each module in the overall software is represented by the network of timed automata, as seen in Figure 1.

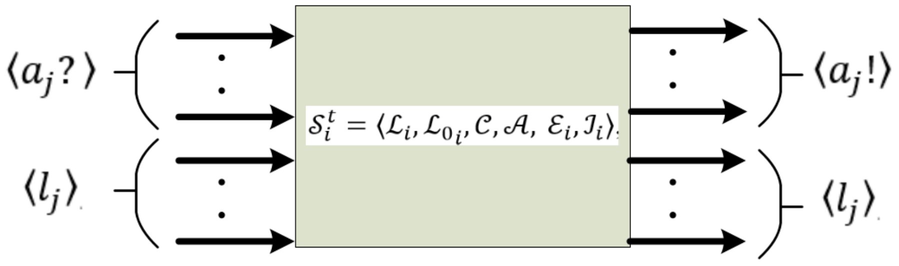

$\langle a_j ? \rangle$
$\langle l_j \rangle$
$\mathcal{S}_i^t = \langle \mathcal{L}_i, \mathcal{L}_{0_i}, \mathcal{C}, \mathcal{A}, \mathcal{E}_i, \mathcal{I}_i \rangle$
$\langle a_j ! \rangle$
$\langle l_j \rangle$

Figure 1. A representation of timed automata as the members of a network.

In Figure 1, $\langle a_j ! \rangle \in \mathcal{A}$ are the sent synch messages from the *ith* module to other modules for triggering *jth* action. Similarly, $\langle a_j ? \rangle \in \mathcal{A}$ are the received synch from the a module to the *ith* module for triggering *jth* action. Some state transitions of the *ith* module may depend on the states of other modules as the parameters of the guard condition. Therefore, $\langle l_j \rangle \in \mathcal{L}_i$ are the locations (states) of a module needed by another module.

3. Multiple Industrial Robot Systems Operating in a Shared Workspace: A Case Study

Some robotic production scenarios need multiple robot manipulators to operate close to each other. In this case, the robots in a shared workspace may cause safety issues that will result in a mutual collision. Various approaches can be chosen to avoid a collision. One of them is to employ online or offline motion planning to prevent collisions. Another approach is to synchronize the robots in possible collision zones and make them wait. In all cases, robot software should be developed to ensure safe operation.

We studied a robotic production system, including two industrial robots sharing a workspace. The scenario discussed includes common problems with many robotic production systems, and the solution can be generalized to be used in all these systems. This system can be seen in Figure 2.

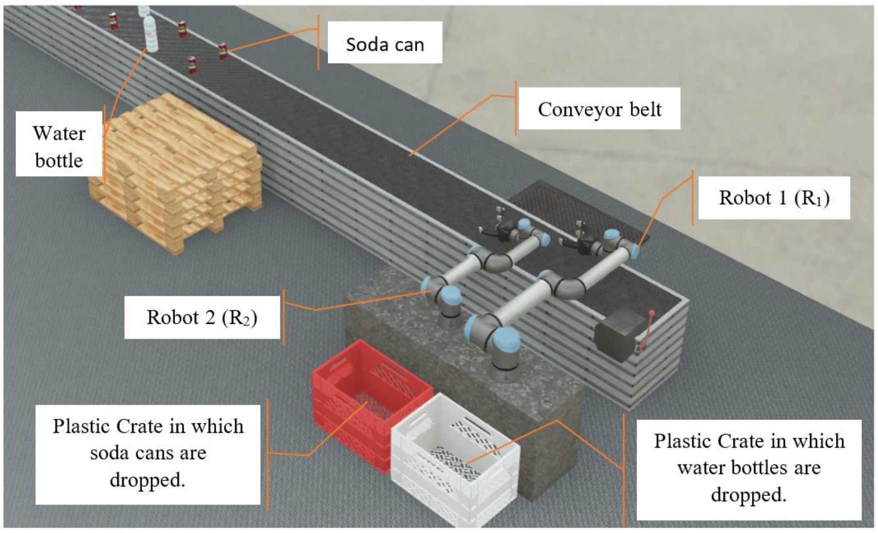

Soda can
Conveyor belt
Water bottle
Robot 1 (R₁)
Robot 2 (R₂)
Plastic Crate in which soda cans are dropped.
Plastic Crate in which water bottles are dropped.

Figure 2. Robotic workspace.

The aim is to throw the red soda cans on the conveyor belt into the red crate and the white water bottles into the white crate. Two robot arms in the system, called UR10 (R_1) and UR5 (R_2), implement the task. There is a gripper mechanism at the end of the robot arms. As shown in Figure 3, there are two distance sensors labeled D_1 and D_2 in the gripper. The water bottle is taller than the soda can. Due to the size of the soda can, it can only be detected with the D_1 sensor. On the other hand, a water bottle can be detected by both D_1 and D_2 sensors. In this case, depending on the distances measured by D_1 and D_2, it can be determined whether the object entering the gripper is soda or water.

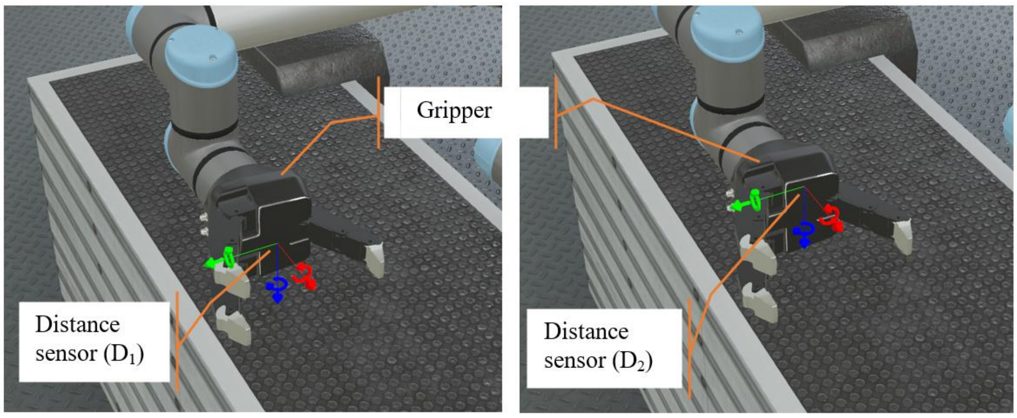


Figure 3. The tips of the robot arms.

The tips of the robot arms move point-to-point through the pre-determined positions. The pre-determined positions are shown in Figure 4. Each robot waits for the object on the conveyor band at the position BAND_i. When the object is subsequently perceived and grasped, then the robot moves at the position HOME_i. If the grasped object is a soda can, the robot moves to RED_i and drops the soda can into the red crate. If the grasped object is a water bottle, the robot moves to WHITE_i and drops the water bottle into the white crate. Then, the robot moves back to the position BAND_i over HOME_i.

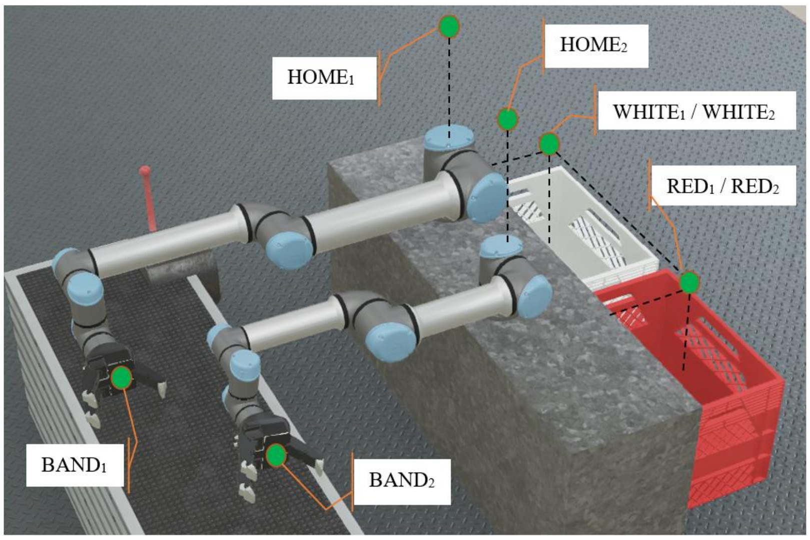


Figure 4. The target and waypoints of robots.

3.1. Description of the Problem

Let robots R_i, $i \in [1, n]$ perform their task in a shared workspace W. The number of robots is n. Each task requires the tip, $\mathcal{X}_i \in \mathbb{R}^6$, of R_i to follow a precomputed path P_i passing through some waypoints on WP_i. At WP_i, R_i has configurations in joints space $\mathcal{Q}_i \in \mathbb{R}^m$ of R_i and in task space $\mathcal{X}_i \in \mathbb{R}^6$, of R_i. The degree of freedom (DOF) for the robot is m. Assume that R_i is moving from any $wp_m \in WP_i$ to $wp_n \in WP_i$, where $m \neq n$ at the same period with R_j, $j \in [1, n] \setminus i$, and $i \neq j$, or is moving from any $wp_k \in WP_j$ to $wp_l \in WP_l$, where $k \neq l$, then the path between wp_m and wp_n for R_i, and the path between wp_k and wp_l for R_j, include the path conflict. If these paths are implemented at the same period by R_i and R_j, then it would probably cause a collision when the robots are not synchronized properly. This is a safety problem that needs to be solved. It is necessary to develop a solution for such problems and to verify that the solution provides safety.

For the case study, there are two robots where each follows the path P_i including $WP_i = (BAND_i, HOME_i, RED_i, WHITE_i)$. The scenario requires each robot (i ϵ [1,2]) to follow the steps below:

- R_i is initially at $BAND_i$;
- When a soda can or water bottle enters the gripper, the type of the product is detected;
- The product is grasped;
- R_i goes to $HOME_i$;
- If the grasped product is a soda can, R_i goes to RED_i;
- If the grasped product is a water bottle, R_i goes to $WHITE_i$;
- R_i drops the water bottle/soda can into the crate;
- R_i goes to $HOME_i$;
- R_i goes to $BAND_i$;
- The process continues.

The safety requirements are defined as

RQ1. R_2 should not move to $WHITE_2$ while R_1 is at $WHITE_1$ or moving from/to $WHITE_1$.
RQ2. R_2 should move to neither $WHITE_2$ nor RED_2 while R_1 is at RED_1 or moving from/to RED_1.
RQ3. R_1 should not move to RED_1 while R_2 is at RED_2 or moving from/to RED_2.
RQ4. R_1 should move to neither $WHITE_1$ nor RED_1 while R_2 is at $WHITE_2$ or moving from/to $WHITE_2$.

RQ5. R_i should not move directly between $BAND_i$ and $RED_i/WHITE_i$ for $\forall i \in [1,2]$.

Any model for task programming of the robots needs to be verified to confirm whether the safety requirements RQ1–RQ5 are satisfied. In this paper, model checking as a formal verification approach is used. Model checking is employed at the design stage of the program, in which a model of the program is constructed. Since the model is typically non-deterministic, each run of the model can be different from the last. A model checker, which implements the model-checking approach, exhaustively analyzes all possible executions of the model in order to establish some properties, usually derived from the requirements [21].

The workflow for verifying the system based on model checking is implemented as shown in Figure 5. For the model-based development of a system, models are constructed at the design stage by using the description and requirements of the system. The description and requirements of the system are used to create the models. It is necessary to verify that the models meet the requirements. Model checking is employed for verification. Then, the verified models are implemented and tested.

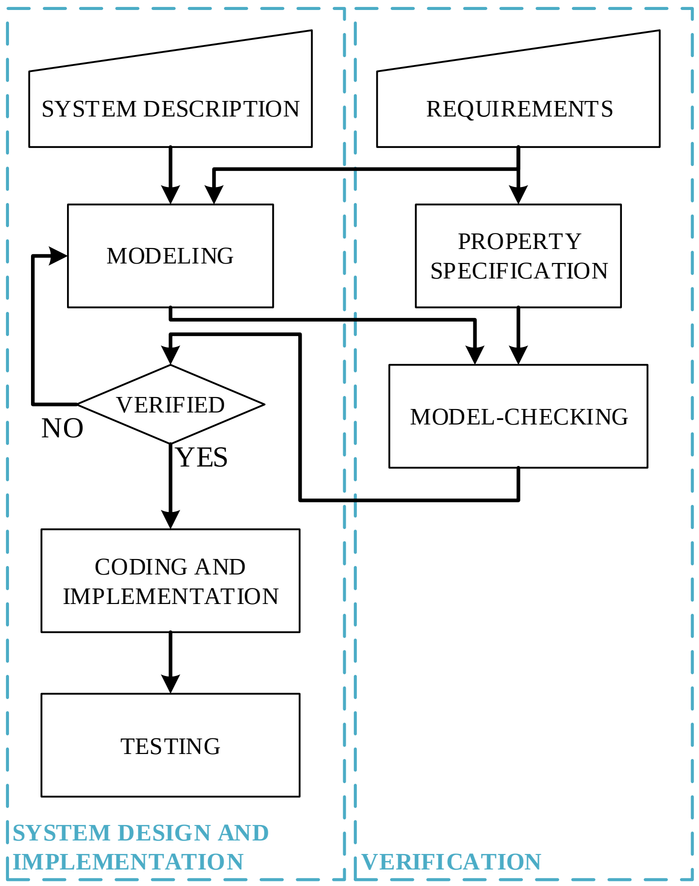


Figure 5. System development and verification stages.

3.2. Timed Automata Modeling

Various V&V tools are specialized in different system behaviors, such as the UPPAAL model checker. It is a commonly used tool for the modeling, simulation, and verification of real-time systems, and was developed by Uppsala University and Aalborg University [11]. It uses timed automata extended by real-value clocks for system modeling. It also uses TCTL (timed computation tree logic) to express properties to be verified against given specifications.

The robotic system is modeled as a network of timed automata. Each automaton may be constructed parametrically as a template in which the automata may represent the behavioral model of various subsystems. For example, there are two robot manipulators, and a single template models both of them. Four different templates–robot controller, gripper, product detector, and conveyor band–are modeled to represent the case study scenario. Each template is modeled using timed automata. In addition, the templates include a parameter that simultaneously enables the execution of multiple instances on one template.

Modeling begins with the identification of functional modules in the system. Then, the behavioral states of each module are determined. For each module, the actions and states related to other modules are determined. Last, a timed automata model of each module is created.

3.2.1. Robot Controller

The case study has four waypoints–$BAND$, $HOME$, RED, and $WHITE$–all of which are also considered to be states of the robot. It takes time for the robots to move between the waypoints. Therefore, another state needs to be defined, namely $MOVING$, which represents the robot moving from/to any primary waypoint, as seen in Figure 6. It is assumed that all $MOVING$ states take three units of time.

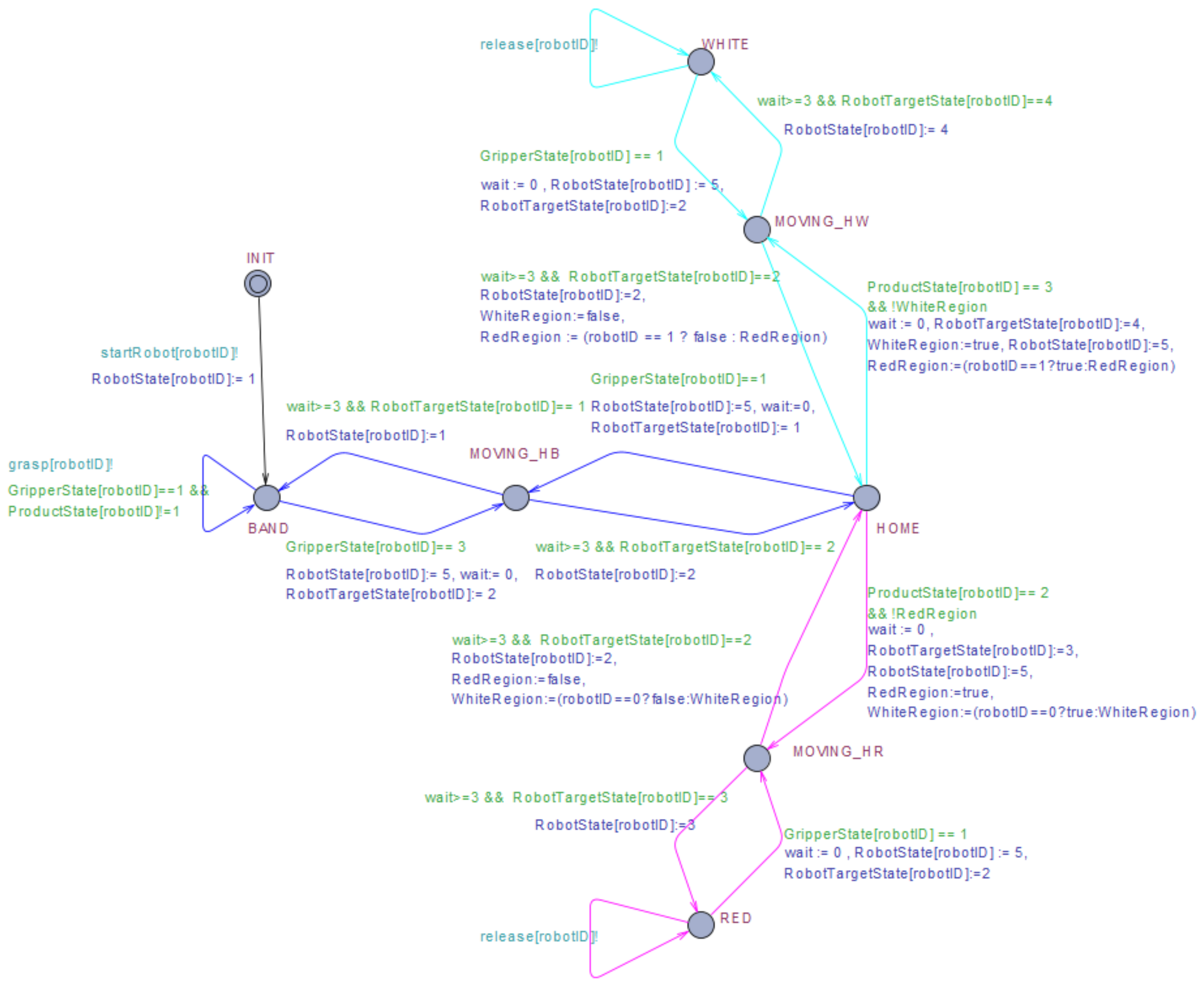

release[robotID]!
WHITE
wait>=3 && RobotTargetState[robotID]==4
RobotState[robotID]:= 4
GripperState[robotID] == 1
wait := 0 , RobotState[robotID] := 5, RobotTargetState[robotID]:=2
MOVING_HW
INIT
wait>=3 && RobotTargetState[robotID]==2
RobotState[robotID]:=2,
WhiteRegion:=false,
RedRegion := (robotID == 1 ? false : RedRegion)
ProductState[robotID] == 3
&& !WhiteRegion
wait := 0, RobotTargetState[robotID]:=4,
WhiteRegion:=true, RobotState[robotID]:=5,
RedRegion:=(robotID==1?true:RedRegion)
startRobot[robotID]!
RobotState[robotID]:= 1
GripperState[robotID]==1
wait>=3 && RobotTargetState[robotID]== 1
RobotState[robotID]:=5, wait:=0,
RobotTargetState[robotID]:= 1
RobotState[robotID]:=1
MOVING_HB
grasp[robotID]!
GripperState[robotID]==1 &&
ProductState[robotID]!=1
BAND
HOME
GripperState[robotID]== 3
RobotState[robotID]:= 5, wait:= 0,
RobotTargetState[robotID]:= 2
wait>=3 && RobotTargetState[robotID]== 2
RobotState[robotID]:=2
ProductState[robotID]== 2
&& !RedRegion
wait := 0 ,
RobotTargetState[robotID]:=3,
RobotState[robotID]:=5,
RedRegion:=true,
WhiteRegion:=(robotID==0?true:WhiteRegion)
wait>=3 && RobotTargetState[robotID]==2
RobotState[robotID]:=2,
RedRegion:=false,
WhiteRegion:=(robotID==0?false:WhiteRegion)
MOVING_HR
wait>=3 && RobotTargetState[robotID]== 3
RobotState[robotID]:=3
GripperState[robotID] == 1
wait := 0 , RobotState[robotID] := 5,
RobotTargetState[robotID]:=2
RED
release[robotID]!

Figure 6. UPPAAL timed automata model of the robot controller.

The state transition system $\mathcal{S}_{RobotController}$ for the robot controller can be defined as follows:

$\mathcal{L} = \{INIT, BAND, MOVING\_HB, HOME, MOVING\_HW, MOVING\_HR, WHITE, RED\}$

$\mathcal{L}_0 = \{INIT\}$

$\mathcal{C} = \{wait\}$

$\mathcal{A} = \{grasp,\ release, startRobot\}$
$\mathcal{I} = \{\varnothing\}$

Inspired by the principle of mutual exclusion to allow robots to move without collision, when the robot comes $HOME$, it can only move to a waypoint if that waypoint is free. Otherwise, the robot must wait for the waypoint to become free. If the waypoint is already allocated as free, the robot moves to the target waypoint and is allocated as busy. Until the robot goes back to $HOME$, no other robot can move to the allocated waypoint.

The robot controller template starts at $INIT$ and passes to $BAND$ without restriction. At the same time, it triggers the gripper to $OPEN$ and the product detector to $NONE$. When the object is perceived and grasped, it passes $HOME$. If all safety requirements are satisfied, it continues forward to the RED or WHITE location, depending on the object type. When it reaches RED or $WHITE$, it triggers the gripper to drop the object into the crate. Then, it moves back to $HOME$ and $BAND$ respectively.

Figure 7 presents the block diagram of the robot controller module, which indicates the actions and states related to other modules.

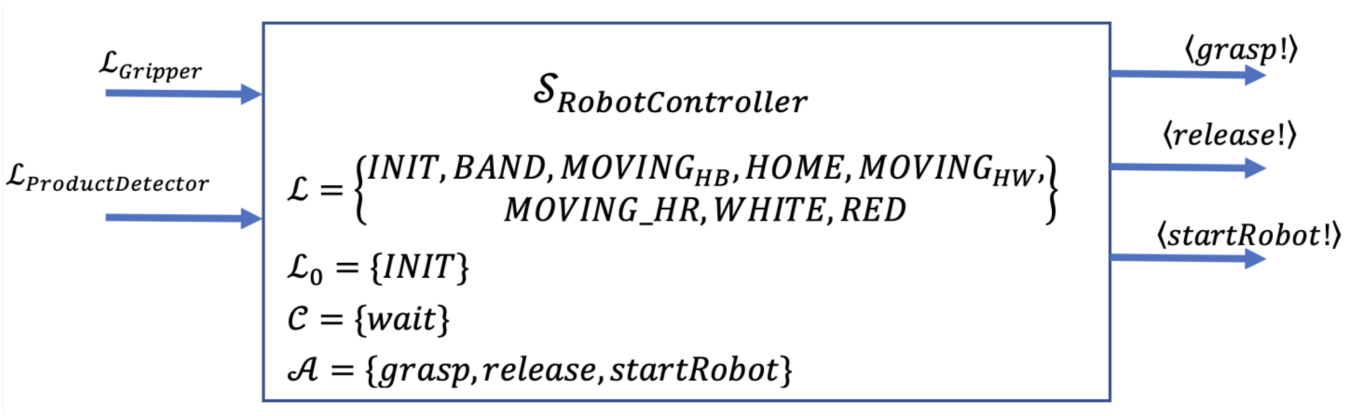

$\mathcal{L}_{Gripper}$
$\mathcal{L}_{ProductDetector}$
$\mathcal{S}_{RobotController}$
$\mathcal{L} = \begin{Bmatrix} INIT, BAND, MOVING_{HB}, HOME, MOVING_{HW}, \\ MOVING\_HR, WHITE, RED \end{Bmatrix}$
$\mathcal{L}_0 = \{INIT\}$
$\mathcal{C} = \{wait\}$
$\mathcal{A} = \{grasp, release, startRobot\}$
$\langle grasp! \rangle$
$\langle release! \rangle$
$\langle startRobot! \rangle$

Figure 7. Block diagram of the robot controller module.

3.2.2. Gripper

The gripper template has two primary states: OPEN and CLOSE, as shown in Figure 8. When the robot starts running, it triggers the gripper to pass to $OPEN$. Whenever it takes the action to grasp, the gripper passes to the state $CLOSE$. Conversely, whenever it takes the action to release, it triggers the gripper to turn back to $OPEN$. When the gripper passes between CLOSE and OPEN, it takes time for the motion to be completed. Therefore, the state MOVING is needed to represent the gripper motion. It is assumed that the $MOVING$ state takes three units of time.

The state transition system $\mathcal{S}_{Gripper}$ for the gripper can be defined as follows:
$\mathcal{L} = \{INIT, CLOSE, MOVING, OPEN\}$
$\mathcal{L}_0 = \{INIT\}$
$\mathcal{C} = \{wait\}$
$\mathcal{A} = \{grasp,\ release, startRobot, clearDetector\}$
$\mathcal{I} = \{\varnothing\}$

Figure 9 presents the block diagram of the gripper module, which indicates the actions and states related to other modules.

3.2.3. Product Detector

The gripper has two distance sensors, namely D1 and D2, which read two different values to detect soda and water. The detection results constitute the state of the product detector, such as SODA, WATER, and NONE. Figure 10 shows the timed automata of the product detector.

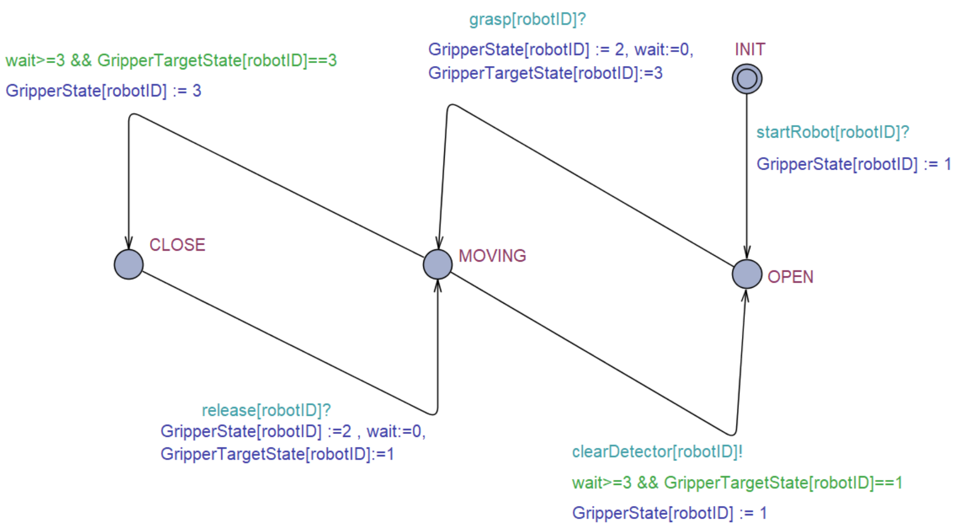

grasp[robotID]?
GripperState[robotID] := 2, wait:=0,
GripperTargetState[robotID]:=3
INIT
wait>=3 && GripperTargetState[robotID]==3
GripperState[robotID] := 3
startRobot[robotID]?
GripperState[robotID] := 1
CLOSE
MOVING
OPEN
release[robotID]?
GripperState[robotID] :=2 , wait:=0,
GripperTargetState[robotID]:=1
clearDetector[robotID]!
wait>=3 && GripperTargetState[robotID]==1
GripperState[robotID] := 1

Figure 8. UPPAAL Timed automata model of gripper.

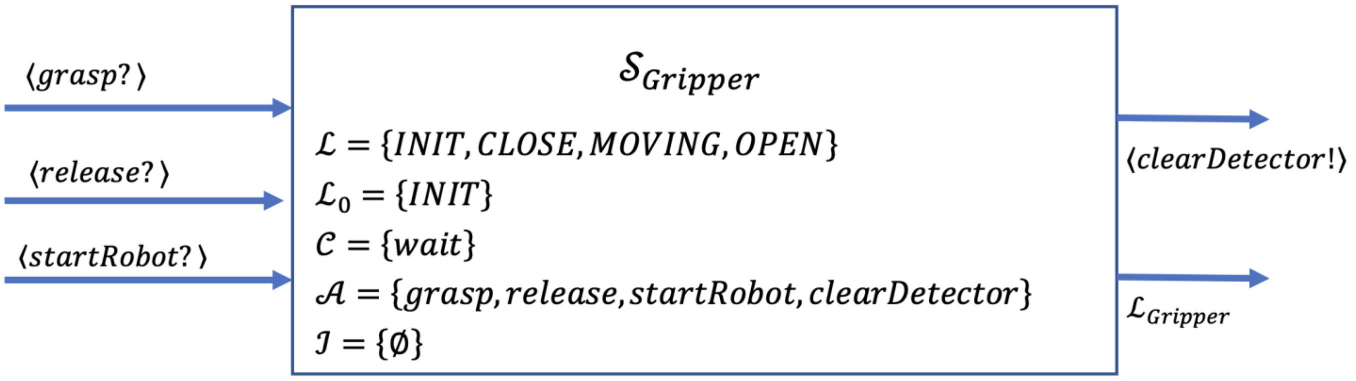

⟨grasp?⟩
⟨release?⟩
⟨startRobot?⟩
$\mathcal{S}_{Gripper}$
$\mathcal{L} = \{INIT, CLOSE, MOVING, OPEN\}$
$\mathcal{L}_0 = \{INIT\}$
$\mathcal{C} = \{wait\}$
$\mathcal{A} = \{grasp, release, startRobot, clearDetector\}$
$\mathcal{I} = \{\emptyset\}$
⟨clearDetector!⟩
$\mathcal{L}_{Gripper}$

Figure 9. Block diagram of the gripper module.

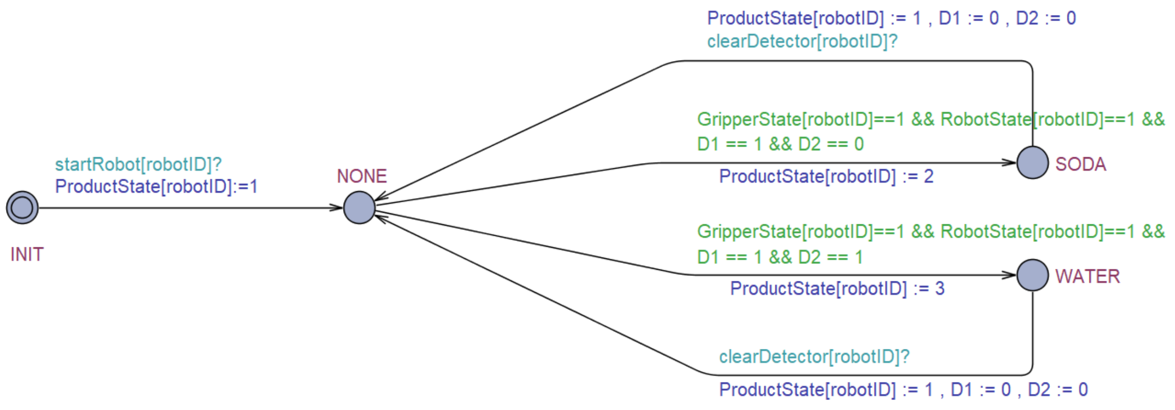

ProductState[robotID] := 1 , D1 := 0 , D2 := 0
clearDetector[robotID]?
GripperState[robotID]==1 && RobotState[robotID]==1 &&
D1 == 1 && D2 == 0
ProductState[robotID] := 2
SODA
startRobot[robotID]?
ProductState[robotID]:=1
NONE
INIT
GripperState[robotID]==1 && RobotState[robotID]==1 &&
D1 == 1 && D2 == 1
ProductState[robotID] := 3
WATER
clearDetector[robotID]?
ProductState[robotID] := 1 , D1 := 0 , D2 := 0

Figure 10. UPPAAL timed automata model of product detector.

The state transition system $\mathcal{S}_{ProductDetector}$ for the gripper can be defined as follows:
$\mathcal{L} = \{INIT, NONE, SODA, WATER\}$
$\mathcal{L}_0 = \{INIT\}$
$\mathcal{C} = \{\varnothing\}$
$\mathcal{A} = \{startRobot, clearDetector\}$
$\mathcal{I} = \{\varnothing\}$

Figure 11 presents the block diagram of the product detector module, which indicates the actions and states related to other modules.

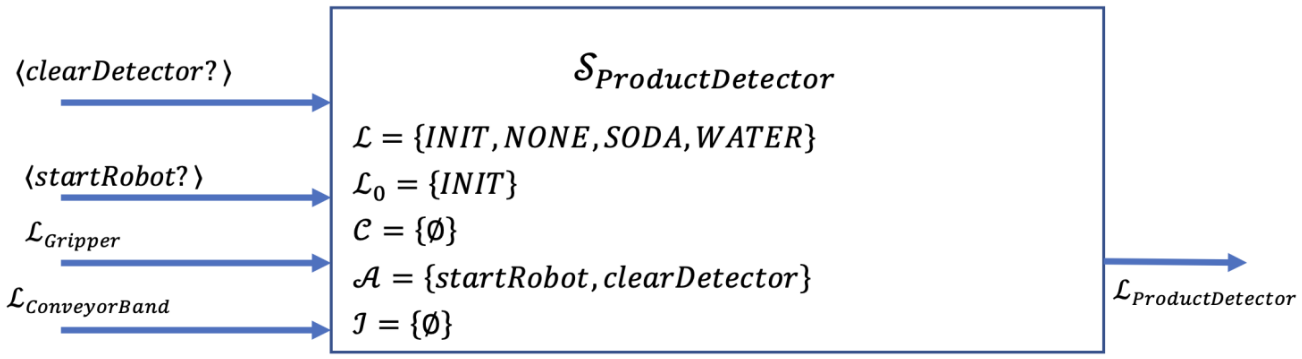

⟨clearDetector? ⟩
⟨startRobot? ⟩
$\mathcal{L}_{Gripper}$
$\mathcal{L}_{ConveyorBand}$
$\mathcal{S}_{ProductDetector}$
$\mathcal{L} = \{INIT, NONE, SODA, WATER\}$
$\mathcal{L}_0 = \{INIT\}$
$\mathcal{C} = \{\emptyset\}$
$\mathcal{A} = \{startRobot, clearDetector\}$
$\mathcal{I} = \{\emptyset\}$
$\mathcal{L}_{ProductDetector}$

Figure 11. Block diagram of the product detector module.

3.2.4. Conveyor Band

The conveyor band is simply a driver system that simulates the presence of soda and water nondeterministically appearing on the band. There are three states, *init*, *soda*, and *water*, as shown in Figure 12. Each state represents the various configuration of the distance sensor readings $(D1, D2) = \{(0,0), (1,0), (1,1)\}$.

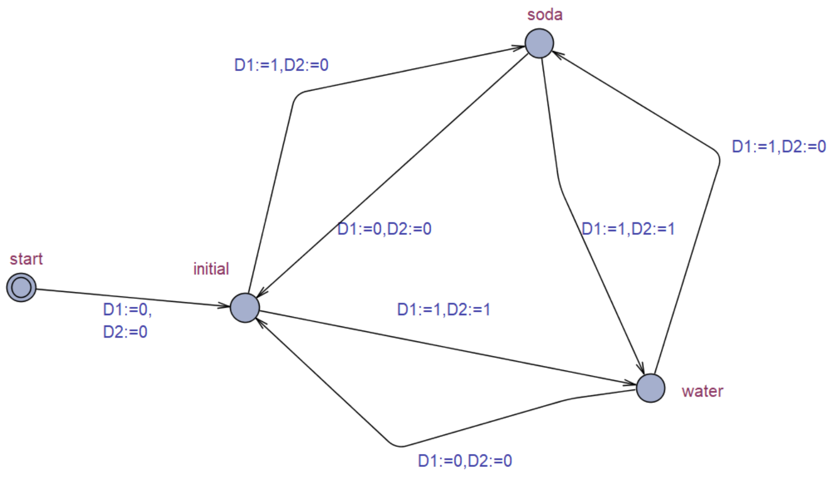


Figure 12. UPPAAL timed automata model of the conveyor band.

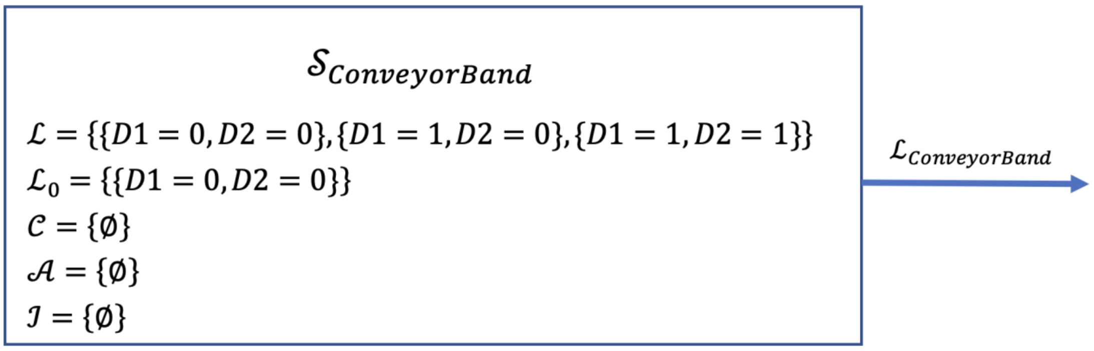
The state transition system $\mathcal{S}_{ConveyorBand}$ for the gripper can be defined as follows:
$\mathcal{L} = \{\{D1 = 0, D2 = 0\}, \{D1 = 1, D2 = 0\}, \{D1 = 1, D2 = 1\}\}$
$\mathcal{L}_0 = \{\{D1 = 0, D2 = 0\}\}$
$\mathcal{C} = \{\varnothing\}$
$\mathcal{A} = \{\varnothing\}$
$\mathcal{I} = \{\varnothing\}$

Figure 13 presents the block diagram of the conveyor band module, which indicates the actions and states related to other modules. It is a driver module in order to use the verification of the overall model. It is not a part of the system software.

The block diagram including all of the modules and their relationships is shown in Figure 14.

3.3. Continuous Time Markov Chains (CTMC) Modeling

PRISM is one of the model-checking tools used to verify a probabilistic system. It can build and analyze many types of probabilistic models. There are three modules for describing the system models: the robot controller, gripper, and product detector. The robot controller module is modeled for each robot separately because of the differences between the behavior of the robots, as shown in Algorithms 1 and 2. On the other hand, there are one gripper and one product detector module. Due to the similar functionalities of each robot, multiple instances of these modules can be used.



$\mathcal{S}_{ConveyorBand}$

$\mathcal{L} = \{\{D1 = 0, D2 = 0\}, \{D1 = 1, D2 = 0\}, \{D1 = 1, D2 = 1\}\}$

$\mathcal{L}_0 = \{\{D1 = 0, D2 = 0\}\}$

$\mathcal{C} = \{\emptyset\}$

$\mathcal{A} = \{\emptyset\}$

$\mathcal{I} = \{\emptyset\}$

Figure 13. Block diagram of the product detector module.

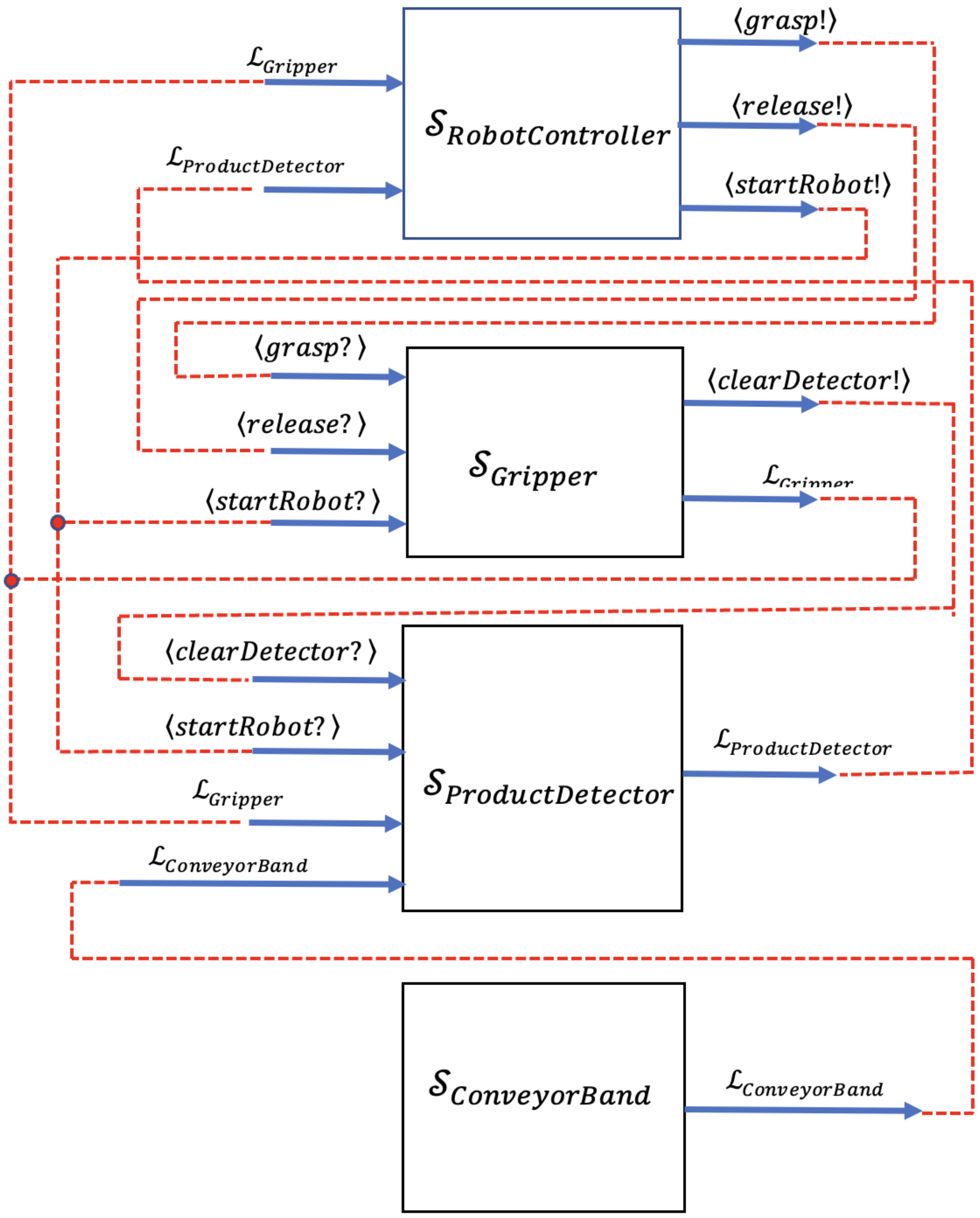

$\langle grasp! \rangle$
$\mathcal{L}_{Gripper}$
$\mathcal{S}_{RobotController}$
$\langle release! \rangle$
$\mathcal{L}_{ProductDetector}$
$\langle startRobot! \rangle$
$\langle grasp? \rangle$
$\langle clearDetector! \rangle$
$\langle release? \rangle$
$\mathcal{S}_{Gripper}$
$\mathcal{L}_{Gripper}$
$\langle startRobot? \rangle$
$\langle clearDetector? \rangle$
$\langle startRobot? \rangle$
$\mathcal{L}_{ProductDetector}$
$\mathcal{S}_{ProductDetector}$
$\mathcal{L}_{Gripper}$
$\mathcal{L}_{ConveyorBand}$
$\mathcal{S}_{ConveyorBand}$
$\mathcal{L}_{ConveyorBand}$

Figure 14. Block diagram of all modules.

Algorithm 1. PRISM CTMC model of robot controller1.

```
ctmc
globalredRegion: bool init false;
global white region : bool init false;
// R1 denotes robot which is located left side conveyor band, as well as R2 denotes the right one.
module RobotController1
        // robot states =>> [ 0: INIT, 1: BAND, 2: HOME, 3: RED, 4: WHITE, 5: MOVING_BH,
6:MOVİNG_HR, 7:MOVİNG_HW ]
        robotState1 : [0..7] init 0;
        robotTargetState1 : [0..7] init 0;
        robotPreState1 : [0..7] init 0;
// BAND <-> HOME
        // Init -> Band
        [startRobot1] robotState1 = 0 -> ( robotState1′ = 1 ) & ( robotPreState1′ =0 );
        // Band -> Band
        [grasp1] robotState1 = 1 & gripperState1 = 1 & productState1 != 1 -> ( robotState1′ = 1 );
        // Band -> Moving_BH
        [] robotState1 = 1 & gripperState1 = 3 -> ( robotState1′ = 5 ) & ( robotTargetState1′ = 2 )
& ( robotPreState1′ = 1 );
            // Moving_BH -> Home
        [] robotState1 = 5 & robotTargetState1 = 2 -> ( robotState1′ = 2 ) ;
        // Home -> Moving_BH
        [] robotState1 = 2 & gripperState1 = 1 -> ( robotState1′ = 5 ) & ( robotTargetState1′ = 1 )
& ( robotPreState1′ = 2 );
            // Moving_BH -> Band
        [] robotState1 = 5 & robotTargetState1 = 1 -> (robotState1′ = 1 );
// HOME <-> RED
        // Home -> Moving_HR
        [] robotState1 = 2 & productState1 = 2 & redRegion = false & whiteRegion = false ->
( robotState1′ = 6 ) & ( robotTargetState1′ = 3 ) & ( redRegion′ = true ) & ( whiteRegion′ = true )
& ( robotPreState1′ = 2 );
        // Moving_HR -> Red
        [] robotState1 = 6 & robotTargetState1 = 3 -> ( robotState1′ = 3 );
        // Red -> Red
        [release1] robotState1 = 3 -> ( robotState1′ = 3 );
        // Red -> Moving_HR
        [] robotState1 = 3 & gripperState1 = 1 -> ( robotState1′ = 6 ) & ( robotTargetState1′ = 2 )
& ( robotPreState1′ = 3 );
            // Moving_HR -> Home
        [] robotState1 = 6 & robotTargetState1 = 2 -> ( robotState1′ = 2 ) & ( redRegion′ = false )
& ( whiteRegion′ = false ) ;
// HOME <-> WHITE
        // Home -> Moving_HW
        [] robotState1 = 2 & productState1 = 3 & whiteRegion = false ->
        ( robotState1′ = 7 ) & ( robotTargetState1′ = 4 ) & ( whiteRegion′ = true ) & (
robotPreState1′ = 2 );
        // Moving_HW -> White
        [] robotState1 = 7 & robotTargetState1 = 4 -> ( robotState1′ = 4 ) ;
        // White -> White
        [release1] robotState1 = 4 -> ( robotState1′ = 4 );
        // White -> Moving_HW
        [] robotState1 = 4 & gripperState1 = 1 -> ( robotState1′ = 7 ) & ( robotTargetState1′ = 2 )
& ( robotPreState1′ = 4 );
        // Moving_HW -> Home
        [] robotState1 = 7 & robotTargetState1 = 2 -> ( robotState1′ = 2 ) & ( whiteRegion′ =
false );
endmodule
```

Algorithm 2. PRISM CTMC model of robot controller2.

```
module RobotController2
        // robot states =>> [ 0: INIT, 1: BAND, 2: HOME, 3: RED, 4: WHITE, 5: MOVING_BH,
6:MOVİNG_HR, 7:MOVİNG_HW ]
        robotState2 : [0..7] init 0;
        robotTargetState2 : [0..7] init 0;
        robotPreState2 : [0..7] init 0;
// BAND <-> HOME
        // Init -> Band
        [startRobot2] robotState2 = 0 -> ( robotState2' = 1 ) & ( robotPreState2' = 0 );
        // Band -> Band
        [grasp2] robotState2 = 1 & gripperState2 = 1 & productState2 != 1 -> ( robotState2' = 1 );
        // Band -> Moving_BH
        [] robotState2 = 1 & gripperState2 = 3 -> ( robotState2' = 5 ) & ( robotTargetState2' = 2 ) & (
robotPreState2' = 1 );
        // Moving_BH -> Home
        [] robotState2 = 5 & robotTargetState2 = 2 -> ( robotState2' = 2 ) ;
        // Home -> Moving_BH
        [] robotState2 = 2 & gripperState2 = 1 -> ( robotState2' = 5 ) & ( robotTargetState2' = 1 ) & (
robotPreState2' = 2 );
            // Moving_BH -> Band
        [] robotState2 = 5 & robotTargetState2 = 1 -> (robotState2' = 1 );
// HOME <-> RED
        // Home -> Moving_HW
        [] robotState2 = 2 & productState2 = 3 & whiteRegion = false & redRegion = false ->
( robotState2' = 7 ) & ( robotTargetState2' = 4 ) & ( whiteRegion' = true ) & ( redRegion' = true ) & (
robotPreState2' = 2 );
        // Moving_HR -> White
        [] robotState2 = 7 & robotTargetState2 = 4 -> ( robotState2' = 4 ) ;
        // White -> White
        [release2] robotState2 = 4 -> ( robotState2' = 4 );
        // White -> Moving_HR
        [] robotState2 = 4 & gripperState2 = 1 -> ( robotState2' = 7 ) & ( robotTargetState2' = 2 ) & (
robotPreState2' = 4 );
        // Moving_HR -> Home
        [] robotState2 = 7 & robotTargetState2 = 2 -> ( robotState2' = 2 ) & ( whiteRegion' = false )
& ( redRegion' = false ) ;
// HOME <-> WHITE
        // Home -> Moving_HR
        [] robotState2 = 2 & productState2 = 2 & redRegion = false ->
( robotState2' = 6 ) & ( robotTargetState2' = 3 ) & ( redRegion' = true ) & ( robotPreState2' = 2 );
        // Moving_HW -> Red
        [] robotState2 = 6 & robotTargetState2 = 3 -> ( robotState2' = 3 ) ;
        // Red -> Red
        [release2] robotState2 = 3 -> ( robotState2' = 3 );
        // Red -> Moving_HW
        [] robotState2 = 3 & gripperState2 = 1 -> ( robotState2' = 6 ) & ( robotTargetState2' = 2 ) & (
robotPreState2' = 3 );
        // Moving_HW -> Home
        [] robotState2 = 6 & robotTargetState2 = 2 -> ( robotState2' = 2 ) & ( redRegion' = false ) ;
endmodule
```

In summary, the robot controller module has three main transitions between the state couples given as (HOME, BAND), (HOME, RED), and (HOME, WHITE). All possible transitions between these states and substates (MOVING_BH, MOVING_HR, MOVING_HW) are defined by actions, guards, and state updates.

The model of the gripper module is given in Algorithm 3. The transitions between the states OPEN and CLOSE occur as a result of the actions named startRobot, grasp, release, and clearDetector.

The model of the product detector module is given in Algorithm 4. There are two different objects on the conveyor belt, and the probability of their arrival is assumed to be the same. Therefore, both have a 50% probability of arrival, as shown in Algorithm 4. After the choice is made randomly, the gripper module is triggered by the robot controller and the object is grasped.

Algorithm 3. PRISM CTMC model of gripper.

```
module GripperTool1
        // gripper states =>> [ 0: START, 1: OPEN, 2: MOVING, 3: CLOSE ]
        gripperState1 : [0..3] init 0;
        gripperTargetState1 : [0..3] init 0;
        // Start -> Open
        [startRobot1] gripperState1 = 0 -> ( gripperState1' = 1 );
        // Open -> Moving
        [grasp1] gripperState1 = 1 -> ( gripperState1' = 2 ) & ( gripperTargetState1' = 3 );
        // Moving -> Close
        [] gripperState1 = 2 & gripperTargetState1 = 3 -> ( gripperState1' = 3 );
        // Closed -> Moving
        [release1] gripperState1 = 3 -> ( gripperState1' = 2 ) & ( gripperTargetState1' = 1 );
            // Moving -> Open
        [clearDetector1] gripperState1 = 2 & gripperTargetState1 = 1 -> ( gripperState1' = 1 ) ;
endmodule
```

Algorithm 4. PRISM CTMC model of product detector.

```
module ProductDetector1
        // sensor states =>> [ 0: Init, 1: None, 2: Soda, 3: Water ]
        productState1 : [0..3] init 0;
        // Init -> None
        [startRobot1] productState1 = 0 -> ( productState1' = 1 );
        // None -> Soda (id : 2) | Water (id : 3)
        [] productState1 = 1 & gripperState1 = 1 & robotState1 = 1 ->0.5: ( productState1' = 2 ) +
0.5: ( productState1' = 3) ;
            // water | soda -> None
        [clearDetector1] productState1 = 2 | productState1 = 3-> ( productState1' = 1 );
endmodule
```

Unlike the robot controller module, the gripper and the product detector modules do not need to be modelled separately for each robot. Therefore, the only requirement is to create a new module instance by replacing the local variables with new ones, as shown in Algorithm 5.

Algorithm 5. PRISM straightforward way to create a new module from an existing one.

```
module Gripper-
Tool2=GripperTool1[gripperState1=gripperState2,gripperTargetState1=gripperTargetState2,
startRobot1=startRobot2,grasp1=grasp2,release1=release2,clearDetector1=clearDetector2]
endmodule

module
ProductDetector2=ProductDetector1[productState1=productState2,startRobot1=startRobot2,

gripperState1=gripperState2,robotState1=robotState2,clearDetector1=clearDetector2]
endmodule
```

4. Verification Analysis

Each robotic system is designed and implemented based on some requirements. The requirements may be categorized as functional and non-functional. For example, placing soda cans from the moving conveyor band in the red crate and water bottles in the white crate by means of the robot arms is a functional requirement. However, the requirements labeled RQ1–RQ5 that are given in Section 3.1 are non-functional safety requirements.

During the construction of the model, the model of the robotic system needs to be checked to establish whether it meets the requirements. Model checkers perform this task by exploring the model state space exhaustively in order to determine whether or not it satisfies the required property. The requirements are formally specified as properties. There are many formal specification languages used by various model checkers.

4.1. Property Specification and Model Checking for UPPAAL

The UPPAAL requirement specification language provides semantics for specifications in CTL. Let φ be a given state formula for the specification of requirements. A state formula is an expression that can be evaluated for a state without looking at the behavior of the model. It supports five types of properties. These are as follows:

- $E <> \varphi$ is named possibly and evaluates to true for a timed transition system if and only if there exists a path starting at the initial state, such that φ is eventually satisfied along that path;
- $A[]\ \varphi$ is named invariantly and evaluates to true if and only if every reachable state satisfies φ;
- $E[]\ \varphi$ is named potentially always and evaluates to true for a timed transition system if and only if there is a sequence for which φ holds in all states;
- $A <> \varphi$ is named eventually and evaluates to true if and only if all possible transition sequences eventually reach a state satisfying φ;
- $\varphi \rightarrow \psi$ whenever φ holds eventually ψ will hold as well.

The requirements RQ1–RQ5 can be specified as follows:

RQ1. A[] not (R2.MOVING_HW && (R1.MOVING_HW || R1.WHITE))
RQ2. A[] not ((R2.MOVING_HW || R2.MOVING_HR) && (R1.MOVING_HR || R1.RED))
RQ3. A[] not ((R1.MOVING_HR) && (R2.MOVING_HR || R2.RED))
RQ4. A[] not ((R1.MOVING_HW || R1.MOVING_HR) && (R2.MOVING_HW || R2.WHITE))
RQ5. No need to query for this requirement since there is no direct path between the states $BAND_i$ and $RED_i/WHITE_i$ on the model of the robot controller.

The properties given in RQ1–RQ5 need to be satisfied. Once the properties and model are given to the UPPAAL tool, it verifies whether or not the properties are satisfied for the given model. The models shown in Figures 5–8 and properties RQ1–RQ4 are verified by UPPAAL.

4.2. Property Specification and Model for PRISM

PRISM is capable of understanding properties written in several well-known probabilistic temporal logics. The system requirements are expressed as properties written in PCTL (probabilistic computational tree logic), which is one of these logics. The PRISM property specification language allows the P operator to be used to express the probability of an event occurring. The P operator is used with a bound value and a path property. *A* bound value could be any of $< p$, $> p$, $\leq p$, or $\geq p$, where p has a range of 0 to 1.

- $P > 0.98$ [$pathprop$] is true in a state S of a model if the probability that $pathprop$ is satisfied by the paths from state S is greater than 0.98.

Through the quantitative properties approach, the P operator has also a usage as follows:

- $P =?$ [$pathprop$] $\cdot$ is used when computing the actual probability of the given path property's occurrences.

The path property is a formula to be verified for a single path in a model. It has different types of temporal operators that can be used inside the P operator as follows:

- X (next): The property X *prop* is true for a path if prob is true in its second state;
- U (until): The property *prop1* U *prop2* is true for a path if *prop2* is true in some state of the path and *prop1* is true in all preceding states;
- F (eventually or future): The property F *prop* is true for a path if *prop* eventually becomes true at some point along the path;
- G (always or globally): The property G *prop* is true of a path if *prop* remains true all states along the path;
- W (weak until): The property *prop1* W *prop2* is true for a path if *prop1* remains true until *prop2* becomes true, but does not require that *prop2* ever does become true;
- R (release): The property *prop1* R *prop2* is true for a path if *prop2* is true until *prop1* becomes true, or *prop2* is true forever.

The requirements RQ1-RQ5 can be specified as follows:

RQ1. P = ? [F(robotState2 = 7) & ((robotState1 = 4) | (robotState1 = 7))]
RQ2. P = ? [F((robotState2 = 6) | (robotState2 = 7)) & ((robotState1 = 3) | (robotState1 = 6))]
RQ3. P = ? [F(robotState1 = 6) & ((robotState2 = 3) | (robotState2 = 6))]
RQ4. P = ? [F((robotState1 = 6) | (robotState1 = 7)) & ((robotState2 = 4) | (robotState2 = 7))]
RQ5. No need to query for this requirement since there is no direct path between the states $BAND_i$ and $RED_i/WHITE_i$ on the model of the robot controller.

The model given in Algorithms 1–5 and properties RQ1–RQ4 are verified by PRISM. The probability of occurrence of the given properties RQ1–RQ4 is zero, as expected.

5. Results and Evaluation

A case study for multiple industrial robot manipulators with path conflicts is discussed. The software of the robotic system in our case study was developed by a model-based approach. First, a state-based verification model of the system was created in order to be verified by two model checker tools, UPPAAL and PRISM.

UPPAAL and PRISM are model-checking tools capable of verifying systems against given specifications. Both tools are used for modeling and verification. However, each tool has strengths and weaknesses compared to others. UPPAAL is more suitable for real-time systems modeled as networks of timed automata, while PRISM is suitable for developing probabilistic models. Furthermore, UPPAAL allows the modeling of the system via a GUI, but the modeling process in PRISM is code-based [32]. Therefore, modeling in PRISM requires more attention than in UPPAAL. Both consist of a simulator that enables the examination of possible dynamic executions of a system. Moreover, their verifier can detect syntax errors, but properties must be verified through the verifier. UPPAAL uses a subset of CTL as its property specification language, but PRISM utilizes logic such as LTL and PCTL. In addition, there is a difference in model constructions that can be inspected from Figure 5 and Algorithms 1 and 2. The model of the robot controller can be described as a template in UPPAAL. Furthermore, the template is instantiated by two processes for robots R1 and R2. However, in PRISM, the robots R1 and R2 are described in two separate modules.

After the model of the system was verified by meeting the desired requirements, then the control software of the system could be developed to reflect the verified model. The object-oriented design approach was employed to implement the control software. The UML class diagram of the system is given in Figure 15. run() methods of each class implement the related state transition models, which are verified for meeting the requirements. The classes are coded in C++.

The simulation environment was constructed by using WEBOTS. Then, the robot program was run in the simulation environment. The products soda and water moved randomly on the conveyor band. During the execution, it was seen that the requirements were met. Figure 16 shows some snapshots from the simulations. The complete simulation video can be watched at the link https://youtu.be/V_Tpsp5NUA0 (accessed on 3 January 2023). At $t = 0$ s , the grippers mounted on the robot arms are over the conveyor band, and the products are coming through the grippers. Before $t = 6$ s , both robot arms hold a water bottle and move towards the white crate at almost the same time. Furthermore, at *t* = 6 s, robot R2 is waiting for R1 while robot R1 is dropping the bottle and returning to the position HOME. Then, at *t* = 8.30 s, the robot R2 also moves toward the white crate and drops the bottle. Similarly, at *t* = 12.30 s, R1 grabs a soda can while R2 grabs a water bottle. In this case, R1 waits at HOME until R2 returns HOME after dropping the water bottle into the white crate.

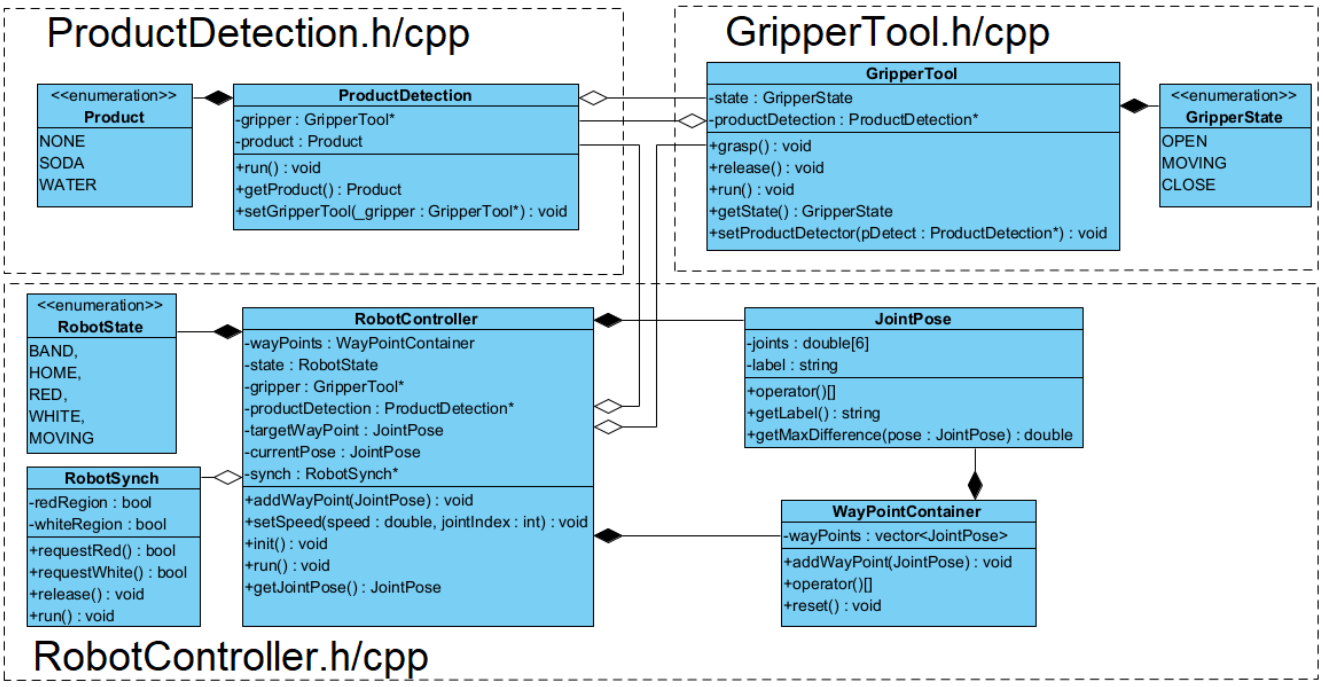


Figure 15. The UML design of the robot software based on the verified model.

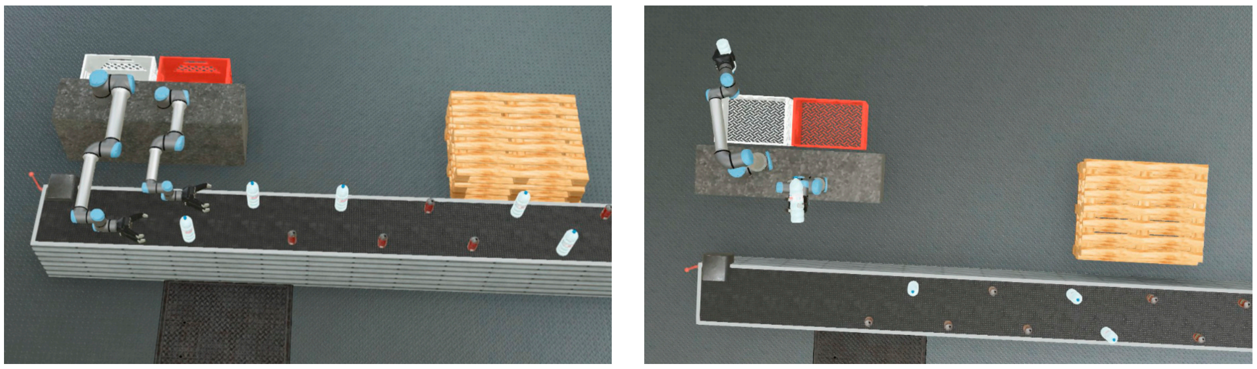


Figure 16. *Cont.*

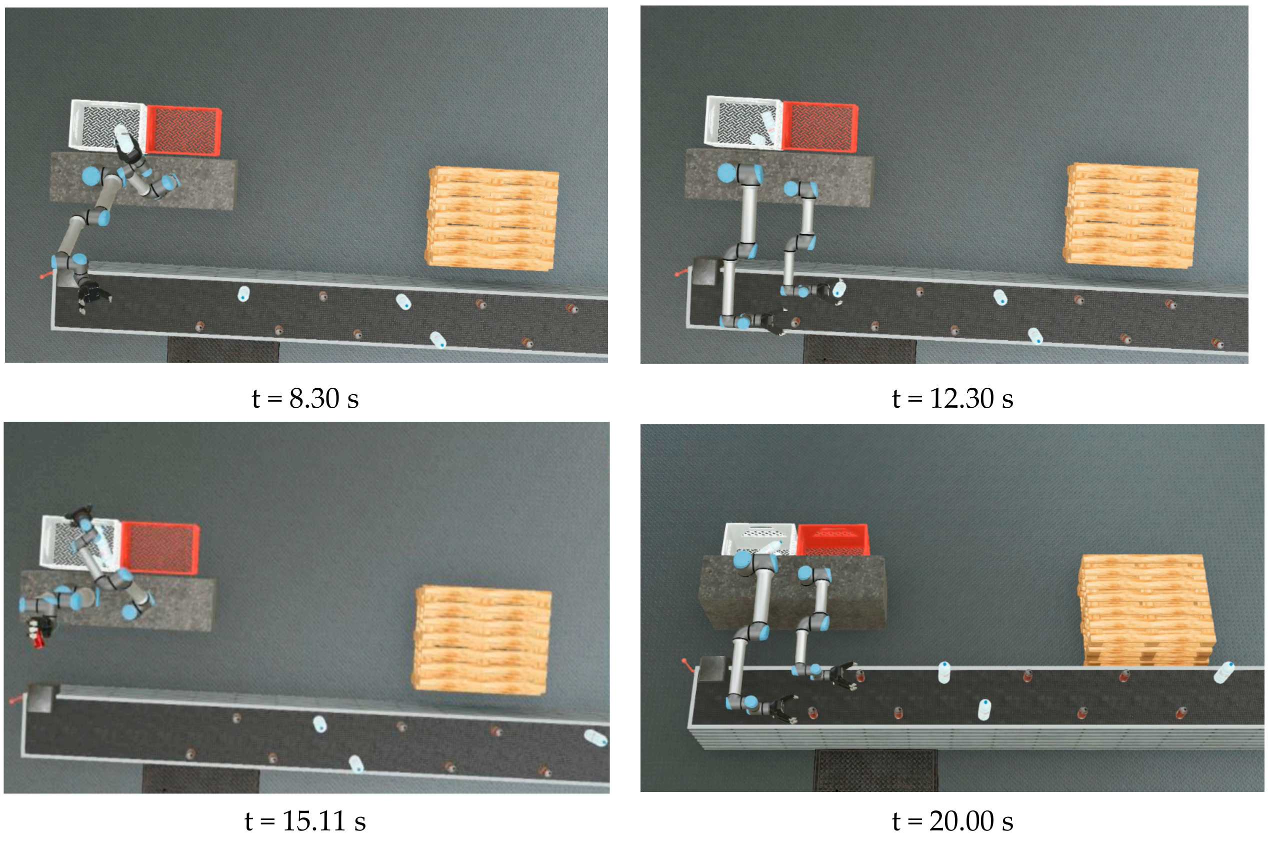


Figure 16. Snapshots from the simulations.

6. Conclusions

This paper addresses the verification problem of robotic systems, including multiple industrial robot manipulators with path conflicts. Robotic system developers generally prefer the use of test processes based on expertise. However, safety issues are likely to arise with this approach. In this study, we demonstrate the use of formal verification methods in robotic systems. The formal verification method is implemented by two model-checker tools: UPPAAL and PRISM. The paper presents a model description and property specifications for two different model checker tools on the same system, revealing the modeling similarities and differences between the two leading tools. In addition, we show how system software designed using an object-oriented approach utilizes the verified model. Finally, the simulation presents the execution of the software that enables the robotic system to perform safely without any collisions between robot manipulators.

Author Contributions: Conceptualization, M.O. and Z.D.; methodology, M.O.; software, Z.D.; validation, M.O., Z.D. and A.Y.; formal analysis, Z.D. and Ö.A.; investigation, M.O. and Z.D.; resources, Z.D.; data curation, Z.D.; writing—original draft preparation, M.O., Z.D., and A.Y.; writing—review and editing, M.O., Z.D., and A.Y.; visualization, M.O. and Z.D.; supervision, M.O.; project administration, M.O.; funding acquisition, A.Y. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by theVALU3S project that has received funding from the ECSEL Joint Undertaking (JU) under grant agreement No 876852. The JU receives support from the European Union’s Horizon 2020 research and innovation programme and Austria, Czech Republic, Germany, Ireland, Italy, Portugal, Spain, Sweden, Turkey (TUBITAK, under contract no:119N356). The views expressed in this work are the sole responsibility of the authors and do not necessarily reflect the views or position of the European Commission. The authors, the VALU3S Consortium, and the ECSEL JU are not responsible for the use which might be made of the information contained in here. This work is supported by the Scientific and Technical Research Council of Turkey (TUBITAK), Contract No 120N800, project title: “Verification and Validation of Automated Systems’ Safety and Security”.

Data Availability Statement: Not applicable.

Acknowledgments: This work was supported by theVALU3S project that has received funding from the ECSEL Joint Undertaking (JU) under grant agreement No 876852. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Austria, Czech Republic, Germany, Ireland, Italy, Portugal, Spain, Sweden, Turkey (TUBITAK, under contract no:119N356).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Garcia, S.; Strüber, D.; Brugali, D.; Berger, T.; Pelliccione, P. Robotics Software Engineering: A Perspective from the Service Robotics Domain. In Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual, 8–13 November 2020.
2. Casalaro, G.L.; Cattivera, G.; Malavolta, I.C.; Wortmann, A.; Pelliccione, P. Model-driven engineering for mobile robotic systems: A systematic mapping study. *Softw. Syst. Model.* **2022**, *21*, 19–49. [CrossRef]
3. Brugali, D.; and Prassler, E. Software engineering for robotics [From the Guest Editors]. *IEEE Robot. Autom. Mag.* **2009**, *16*, 9–15. [CrossRef]
4. Miyazawa, A.; Ribeiro, P.; Li, W.; Cavalcanti, A.; Timmis, J.; Woodcock, J. RoboChart: Modelling and verification of the functional behaviour of robotic applications. *Softw. Syst. Model.* **2019**, *18*, 3097–3149. [CrossRef]
5. Ye, K.; Cavalcanti, A.; Foster, S.; Miyazawa, A.; Woodcook, J. Probabilistic modelling and verification using RoboChart and PRISM. *Softw. Syst. Model.* **2021**, *21*, 667–716. [CrossRef]
6. Sinha, R.; Patil, S.; Gomes, L.; Vyatkin, V. A Survey of Static Formal Methods for Building Dependable Industrial Automation Systems. *EEE Trans. Ind. Inform.* **2019**, *15*, 3772–3783. [CrossRef]
7. Luckcuck, M.; Farrell, M.; Dennis, L.; Dixon, C.; Fisher, M. Formal Specification and Verification of Autonomous Robotic Systems: A Survey. *ACM Comput. Surv.* **2019**, *52*, 1–41. [CrossRef]
8. Zheng, X.; Julien, C.; Kim, M.; Khurshid, S. Perceptions on the State of the Art in Verification and Validation in Cyber-Physical Systems. *IEEE Syst. J.* **2017**, *11*, 2614–2627. [CrossRef]
9. Ingrand, F. Recent Trends in Formal Validation and Verification of Autonomous Robots Software. In Proceedings of the 2019 Third IEEE International Conference on Robotic Computing (IRC), Naples, Italy, 25–27 February 2019.
10. Kanter, G.; Vain, J. Model-based testing of autonomous robots using TestIt. *J. Reliab. Intell. Environ.* **2020**, *6*, 15–30. [CrossRef]
11. Larsen, K.G.; Pettersson, P.; Yi, W. UPPAAL in a nutshell. *Int. J. Softw. Tools Technol. Transf.* **1997**, *1*, 134–152. [CrossRef]
12. Wang, R.; Luo, P.; Guan, Y.; Wei, H.; Li, X.; Zhang, J.; Song, X. Timed automata-based motion planning for a self-assembly robot system. In Proceedings of the 2014 IEEE International Conference on Robotics and Automation (ICRA), Hongkong, China, 31 May–7 June 2014.
13. Webster, M.; Western, D.; Araiza-Illan, D.; Dixon, C.; Eder, K.; Fisher, M.; Pipe, A.G. Corroborative approach to verification and validation of human-robot teams. *Int. J. Robot. Res.* **2019**, *39*, 73–79. [CrossRef]
14. Kwiatkowska, M.; Norman, G.; and Parker, D. PRISM 4.0: Verification of Probabilistic Real-time Systems. In Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11), Snowbird, UT, USA, 14–20 July 2011; Springer: Berlin/Heidelberg, Germany, 2011; Volume 6806, pp. 585–591.
15. GAZEBO. 2021. Available online: https://gazebosim.org/ (accessed on 3 January 2023).
16. Villani, E.; Pontes, R.P.; Coracini, G.K.; Ambrósio, A.M. Integrating model checking and model based testing for industrial software development. *Comput. Ind.* **2019**, *104*, 88–102. [CrossRef]
17. Martins, E.; Sabião, S.B..; Ambrosio, A.M. ConData: A tool for automating specification-based test case generation for communication systems. *Softw. Qual. J.* **1999**, *8*, 303–319. [CrossRef]
18. Kejstova, K.; Rockai, P.; Barnat, J. From Model Checking to Runtime Verification and Back. RV2017: Runtime Verification. In *Lecture Notes in Computer Science*; Springer: Berlin/Heidelberg, Germany, 2017; Volume 10548, pp. 225–240.
19. Desai, A.; Dreossi, T.; Seshia, S.A. Combining Model Checking and Runtime Verification for Safe Robotics. Runtime Verification. RV 2017. In *Lecture Notes in Computer Science*; Springer: Berlin/Heidelberg, Germany, 2017; Volume 10548, pp. 172–189.
20. Halder, R.; Proença, J.; Macedo, N.; Santos, A. Formal Verification of ROS-Based Robotic Applications Using Timed-Automata. In Proceedings of the 2017 IEEE/ACM 5th International FME Workshop on Formal Methods in Software Engineering, Buenos Aires, Argentina, 27 May 2017; pp. 44–50.
21. Webster, M.; Dixon, C.; Fisher, M.; Salem, M.; Saunders, J.; Koay, K.L.; Dautenhahn, K.; Saez-Pons, J. Toward Reliable Autonomous Robotic Assistants Through Formal Verification: A Case Study. *IEEE Trans. Hum.-Mach. Syst.* **2016**, *46*, 186–196. [CrossRef]
22. Konur, S.; Dixon, C.; Fisher, M. Analysing robot swarm behaviour via probabilistic model checking. *Robot. Auton. Syst.* **2012**, *60*, 199–213. [CrossRef]
23. Gjondrekaj, E.; Loreti, M.; Pugliese, R.; Tiezzi, F.; Pinciroli, C. Towards a Formal Verification Methodology for Collective Robotic Systems. In *Formal Methods and Software Engineering. ICFEM 2012. Lecture Notes in Computer Science*; Aoki, T., Taguchi, K., Eds.; Springer: Berlin/Heidelberg, Germany, 2012; Volume 7635.
24. Dixon, C.; Winfield, A.F.T.; Fisher, M.; Zeng, C. Towards temporal verification of swarm robotic systems. *Robot. Auton. Syst.* **2012**, *60*, 1429–1441. [CrossRef]

25. Weißmann, M.; Bedenk, S.; Buckl, C.; Knoll, A. Model Checking Industrial Robot Systems. In Proceedings of the International SPIN Workshop on Model Checking of Software, Snowbird, UT, USA, 14–15 July 2011.
26. Quottrup, M.M.; Bak, T.; Izadi-Zamanabadi, R. Multi-robot planning: A timed automata approach. In Proceedings of the IEEE International Conference on Robotics and Automation, ICRA '04, New Orleans, LA, USA, 26 April–1 May 2004.
27. Gu, R.; Enoiu, E.; Secelenau, C. TAMAA: UPPAAL-based mission planning for autonomous agents. In Proceedings of the 35th Annual ACM Symposium on Applied Computing, Virtual, 30 March–3 April 2020; Association for Computing Machinery: Brno, Czech Republic, 2020; pp. 1624–1633.
28. Wang, R.; Guan, Y.; Song, H.; Li, X.; Li, X.; Shi, Z.; Song, X. A Formal Model-Based Design Method for Robotic Systems. *IEEE Syst. J.* **2019**, *13*, 1096–1107. [CrossRef]
29. Alur, R. Timed Automata. In *Computer Aided Verification. CAV 1999. Lecture Notes in Computer Science*; Halbwachs, N., Peled, D., Eds.; Springer: Berlin/Heidelberg, Germany, 1999; Volume 1633. [CrossRef]
30. Alur, R.; Dill, D.R. A theory of timed automata. *Theor. Comput. Sci.* **1994**, *126*, 183–235. [CrossRef]
31. Baier, C.; Haverkort, B.; Hermanns, H.; Katoen, J. Model Checking Algorithms for Continuous-Time Markov Chains. *IEEE Trans. Softw. Eng.* **2003**, *29*, 6. [CrossRef]
32. Naeem, A.; Azam, F.; Amjad, A.; Anwar, M.W. Comparison of Model Checking Tools Using Timed Automata—PRISM and UPPAAL. In Proceedings of the 2018 IEEE International Conference on Computer and Communication Engineering Technology, Beijing, China, 19–20 September 2018.