Norwegian University
of Life Sciences

Master's Thesis 2024 30 ECTS
Faculty of Science and Technology

Formal verification of synchronization properties of a multi-robot welding system

Henrik Nordlie
Applied Robotics

Preface

First and foremost, I would like to thank my supervisors, David A. Anisi [1] and Yvonne Murray [2], for being easily available and eager to help me throughout this process. I would also like to thank Pedro Ribeiro from the University of York for his quick responses to my many questions about RoboChart.
A special thanks goes out to my fellow students, who have inspired me with their enthusiasm and nerdiness which helped me become the engineer I am today. I also want to thank my parents for letting me find my own way and encouraging me to pursue my interests.
Finally, I would like to thank my lovely girlfriend, Kathrine, for her constant support while we both fought our way through a semester full of frustration and hard work.

- Henrik Nordlie

[1]Faculty of Science & Technology, Norwegian University of Life Sciences
[2]Pioneer Robotics/Faculty of Engineering and Science, University of Agder

Abstract

Welding tasks are among the most prevalent use cases of robots. In this thesis, the synchronization properties of a welding system, the IntelliWelder M06, are formally verified. The welding system consists of a robotic arm (UR10e) and a welding table (Carpano FIVE MOT) external axis. This system allows the user to define a welding operation directly on a CAD model in simulation and consequently plans trajectories for the robotic arm and external axis so that the welding operation can be performed autonomously. The relevant sections of this system are modeled using RoboChart, a UML-based modeling language tailored to represent robotic applications accurately. This work is performed to verify properties critical to the synchronized operation of the system through model checking. Desired properties were defined as assertions through *tock*-CSP, a process algebra that includes the *tock* event representing the passage of discrete time. The assertions required the following properties:

- Each time a movement request is received for the robotic arm or the external axis, a movement operation is called for the corresponding component.
- The state machines corresponding to the system state tracker, the external axis, and the robotic arm do not terminate.

The CSP refinement checker, FDR4, was used to verify the validity of these properties. The results showed that the system requirements were fulfilled for sets of inputs based on the assumption of correctness outside of the model. Also, they were not fulfilled if the components outside the model were not assumed to be ideal, which shows that the modeled part of the system is input-dependant.

Sammendrag

Sveiseoppgaver er ett av de vanligste bruksområdene for roboter. I denne masteroppgaven blir synkroniseringsegenskapene til et sveisesystem, IntelliWelder M06, formelt verifisert. Dette systemet består av en robotarm (UR10e) og et sveisebord (Carpano FIVE MOT) som fungerer som en ekstern akse. Løsningen laget av Pioneer Robotics tillater operatøren å definere en sveis direkte på en CAD-modell i simulering, og planlegger baner for robotarmen og den eksterne aksen slik at sveiseoppgaven kan utføres autonomt. De relevante delene av systemet blir modellert ved hjelp av RoboChart, et modelleringsspråk basert på UML som er skreddersydd for å nøyaktig representere anvendelser innen robotikk. Dette arbeidet ble gjort for å verifisere egenskaper som er kritiske for sykroniseringen av kompentene i dette systemet ved bruk av modell-sjekking. Nødvendige egenskaper ble definert som påstander i *tock*-CSP, en prosess-algebra som inkluderer *tock* hendelsen for å representere at diskretisert tid passerer. Påstandene krevde at de følgende egenskapene ble oppfylt:

- Hver gang det ble mottatt en forespørsel om at en bevegelse skulle bli gjennomført av den eksterne aksen eller robotarmen, så ble en bevegelseskommando sendt til den tilsvarende komponenten.
- Tilstandsmaskinene som representerer den eksterne aksen, robotarmen og komponenten ansvarlig for å holde styr på systemets status avsluttes ikke.

FDR4 er en "refinement-checker" for prossess-algebraen CSP. Dette programmet ble brukt til å verifisere disse egenskapene. Resultatene viste at systemkravene ble oppfylt for input som tilsier at alt utenfor modellen oppfører seg ideelt. Disse systemkravene ble ikke oppfylt dersom ugyldig input ble tillatt, noe som viser at modellen er avhengig av at den får gyldig input.

Abbreviations

CAD	Computer-Aided Design
CSP	Communicating Sequential Processes
DSL	Domain-Specific Language
EXAX	External Axis (Refers to the two-axis welding table)
FDR	Failures-Divergences Refinement (Model checking software)
GUI	Graphical User Interface
IDE	Integrated Development Environment
IP	Internet Protocol
IPC	Industrial Personal Computer
OLP	Off-Line Programming
PID	Proportional–integral–derivative (controller)
PLC	Programmable Logic Controller
RAM	Random Access Memory
RTDE	Real-Time Data Exchange
STM	State Machine
TCP	Tool Center Point or Transmission Control Protocol
UML	Unified Modeling Language
UR	Universal Robots

Contents

Chapter 1

Introduction

1.1 Motivation

Robotic manipulators are used to perform a wide range of tasks, from surgical operations [1] to spray painting [2]. One of the most prevalent industrial use cases of robotic manipulators is welding. "It is estimated as much as 25% of all industrial robots are being used for welding tasks." [3]. Companies come in many shapes and sizes, and they require different solutions to optimize their productivity. Pioneer Robotics [1] is a Norwegian provider of robotic system integration and tailor-made solutions. One of their products, the IntelliWelder M06, targets small and medium-sized production lines. At companies where the set of tasks being performed varies throughout the year, a flexible robotic system is a good fit. An image of the IntelliWelder M06 can be seen in Figure 1.1. It is a system centered around a robotic arm (UR10e [4]), a welding table (Carpano FIVE MOT [5]) external axis, and a mobile pedestal that allows the UR10e to be moved further away or closer to the external axis (EXAX). The EXAX is used to simplify the movement of the robotic arm. It can also enable the welding of large objects that would normally require a reach beyond what the UR10e provides by itself. Pioneer Robotics are developing a workflow where welding operations are done simply by defining an edge to be welded directly on a CAD model. From there, the system takes care of the planning and execution. One challenge they face is the synchronization of movements between the EXAX and the UR10e. In this thesis, system modeling and formal verification will be used to ensure that the properties necessary for the synchronized operation of the system are satisfied.

[1] `https://www.pioneer-robotics.no/`

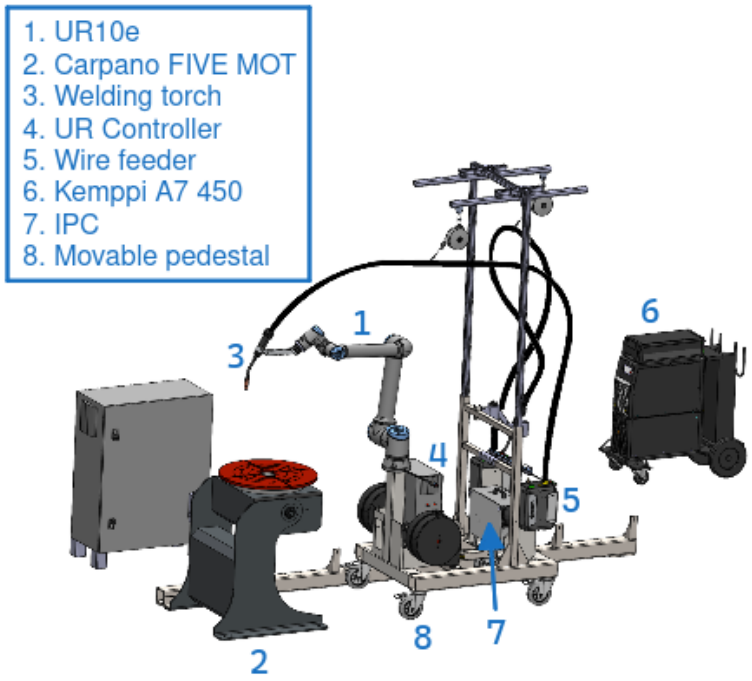


Figure 1.1: The IntelliWelder system with components numbered.

Commonly, before deploying a robot into the real world, the robot is modeled and tested in simulation. A survey of robotics developers found that 84% used simulation during the testing stage [6]. This helps avoid undesired behaviour in the real world, which can save costs and increase the iteration speed of development [7]. Testing at the system level through simulation, or in the real world has been deemed insufficient to cover complex autonomous robotics systems. [8]. This is due to the systems operating in uncertain and dynamic environments where reproducing the exact sequence of inputs that cause a failure is difficult.

Another useful tool for weeding out undesired behaviour in the development stage is model checking. Model checking [9] is a method of verifying that desired properties of a model are maintained during operation. The workflow typically starts with the creation of a model and the specification of the properties that the system must satisfy. Then, the verification procedure is done by exhaustively searching all possible permutations of the states of the model. The result is a proof that the properties are always satisfied for all possible inputs to the model. A few example use cases are applying model checking to ensure that multiple robotic manipulators can work in the same environment without colliding [10], and the safety assurance of an industrial robotic control system using hardware/software co-verification [11].

One of the most notable approaches for modeling systems is the Unified Modeling Language (UML) [12]. There exist domain-specific versions made to suit certain industries and systems. RoboChart [13] is based on a subset of UML. It includes time primitives to capture the real-time nature of robotic applications. RoboChart has a suite of plugins for the Eclipse IDE [14] named Robotool [15]. These provide a GUI that lets the user define RoboChart models. Based on these models, CSP [16] is generated, ready for refinement checking in FDR [17].

1.2 Problem statement

The aim of this thesis is to model a section of the IntelliWelder M06 in RoboChart in order to verify that certain requirements are fulfilled during the operation of the system. Requirements for the system will be defined through RoboChart's domain specific language, and verified through model checking in FDR4, a refinement checker for CSP. If any of the requirements do not hold, a counterexample will be produced by FDR4 that highlights the faulty behaviour. If all requirements are fulfilled, the scope of problematic components in the system will be narrowed down as certain aspects of their behaviour have been formally verified. An ambition is that valuable insight can be gained about both the IntelliWelder M06 and RoboChart, and that there is some value to be gained through the merging of these systems.

1.3 Goals and objectives

- Familiarization with the RoboStar workflow as well as previous case studies where RoboChart has been used.
- Analysing the IntelliWelder solution to identify a suitable section for modeling as well as appropriate abstractions.
- Modeling a well defined section of the IntelliWelder in an adequate and representative manner in RoboChart.
- Defining requirements for the model in a format that allows refinement checking in FDR4.
- Evaluating the results of the model checking, as well as the model itself and the assertions defined.
- Stating recommendations and further work based on the experiences and findings from the thesis.

Chapter 2

Theory

For the modeling process to be successful, it is crucial that well informed decisions are made about how to represent the system. This section will aim to provide the necessary theoretical background, so that this reasoning is sound.

2.1 Robot manipulators and welding

This section gives a brief introduction to terminology and key concepts for robotic manipulators, as well as a short description of the requirements and challenges of performing welding tasks with robots.

2.1.1 Terminology

The reader should be familiar with some core terminology before the theory and application is presented in the following chapters. In this section, some frequently used terms are explained, so that nuances in the definitions can be clarified before proceeding.
`Coordinate systems` is always referring to Cartesian coordinate systems in two or three dimensions.
`End effector` refers to the coordinate system placed at the end of the final link of a robotic arm. It specifies where a tool might be mounted.
`Tool center point` is a coordinate system based on the mounted tool. For a welding torch it will be the tip of torch, which is the position of concern when welding. For a gripper tool, it may be between the "fingers" of the gripper.
`Joint angles` are the positions of the joints expressed as angles. A joint may be

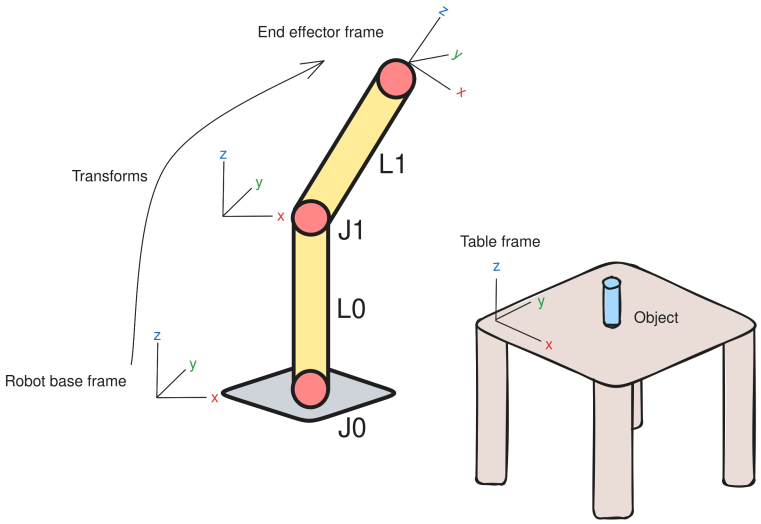


Figure 2.1: Robotic arm and table with frames and transforms.

able to move in the range +/- 90° from its "middle" position.
`Configurations` refers to the different solutions to an inverse kinematics problem. Often, there are multiple combinations of joint angles that result in the end effector being in the desired pose.
`Singularities` are points where the end effector is blocked from moving in certain directions. Commonly, this happens because one or more of the joints are close to their maximum values, meaning they cannot move further in that direction.

2.1.2 Frames and kinematics

Frames are coordinate systems, typically used to describe the positions of objects with some predictable relation to that coordinate system. As an example, imagine a robotic arm interacting with objects on a table. This example is visualized in Figure 2.1. The objects may be positioned at specific positions on the table so that the robotic arm knows where they can be found in the table frame. In that case, it is important to know where the picking tool of the robot is relative to the table frame. The table frame is a coordinate system that could be defined as one of the corners of the table as in Figure 2.1. Points of interest can then be defined in this coordinate system. This is useful since the object is expected to stay stationary in the table frame, but will move in the frame of the picking tool.

When you want to describe the location of an object in one frame relative to another, you can perform a transformation. A transformation defines how

coordinates in one frame relate to coordinates in another. The transformation typically consists of a linear translation, as well as a rotation.

A robotic arm can be viewed as a series of links and joints. Given the lengths of the links and the angle of the joints connecting them, the position of the end effector of the robot can be calculated through a series of transformations [18]. This is called forward kinematics. Target positions for the robot are often defined through the joint angles the robot should have once it has reached its goal. While the robot is moving towards that goal, the feedback received is commonly the joint positions, tracked perhaps by odometers placed in each of the joints. To get information about where the end effector is along the way, forward kinematics can be used to calculate that position. Figure 2.1 also shows the transformations from the base of a robotic arm to the end effector.

When a robotic arm is used in a real world application, it is rarely the case that the joint positions themselves are of interest. The point of interest, or perhaps pose of interest, is that of the end effector. Normally, the task requires the tool of the robotic arm to be in a certain pose, so that it can perform its task. Inverse kinematics is the problem of finding the possible combinations of joint positions that place the end effector of the robot in the desired pose [18]. Often, there are multiple solutions to this problem and this leads to multiple configurations.

2.1.3 Movement types

A set of different movement types are normal when working with robotic arms. A few of the most common ones are moveL, moveJ and moveP. These represent different ways the robot can travel from point A to point B, and they all have different features.

MoveL

The ideal for a MoveL command is to move the end effector in a straight line from point A to point B. That means that the joints of the robot may travel longer distances in the joint space in order to facilitate straight line movement. This typically means more uneven movement in the joint space in order to keep the end effector on this straight line path throughout the movement.

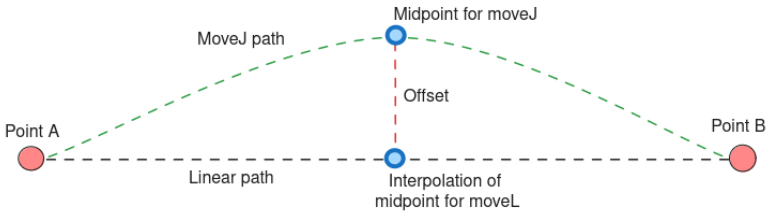


Figure 2.2: Example of how the offset might look in a 2D path from point A to point B

MoveJ

For a UR10e, the MoveJ command can be used with different sets of arguments. In the IntelliWelder it is called with the arguments: target pose, acceleration, joint velocity of leading axis and blending radius. The robot will then use the acceleration argument to accelerate to the correct joint velocities based on the acceleration input. The velocities calculated are based on current joint poses and target joint poses. MoveJ is the only move command used by the IntelliWelder that uses joint velocity. This is because the movement is performed in the joint-space instead of in the tool space. Another consequence of this is that the movement will be linear in the joint space, but not necessarily in the tool space. This can lead to the movement deviating from the linear path between the waypoints. This concept is visualized in Figure 2.2.

Blending

It is often undesirable for a robotic arm to stop momentarily for every waypoint it completes. This makes the movement less smooth than if some blending functionality is used. The UR10e takes blending radius as an argument. This means that once the robot is within a certain distance of the target point, it will look towards the next waypoint and start moving in that direction, instead of continuing to move towards the current waypoint. This means that instead of coming to a halt at the waypoint, it keeps moving toward the next, adjusting the velocity based on the target velocity of the next waypoint. This is useful for the IntelliWelder since jagged movements will result in a bad weld. Therefore, this blending feature is used extensively in the code that is being modeled. One downside of using blending is that it is more difficult to predict exactly how the robot will behave. The velocity with which it moves towards the current waypoint, and when it will switch to the next waypoint is more uncertain. The blending may allow some "shortcuts" between waypoints, since the robot does not move exactly to the coordinates of the waypoint,

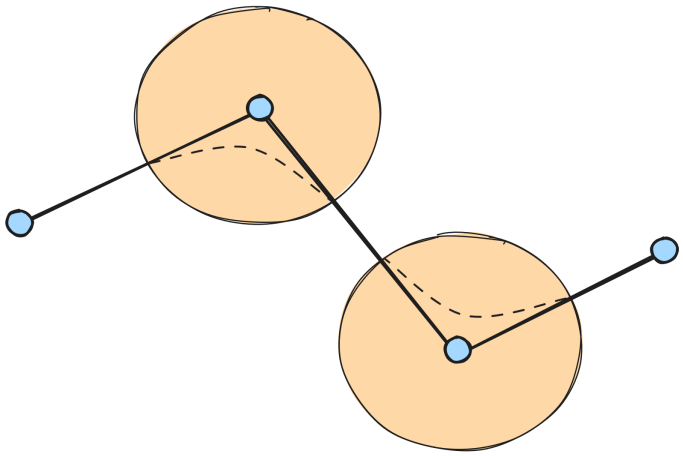


Figure 2.3: An example of a moveP movement with circular blending between waypoints.

but instead a somewhat shorter path. This could lead to the robot arriving early while performing a sequence of waypoints, if no adaptive planning is implemented. A requirement for commands with blending is that the blending radius is sufficiently small, so that there is no overlap between the blending zones of two waypoints. This would cause the skipping of a waypoint, which is usually undesirable [19].

MoveP

The moveP movement type leverages the blending principle, and attempts to plan a circular trajectory when switching from one waypoint to the next. This means that once the end effector is within the blending radius, it will move along a circular path until it is heading in the correct direction for the next waypoint. This is visualized in Figure 2.3.

2.1.4 Welding and robots

Welding is the process of joining two or more objects, typically metals, by melting the edges of the objects so that they fuse together [20]. Normally some filler material is also added to ensure a solid joining of the objects. The melting is most often done by applying a large current to the area being welded, so that the temperature rises enough to make the metal melt. This is a time-sensitive operation, since moving too fast means that the metal does not heat up enough to melt, and moving too slow means that too much of the metal will melt or too much filler material will be applied. In the case of using robotic arms for welding, it means that the tool should move with an even pace in the weld frame throughout

the welding operation. This is given that the object being welded is stationary. This is not always the case, and will be explored further in the next chapter.

2.2 Formal verification and model checking

Many different models are used to describe systems used for computation. Perhaps the most famous is the Turing machine [21]. It is also seen as the most general, as no other model of computation exists that has produced computations that the Turing machine cannot [22]. In this chapter, different approaches to modeling will be explored. Some approaches for formally verifying the properties of models will also be presented.

2.2.1 Transition systems and state machines

In software engineering, transition systems are a go to for explaining the behaviour of systems [23]. These systems represent variables as possible states of the system, where each value the variables can take on is a separate state. The mechanisms that can cause these variables to change are viewed as transitions between states. This gives rise to a common construct known as a state machine. A state machine is a transition system that contains input events. These can trigger transitions in the state machine, and potentially also output events that allow the state machine to trigger events elsewhere. To implement such state machines, it is necessary to have a language in which these logical operations can be expressed syntactically.

2.2.2 Unified Modelling Language - UML

As the field of software development has grown, many common concepts and methodologies have emerged. It became crucial to avoid that different developers implemented similar solutions with slight differences that would make them incompatible. In an effort to unify the models and definitions used, the Object Management Group (OMG) voiced their wish for a standard approach to object-oriented modeling in 1996 [12]. This later became the Unified Modeling Language. Today it is used ubiquitously for modeling systems. Since its creation it has taken on many different forms to accommodate new use-cases. Extensions have also been made to automatically generate code from UML-diagrams [24].

2.2.3 Communicating Sequential Processes - CSP

In 1978 C.A.R. Hoare suggested that the primitives of programming are input, output and concurrency [16]. These primitives in combination with Dijkstra's guarded command [25] have proven adaptable for many categories of software, and are still commonly used through CSP almost 50 years later. CSP is a process algebra, and not directly readable by a machine. CSP_m is a machine-readable version used to interface with other programs. Since the creation of CSP, further progress has been made to capture the diversity of computational tasks being performed. One of the most important additions is capturing the real-time aspects of systems operating in the real world. Early attempts resulted in the development of timed CSP [26]. Today, expressing time-budgets and deadlines for systems that are time-critical is possible through *tock*-CSP [27].

2.3 Tools

The following section provides an overview of the tools used in the modeling and verification of the IntelliWelder system.

2.3.1 Verification tool - FDR

A key feature of such process algebras is the possibility of refinement checking. The most popular refinement checker for CSP_m is FDR [17]. It enables the comparison of processes defined in CSP_m, to check whether one process is a refinement of the other. This bears fruit in the space of modeling, where a common need is to check whether certain requirements hold true during execution. If a model and the requirements for that model are defined as processes in CSP_m, it is possible to check whether the model is a refinement of the requirement. In other words, it can be verified whether the model fulfills the requirement or not. FDR exhaustively checks all possible traces of the model to see if any of them violate the requirement. If that occurs, a counterexample is produced. This counterexample is useful to identify the behaviour that led to the violation. In this thesis, FDR4 [1] is the tool chosen for model checking.

[1] FDR4: `https://www.cs.ox.ac.uk/projects/fdr`

2.3.2 RoboChart

To simplify the process of working with these tools, RoboChart [15] introduces a GUI tool that allows the developer to model systems in an intuitive way. There are a number of useful constructs available in RoboChart. Together they form a basis for modeling complex systems. These models can be directly translated to CSP for refinement checking in FDR4. In the following section, the constructs used in this thesis will be presented and explained.

Robotic Platform

The Robotic Platform (RP) is introduced to abstract away the parts of a system that reside outside of the model. It allows the triggering of external events to be used as inputs to the model. This is a natural approach, as robotics solutions often are modular, and are comprised of multiple layers of abstraction.

Controller

A controller is a construct used to model the behaviour of some section of the system. It has to contain one or more state machines. It can also require and define interfaces and events. The events of the RP can be connected to controller asynchronously.

State Machines

State machines are mainly comprised of states and transitions. These transitions can require an event to be triggered. They can also have a guard that makes the transition conditional. A transition can also be associated with an action. This action could be changing the value of a variable or triggering an event. The behavior of the system can be modularised through state machines.

Operations

In RoboChart, operations can either be provided by the RP, or implemented as part of the model. If it is provided by the RP, it can be called with arguments, without that having any effect on the state of the model. This makes sense if the model is not intended to capture the behaviour of the operation.

Events

An event in RoboChart can have a type. This means that when that event triggers, it arrives with an object of that type. This object can be written to a variable, which can later be used in the state machine, for example as a guard for a transition.

Variables, interfaces, record types and enumerations

There are a few ways to store information in RoboChart models compactly. It is possible to define variables in a similar way to when programming in a language like `C++`. The variable has a type and a name, and can be initialized with a value. Interfaces provide a way to create one-off structures for storing information. They can contain a list of variables and constants that are needed in a state machine. Record types are very similar to structs in `C++`. They enable passing one object containing multiple variables instead of them being separate. Events can also carry record types which makes them useful for modeling updates where multiple variables are received at the same time.

Assertions

When a model has been created in RoboChart, a series of standard assertions are automatically generated. These check the model for properties such as deadlock-freedom, determinism, and divergence. These can cover the needs of some models, but it is also possible to define custom assertions in CSP or in the domain-specific language (DSL) of RoboChart. Assertions defined in this language are automatically translated to CSP that is ready to be checked in `FDR4`.

2.3.3 Delfoi

Delfoi Robotics is a part of Visual Components Group. They provide tools for Offline Programming (OLP) [2]. These tools can be used for 3D layout planning, simulations and the integration of systems. For the IntelliWelder, Delfoi can create an environment consisting of the UR10e, Carpano FIVE, and the CAD model of the object that will be welded. The Delfoi OLP tool lets the user specify an edge directly on the CAD model. That is the edge the user wants to weld along. This edge is used to calculate a series of synchronized waypoints for the UR10e and the Carpano FIVE.

[2] `https://www.visualcomponents.com/olp-products/robotics-olp/`

Chapter 3

Problem analysis

This chapter provides a deeper analysis of the setup and the architecture of the IntelliWelder system. The different components of the IntelliWelder will be presented along with parts of the code. Key concepts relevant to the modeling decisions will also be explained. By the end, a more concrete definition of the scope of the modeling task will be in place. A set of requirements will also be defined, both for the full system, and for the section covered by the model.

3.1 System architecture

This section presents the architecture of the IntelliWelder system, and goes through the process from the definition of the weld to the execution of the movements. The full system architecture of the IntelliWelder is shown in Figure 3.1.

3.1.1 Delfoi

At the top of the IntelliWelder system architecture is Delfoi. It allows the user to select an edge directly on a CAD model, and turns that edge into two sets of waypoints. One set for the Carpano FIVE and one for the UR10e. This offline planning is based on a kinematic model, meaning the focus is on joint positions and end effector poses, rather than a dynamic model focusing on the forces and torques experienced by the system during operation. These waypoints have ideal synchronization. They are planned as one-to-one pairs for the external axis and UR10e. This means that at those waypoint-pairs, the welding torch is placed correctly at the object being welded, based on the corresponding position of the welding table.

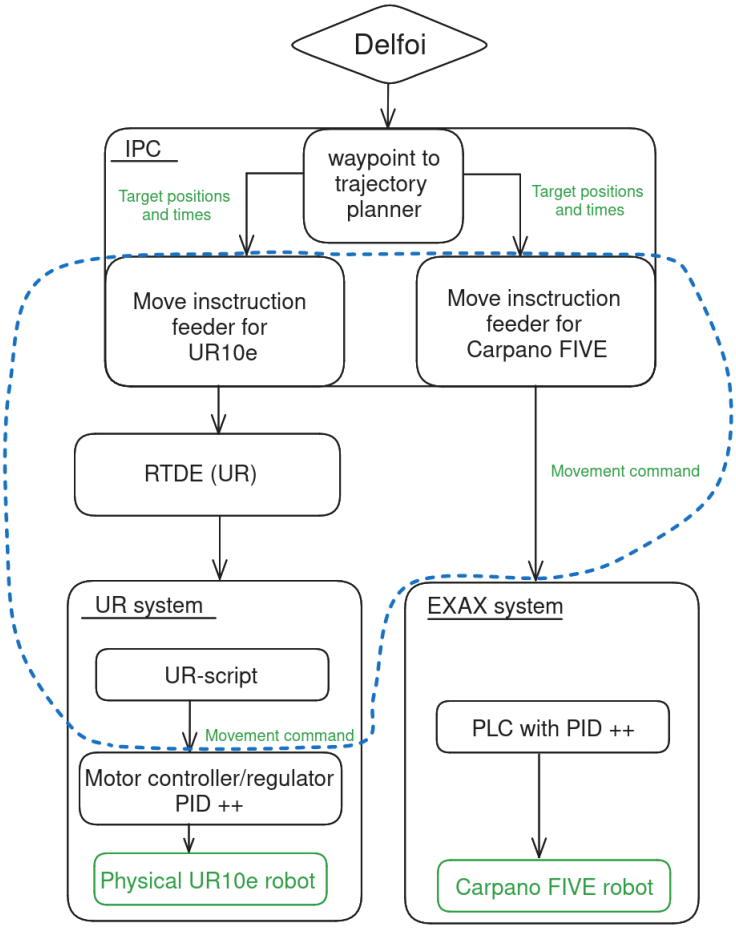


Figure 3.1: The full system architecture of the IntelliWelder. The dotted line shows the section of the system being modeled.

3.1.2 IPC

After this path has been created, it is passed to the Industrial PC (IPC). The lists of waypoints are then processed through `C++` code that turns the waypoints into trajectories. This process is based on the desired forward welding speed as well as a few other parameters such as weaving. These other parameters are assumed not to affect the analysis. As described in the theory section, the speed with which the welding torch moves in the weld-frame is a key property. The waypoints for the external axis are kept the same, and turned into a trajectory, while the waypoints for the UR10e are sampled at a higher resolution, and then turned into the trajectory that will be performed by the UR10e. A consequence of this is that the number of waypoints is no longer the same for the UR10e and the Carpano FIVE, and the UR10e will have more waypoints to iterate through. Figure 3.2 shows an example of what this might look like for two parallel processes, executing three and five waypoints over the same time period respectively. Once the trajectories have been generated, the IPC feeds them as movement requests for the rest of the system to execute.

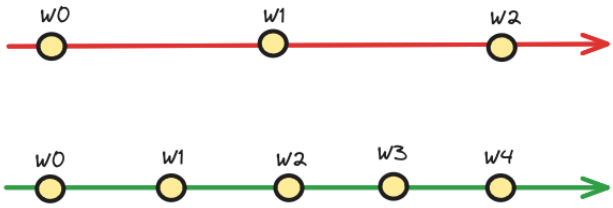


Figure 3.2: Visualization of how different numbers of waypoints can start and finish at the same time. Here, the arrows represent the passage of time.

3.1.3 Programmable Logic Controller - PLC

The control of the Carpano FIVE is performed through a Programmable Logic Controller (PLC). The IPC is responsible for feeding move commands to the PLC. The PLC receives a command that is either based on position and velocity, or just velocity. Then, the PLC regulates the movement through PID-control.

3.1.4 Real-Time Data Exchange - RTDE

The Real-Time Data Exchange (RTDE) is responsible for the synchronization of external applications with the UR10e [28]. It is responsible for relaying messages from the IPC to the UR Controller. Messages are passed through a TCP/IP

connection and written to registers in the UR Controller. These messages are customizable and can be tailored to fit the application. The RTDE is also responsible for controlling the power source for the welding.

3.1.5 UR10e

The UR10e is a cobot [29] produced by Universal Robots. A cobot is a robot that is designed to be safe for collaborating with humans. That places a particular emphasis on monitoring the maximum speeds, forces and torques that are allowed for the robot. In the case of the IntelliWelder system, the UR10e is used to perform the welding task. For this, it has a welding torch as the mounted tool. The UR10e is comprised of both the physical robot and the UR Controller. The physical robot has six joints and a reach of $1.3m$. It weighs $33.5kgs$ and can carry payloads of up to $12.5kgs$. The UR Controller is responsible for running applications written in URScript. The controller also regulates movement through PID control. The URScript used for the IntelliWelder will be presented in the next section. Specifications and a manual for the use of the UR10e can be found online: [4].

3.1.6 Carpano FIVE MOT

The Carpano FIVE MOT is the welding table used for the IntelliWelder. It is also referred to as the external axis or EXAX in this thesis. It is capable of tilting and rotating the top plate of the table about its own axis. The IntelliWelder uses a motorized version, but the Carpano FIVE can also be controlled by hand-wheel. It weighs $340kgs$ and can carry loads of up to $500kgs$. Further specifications for the Carpano FIVE MOT are also found online: [5].

3.2 URScript

The IntelliWelder uses a script written in Universal Robots' programming language, URScript. It is used to select what kind of movement is performed by the UR10e, and to call the movement functions with calculated arguments. This language includes a number of built-in functions, some of which are used in the code for the IntelliWelder. These will be presented and explained in this section along with custom functions that are used in the implementation.

3.2.1 Function - get_target_tcp_pose

This function is a built-in function of URScript that returns the tool center pose at the waypoint the UR10e is currently headed towards. This should not be confused with `get_current_tcp_pose` with returns the pose of the TCP at the current time.

3.2.2 Function - point_dist(a, b)

This is another built-in function. It returns the distance from pose `a` to pose `b`, including rotation.

3.2.3 Function - calculate_offset

As discussed in Section 2.1.3, using moveJ can result in a deviation from the linear path from point A to point B. In the URScript, this deviation is calculated by a function called `calculate_offset`. This function performs inverse kinematics on the `curr_target_pose` and `next_target_pose` to retrieve the joint values of the robot in those positions. It is then possible to find the joint values at the point in the middle of the movement. This is done by finding the middle of the two joint positions for each joint. Then, forward kinematics can be performed on the joint positions at the halfway mark to find the pose of the robot at that point. The halfway point when traveling along a linear path from point A to point point B in the tool space, can be found by using the built-in `interpolate_pose` function of URScript. In addition to start- and end-poses, A and B, this function also expects an interpolation parameter, $\alpha \in [0, \quad 1]$. If `alpha` is set to 0.5, the function will return the position and orientation at the halfway point. This was also shown in Figure 2.2 as the interpolated midpoint. Then, the built-in `point_dist` function can be used to find the offset.

3.2.4 Function - sharp_angle

Certain movement types are undesirable if the movement involves making a sharp turn. By finding the angle between the direction of the current velocity and the heading for the next waypoint it can be determined if the TCP will experience such a sharp turn. It is assumed that the robot is moving straight toward its current waypoint, so that its velocity upon arrival will be in the same direction as the current velocity. An illustration can be seen in Figure 3.3. The angle can now be found based on the formula for the dot product:

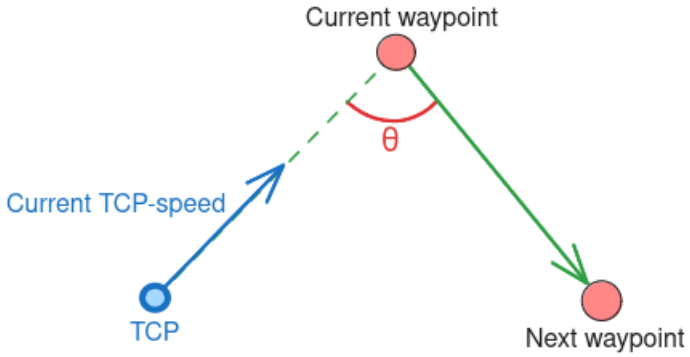


Figure 3.3: Visualization of the angle between current TCP-speed and direction for next waypoint.

$$\vec{a} \cdot \vec{b} = |\vec{a}| \cdot |\vec{b}| \cdot \cos(\theta) \tag{3.1}$$

Then, by rearranging this formula, a formula for the angle between two vectors can be obtained:

$$\theta = \arccos \frac{\vec{a} \cdot \vec{b}}{|\vec{a}| \cdot |\vec{b}|} \tag{3.2}$$

3.2.5 Function - calculate_jv

For moveJ commands, the robot takes the joint velocity of the leading axis as an argument. The leading axis is the axis that has to travel the furthest distance. The velocity for that joint is calculated by first finding the distance between the `curr_target_pose` and the `next_target_pose`. That distance is then divided by `planned_velocity` to find the time budget for the move. The largest joint distance is then calculated based on the inverse kinematics of the `curr_target_pose` and the `next_target_pose`. This distance is then divided by the time budget to find the joint velocity of the leading axis, `jv`.

3.2.6 Function - moveL_with_t

MoveL_with_t is a custom function in the URScript used to call moveL with the arguments `next_target_pose` and `next_target_time`. When these arguments are used in moveL, the UR10e will calculate the velocity necessary to perform the movement within the given target time, and accelerate to that velocity with its maximum acceleration. Since there is no blending argument, the robot will slow

down, and come to a halt at the waypoint. This leads to more jagged movement, which is undesirable for a welding process. Therefore, this type of movement is used as a fallback when the other move commands are considered non-feasible options.

3.3 Pseudo-code

The pseudo-code for the URScript can be seen in Listing 1. Each time a new movement request is received, this code runs. The RDTE has already updated registers with the necessary information for the script. The code begins with updating the required variables by reading from these registers. This happens in the function `update_variables`. The variable `curr_target_pose` is not read from a register, but rather fetched by calling the function `get_target_tcp_pose`.
The first `if`-statement checks whether the target time for the next waypoint has already passed. If so, the current solution logs it as a warning, and continues execution. Next, it is checked whether the command includes a blending radius. If so, the preferred response is to use MoveJ or MoveP. The offset calculation is then performed as previously described. A check is also performed to see if the next movement will involve making a sharp turn.
First, if the calculated offset is above a certain threshold, in this case 0.8mm, a different move command is selected. If it is smaller, MoveJ is considered a good option. In that case, the joint velocity of the leading axis is calculated, and MoveJ is called with the appropriate arguments.
In the case where the offset is above the threshold, the script checks if the next movement involves making a sharp turn. Using MoveP is seen as undesirable if there are sharp turns involved. If that is not the case, MoveP is called. The fallback is to call MoveL_with_t.
In the case where there is no blending, the distance of the movement is calculated. This distance is calculated based on the distance from the `curr_target_pose` to the `next_target_pose`. If the distance is above a threshold of 2mm, MoveL is called with the normal arguments. For short distances MoveL struggles to complete in time, and so MoveL_with_t is called. It is noteworthy that MoveL_with_t is used as a fallback in both cases. This is due to it causing jagged movements as previously described.

```
# New command received
acceleration = 0.8
curr_target_pose = None
next_target_pose = None
curr_tcp_speed = None
curr_time = None
next_target_time = None
blending = None
planned_vel = None
update_variables(curr_target_pose, next_target_pose, curr_tcp_speed,
                 curr_time, next_target_time, blending, planned_vel)
if next_target_time < curr_time
   log("Waypoint is in the past. Skipping waypoint")
elif blending > 0.1
   offset = calculate_offset(curr_target_pose, next_target_pose)
   sharp_corner = check_sharp_corner(curr_tcp_speed,
                    curr_target_pose,
                    next_target_pose) #returns boolean
   if offset < 0.8:
       joint_vel = calculate_jv(curr_target_pose,
       next_target_pose,
       planned_vel )
       moveJ(next_target_pose, joint_vel, acceleration, blending)
   elif not sharp_corner:
       moveP(next_target_pose, planned_vel, acceleration, blending)
   else:
       moveL_with_t(next_target_pose, next_target_time)
else:
    dist = get_distance(curr_target_pose, next_target_pose)
    if dist > 2:
        moveL(next_target_pose, planned_vel, acceleration)
    else:
        moveL_with_t(next_target_pose, next_target_time)
```

Listing 1: URScript pseudo-code.

3.4 Modeling

Now that the system architecture and components have been presented, the scope of the model considered in this thesis can be more accurately stated. This section will show what part of the system is being modeled, as well as stating some of the implications this has for the applicability of the model to the real system.

3.4.1 Section of system that will be modeled

The part of the system being modeled can be seen inside the dotted line in Figure 3.1. The inputs to the model are movement requests. These are sent from the IPC, based on the trajectories that have been planned for the UR10e and the Carpano FIVE. As previously mentioned, the number of waypoints is different, and so they are expected to receive and execute commands concurrently. The model will span the system until it reaches the point of calling movement commands for the Carpano FIVE and the UR10e. From there, the UR Controller and PLC take over the execution of the movements. In terms of responsibility, this means that the planning of trajectories lies outside of the model, and the feasibility of a planned movement is not checked in the model. The model has the responsibility of calling the movement type it considers best fit for each movement request. It should also detect whether or not the system is out of sync, meaning the target time for the movement request it received is already in the past. In the model, this is represented by an out-of-sync event, which is considered a critical failure for the system. The correct execution of the movements themselves is also naturally assumed and, therefore, outside the scope of the model. These are represented by calls to operations in the model.

3.4.2 System-wide synchronization requirements

There are a number of candidate requirements for the IntelliWelder system. For a single robot, it could be possible to split the requirements into "right place, right time". One requirement could be that the robot does not deviate more than a given distance from the planned path. Another could be that the waypoints are all reached within a certain margin of error of the target time. This becomes more complex when the movements of the robots are mutually dependent. If the Carpano FIVE welding table is lagging behind in its movement, the UR10e would ideally slow down, but as previously discussed, welding is a time-sensitive operation, and so

the forward weld speed is a key property. With this in mind, requirements could be specified with respect to the weld frame. The following requirements are suggested:

1. The welding torch always stays within a maximum deviation from the weld frame. This includes both position and orientation.
2. The welding torch always moves forward in the weld frame within a given maximum deviation of the desired forward weld speed.

These requirements are not easy to implement directly. Therefore, they need to be further developed and split into requirements specifically for Delfoi, the IPC planner, the calculation of arguments for the robot commands, and the execution of the movements. They could be extended to include requirements regarding the communication of information and the execution time of code.

3.4.3 Requirements for the modeled section of system

With this in mind, requirements that reside within the scope of the model can be defined. These requirements should aim at ensuring the desired properties of the system. The following requirements are suggested for the modeled section:

R1: The model should catch events that imply that the system is out of sync.

- This places a responsibility on the model to catch certain scenarios where the system is out of sync. This is necessary as a safety precaution since not all problematic cases will be caught by the PLC or UR Controller during execution.

R2: For each movement request received from the IPC, the corresponding robot should receive a movement command unless the model discovers that the system is out of sync.

- This is a natural requirement as it is undesirable for the system to plan a movement that is not executed.

R3: The synchronization properties of the system should not be compromised due to calculations performed in the model.

- This requirement is necessary because arguments such as `jv` are calculated inside the model. If these are not calculated correctly, the movement would be executed incorrectly, potentially causing system failure.

Chapter 4

Method

In this section, the results of the modeling process are presented along with the assertions defined to capture key properties. The chosen abstractions, all state machines, and custom functions are shown and explained. Many of the modeling decisions are based on patterns seen in the various case studies found on the RoboStar website [1]. Examples include an Autonomous Chemical Detector [30], the Alpha Algorithm [31], and a robot performing UVC-treatment of plants [32]. All source code produced for this thesis can be found on GitHub [2].

4.1 RoboChart workflow

The RoboStar group have suggested an idealized workflow for projects using their framework. This workflow can be seen in Figure 4.1. This workflow, presented in [33], shows a complete process, from defining a software model and assumptions about the platform and environment, to eventually reaching a solution that is certified for safe use through model checking.

4.1.1 Idealized workflow

The idealized workflow according to the RoboStar philosophy starts with the definition of a software model in RoboChart in addition to a specification of the assumptions being made about the platform and environment [33]. The first step to defining a model is to identify what variables, events and operations should be provided by the Robotic Platform (RP). This defines the scope of the model,

[1] https://robostar.cs.york.ac.uk/case_studies/

[2] https://github.com/henrik-nordlie/IntelliWelder_RoboChart

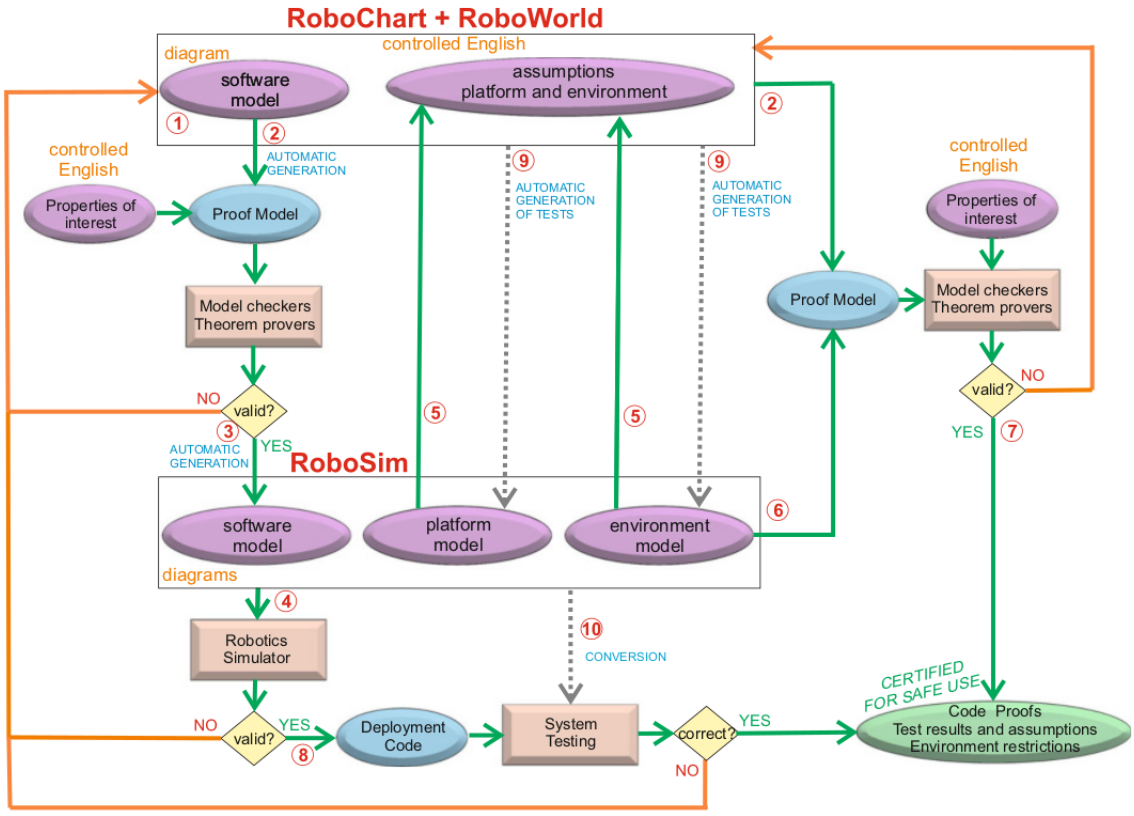


Figure 4.1: Idealized workflow using RoboStar technology. The figure is from [33]. Reproduced with permission from Springer Nature.

cementing what part of the chosen system will be analysed. The model checker will consider all possible sequences of events that the RP can produce and thus the actual behaviour will always be a refinement of this. When the scope of the model has been restricted, the internal logic can be defined using the various constructs available in RoboChart. Once the model is complete, a set of requirements for the model can be defined. These are the maxims that the model should adhere to. If these requirements are fulfilled, the solution can be used to automatically generate a RoboSim counterpart according to the idealized structure.

4.1.2 Actual workflow

Now the chosen system needs to be placed into this workflow. Previously, the section of the system being modeled, as well as the assumptions made about the platform and environment, were described in Chapter 3. The requirements for the model have been defined and will be implemented in the form of assertions. If those assertions pass when tested using the `FDR4` model checker, the model would be ready to move on to the next part of the idealized workflow. That is also where the scope of this thesis ends. In [33] it is stated that ideally, development would begin with the definition of the model in RoboChart. In this thesis, that has not been the case. Instead, the starting point is the existing solution which has a structure and code that has already been determined. Thus, the natural place to continue is to consider the chosen section of the existing solution and define the events and operations that mirror the functionality of the implementation.

4.2 Abstractions and simplifacations

Figure 4.2 shows the model of the RP. It contains three events that can be triggered by the platform, and provides five operations that can be called. The events are defined in the `events` interface and are used to start the system and send new move requests as the previous moves are completed. The operations are defined in two separate interfaces: `ur_ops` and `exax_ops`. The operations correspond to the move commands that the physical UR10e and the Carpano FIVE can perform. These events and operations make use of a number of abstractions that will be presented and justified in these sections. These abstractions are made with **R3** from Section 3.4.3 in mind. It is important to ensure that the synchronization properties are not compromised due to the introduction of abstractions.

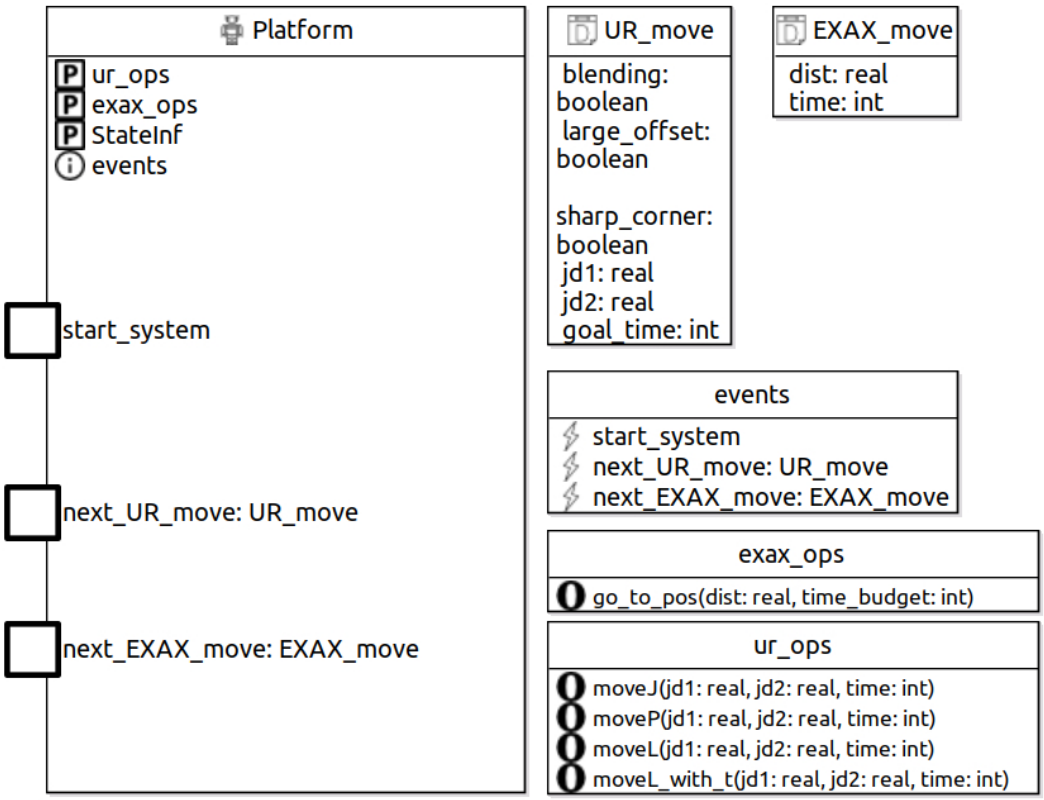


Figure 4.2: RP with its defined events, provided operations and custom record types to represent move commands.

4.2.1 Number of joints

The first abstraction considers the number of joints represented, both for the Carpano FIVE and the UR10e. The Carpano FIVE has two rotational axes for tilt and rotation respectively. During normal operation, however, only the rotational axis of the top plate of the welding table is used. This is because when working with large objects, the ability to rotate the object about its own axis can greatly reduce the traveling distance to reach a point on the object, while tilting the welding table back and forth is often less useful. Therefore, this abstraction is considered very reasonable, and does not constrain the normal usage of the EXAX. The UR10e consists of six joints. In the model, it is abstracted to only having two axes. This is justified mainly on the basis that the movement operations are not performed within the scope of the model, and so issues regarding singularities and other complex physical relations are not expected to be relevant to the section selected for modeling. The functions involving calculations related to kinematics have also been abstracted to boolean values. Therefore, the reduction of the number of joints for the UR10e is not expected to affect what properties are captured by the model. Since there are still two axes, the model will catch any problems related to the fact that multiple joint values are being handled. The method used for the two-axis version is expected to be extendable to six axes as well, only affecting the computational complexity of the verified model.

4.2.2 Move request record types

The UR10e and the Carpano FIVE use different move commands that require different arguments. As described in Chapter 3, there is also a more complex system for selecting the most suitable movement command. Because of this, the model includes record types that include all necessary variables to make these decisions and call the move commands with the required arguments.

EXAX move

The record type for EXAX moves is shown in Figure 4.3. The type only contains two variables since that is all that is needed to call the movement operation. The distance represents how far the external axis needs to rotate. If the positive direction is clockwise, a positive value means to rotate that angular distance clockwise. A negative value means rotating counterclockwise. If the value is zero, the joint can remain still for that waypoint. In this model, a distance of 1 corresponds to moving the joint 60 degrees. A visual representation of this abstraction can be seen in Figure 4.4. The time variable is an abstraction that imagines a time budget instead of an absolute time value for when the waypoint should be completed. For example: if the current time is 35 seconds after system start and the waypoint should be reached at 36 seconds, the time budget will be 1 second. If the previous move command took too long and the current time is 37 seconds after system start, the target time has already passed, and a time budget of -1 seconds will appear.

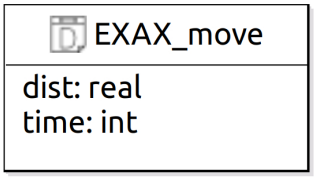


Figure 4.3: Custom record type for EXAX move requests

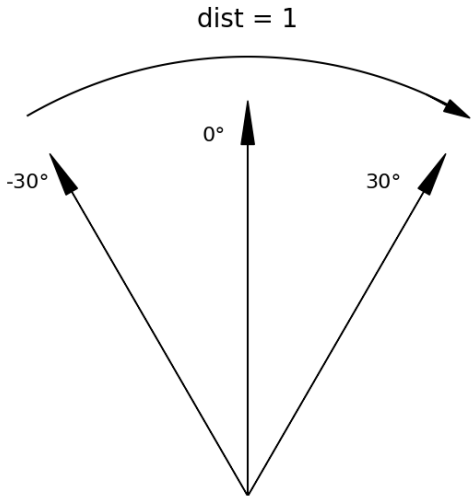


Figure 4.4: Angle abstraction used for all joint distances. A distance of 1 corresponds to moving 60° in the joint space.

UR move

The corresponding record type for UR movement requests is shown in Figure 4.5. This type contains two distances since the UR10e has been simplified to have two axes instead of six like the physical robot has. These are the variables `jd1` and `jd2`. The distances and the `goal_time` variable use the same abstractions as the EXAX move. In addition, there are three boolean variables that are used in the model to make decisions about what move command to select. In the pseudo-code presented in section 1, blending and offset are numerical values rather than boolean values. The blending value is an input to the URScript that could have been numerical in the model, but since the only use of it in the model is to check whether it is larger or smaller than a threshold value, it was simplified to a boolean representing whether or not the input would have exceeded that threshold or not. The method of calculating the offset was shown in Section 3.2.3, and again the numerical value is compared to a threshold. Since the calculation of the offset is not modeled, it can be simplified in the same way as the blending to a boolean stating whether or not there is a "large" offset from the ideal path if using `moveJ`. Large is defined as a distance of more than $0.8mm$ in the URScript. The sharp corner check used to determine whether or not `MoveP` can be used is actually a boolean value in the code, and so this is a natural way to represent it in the model as well. These variables together form the necessary basis for modeling the decision process while keeping the computational complexity as low as possible.

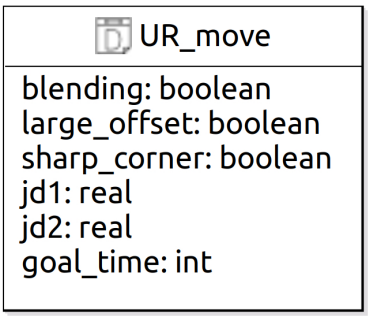


Figure 4.5: Custom record type for UR movement requests

4.2.3 Value ranges

The distance variables in these records are of type `real`. The value range of `real` can be defined in a file called `instantiations.csp` in RoboChart. As mentioned, the distance variables should cover positive, negative and zero-values. To keep the

cardinality as low as possible, the range of `[-1..1]` was chosen. This covers all cases, but does not excessively increase the computational complexity of the model. The variables related to time are of type `int`. This has been chosen so that the range of time can easily be altered without also altering the distance ranges, which is what would happen if they used the same type. The assertions for the model are checked both with `int` being in the range `[0..2]` and the range `[-1..1]`. When the range has strictly positive or zero-values, the model assumes that the UR10e will never arrive too late for its waypoint to the degree where the next waypoint is already in the past. In the other range where the negative value of -1 is included, the model can be checked also for the case where the goal time for the next waypoint has passed. Another type for which the range is defined in the `instantiations.csp` file, is `nat`. Variables of type `nat` are used when considering what waypoint is currently being executed. It is set to have the range `[0..3]`, which means natural numbers from 0 to 3 are considered. This means a maximum of four waypoints can be executed. However, in the model this full range is only used for the `UR` state machine while the EXAX is limited to operating with waypoints in the range `[0..1]`. This is to capture the feature of the actual system that the UR10e and the external axis can have a different number of waypoints as discussed in Section 3.1.2.

4.2.4 Waypoint queuing

One difference between the actual system and the model is how the IntelliWelder performs the planning of the following waypoint during the execution of the current one. In the pseudo-code of the URScript, the variables `curr_target_pose` and `next_target_move` are accurate to how the planning is done, since the UR uses the waypoint it is currently headed towards as the assumed starting point of the next command. Since the robot does the planning beforehand, not at the moment it arrives at the current waypoint, there is no need to worry about the execution time of the planning software given that the planning is done sufficiently early.

4.2.5 Omitted variables

Some of the variables that are defined in the pseudo-code are never passed into the model. This decision keeps the number of variables at a minimum while still capturing all necessary features in the model. An example is that there is no need to include current and target positions if they are only used to calculate a distance. Instead, the distance can be an input directly. For the `UR_move`, that means two

distance variables are passed instead of four position variables. Since some variables are left out and some are transformed to boolean values instead of numerical values, the arguments passed to the operations in the model do not exactly correspond to the arguments in the pseudo-code. For example, the distance is passed as an argument to the operation call instead of the target pose. This should not be an issue since the execution of operations is a part of the robotic platform, not modeled, and therefore do not affect any of the properties of the model.

4.3 RoboChart model

4.3.1 Main

The `main` module of the RoboChart project is shown in Figure 4.6. The module contains one controller and the RP. The controller consists of five state machines, three required interfaces and three events used as input. The RP has the ability to trigger the three events `start_system`, `next_EXAX_move` and `next_UR_move` which in turn will lead to different behaviours in the model. Figure 4.7 shows all of the interfaces defined and provided by the `RP` and required by the `Controller`. The `exax_ops` and `ur_ops` contain the operations called from the `EXAX` and `UR` state machines respectively. The `events` interface include all events that can be triggered by the `Platform`, and `StateInf` contains a variable passed to represent the state of the `System` state machine.

4.3.2 System state machine

The `System` state machine can be considered the top level of the planning. It is responsible for keeping track of whether or not the system has been started, if it is out of sync or if any of the two state machines representing the two robots have finished all of their waypoints. The state machine can be seen in Figure 4.8. Once a state is entered in this state machine, an update to the variable `sys_state` is performed to reflect the current state of the system. The variable has type `SystemState` which is based on the enumeration shown in Figure 4.9. This variable is passed through the interface `StateInf` which is provided by the RP and required in other state machines. The transitions between states in `System` are all triggered by events. Initially, the state machine is in the `wait_for_start` state waiting for the event `start_system`. Once that occurs, the system is in the `working` state until

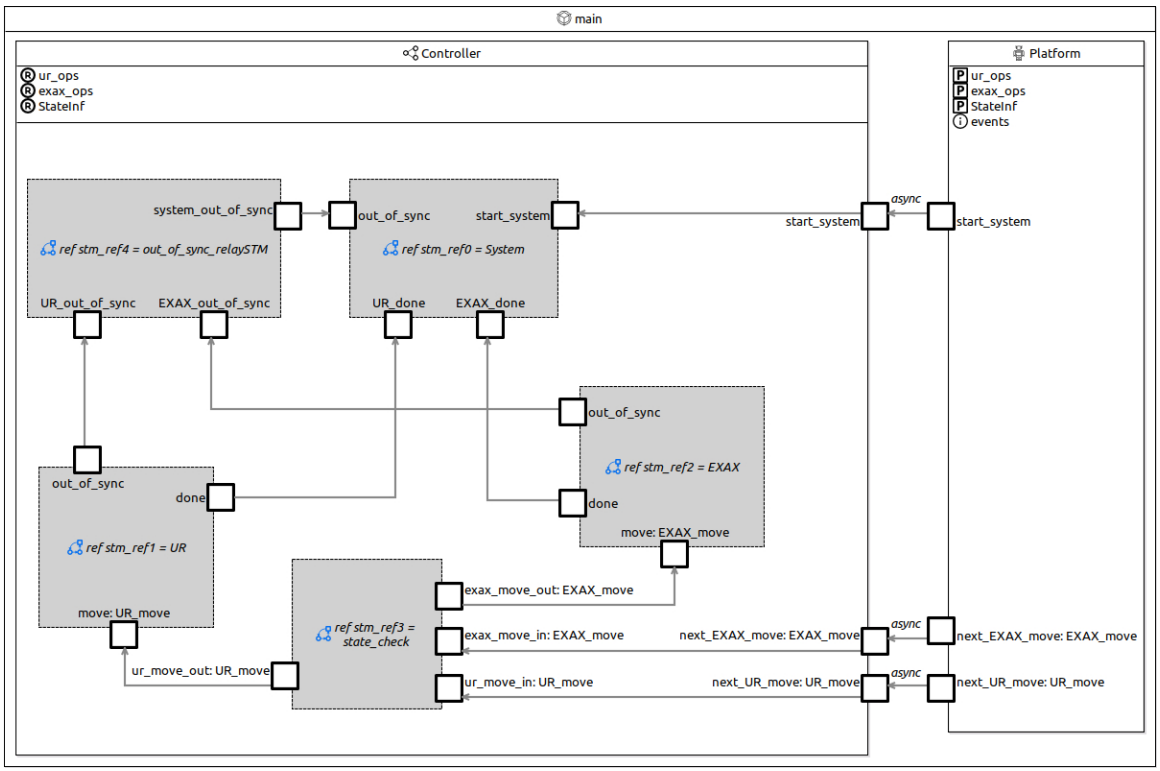


Figure 4.6: Main module of RoboChart project. It includes the Controller with all state machines and the RP.

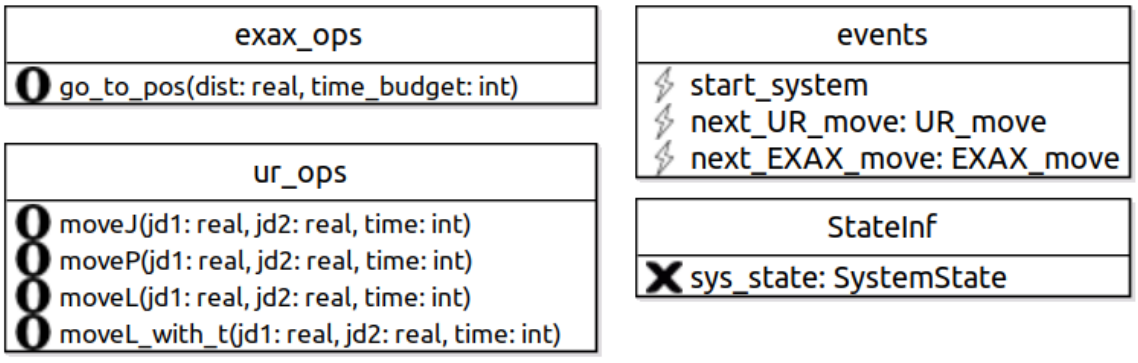


Figure 4.7: Interfaces defined and provided by the RP

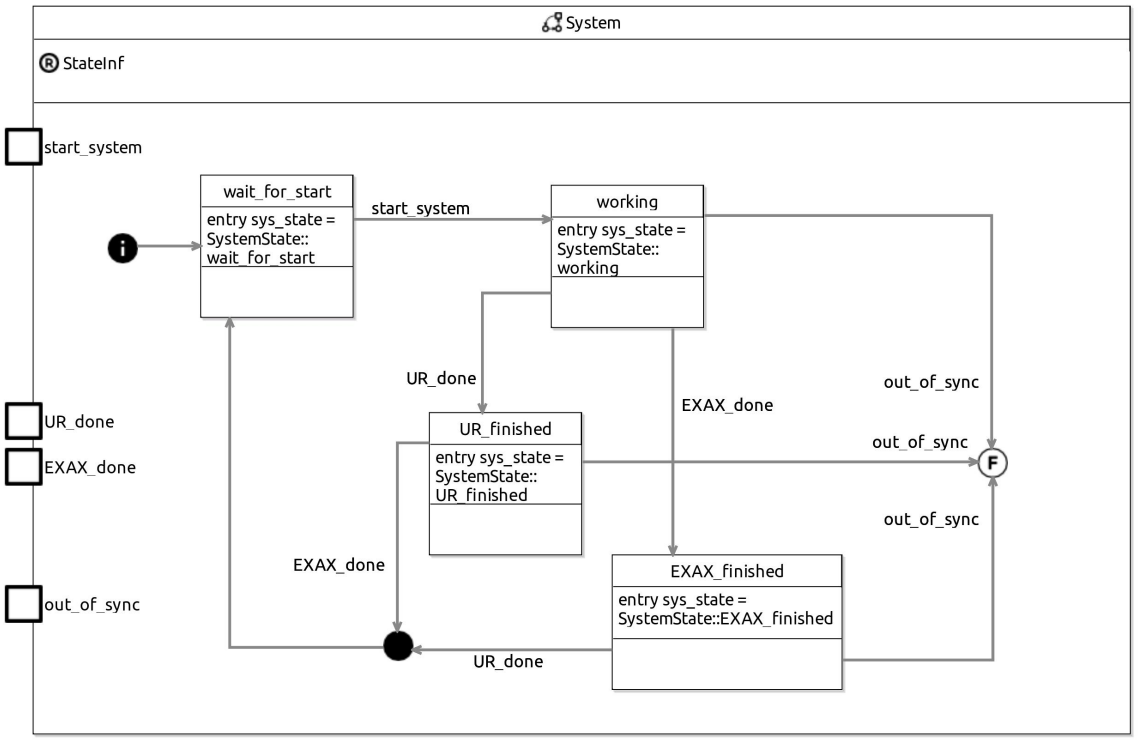


Figure 4.8: `System` state machine. Note the variable update on entry to each state and the possibility of termination through the out_of_sync event.

one of the robots finishes all of their waypoints or an `out_of_sync` event occurs. If the system receives an `out_of_sync` event while it is in the states `working`, `UR_finished` or `EXAX_finished`, it transitions to the `final` node which means the system terminates. It is not possible to define a transition back out of the `final` state. Otherwise, if the event `UR_done` occurs it means the `UR` state machine has finished all of its waypoints and triggered the `UR_done` event. Since the system still has to wait for the `EXAX` state machine to finish, the system waits in the `UR_finished` state until the event `EXAX_done` triggers. The same system is employed for the case where the `EXAX` state machine finishes first. Then, the system waits in the state `EXAX_finished` until the event `UR_done` triggers. Once both the `EXAX` and `UR` state machines have finished all waypoints, the system goes back to the `wait_for_start` state. This can be viewed as one welding operation finishing, and the system is ready for the next task to be defined and started.

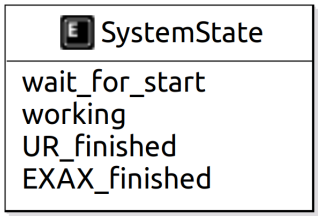


Figure 4.9: Enumeration containing the different states the system can be in. these exactly mirror the states of the `System` state machine.

4.3.3 EXAX state machine

The state machine representing the Carpano FIVE welding table can be seen in Figure 4.10. On initialization, the state machine is in the `wait_for_move` state. Here it waits for a move command to arrive. It initializes the variable `curr_waypoint` to have the value 0, and it currently has `n_waypoints=1`, which means it will go through two waypoints since it starts counting from 0. Note that the type of the variables responsible for counting waypoints is `nat`, which is bound to be in the range `[0..3]`. When it receives a move event, it stores the move request in a variable named `exax_move`. Then, it reaches a junction where it checks whether the time value of the move request is negative or not. If it is negative, that means that the move command has been given a negative time budget to complete the movement, which in this case has been categorized as an `out_of_sync` event. Such an event will make the state machine terminate. If the time budget is zero or positive, it will call the operation `go_to_pos` with the arguments `exax_move.dist` and `exax_move.time`. This operation is defined in the required interface `exax_ops`. After this operation has been called, the system is either finished or has more waypoints to iterate through. This is checked with a guard comparing `curr_waypoint` to `n_waypoints`. If it has reached `n_waypoints` it resets `curr_waypoint` and triggers the `done` event. The done event is relayed to the event `EXAX_done` in the `System` state machine transitioning the system either to the state `EXAX_finished` if the `EXAX` state machine finishes its waypoints first or `wait_for_start` if the `UR` state machine is already finished.

4.3.4 UR state machine

The state machine representing the UR10e robot is shown in Figure 4.11. The structure is quite similar to the `EXAX` state machine, but the `by_position` state has been turned into a composite state named `choose_cmd`. This is due to the system

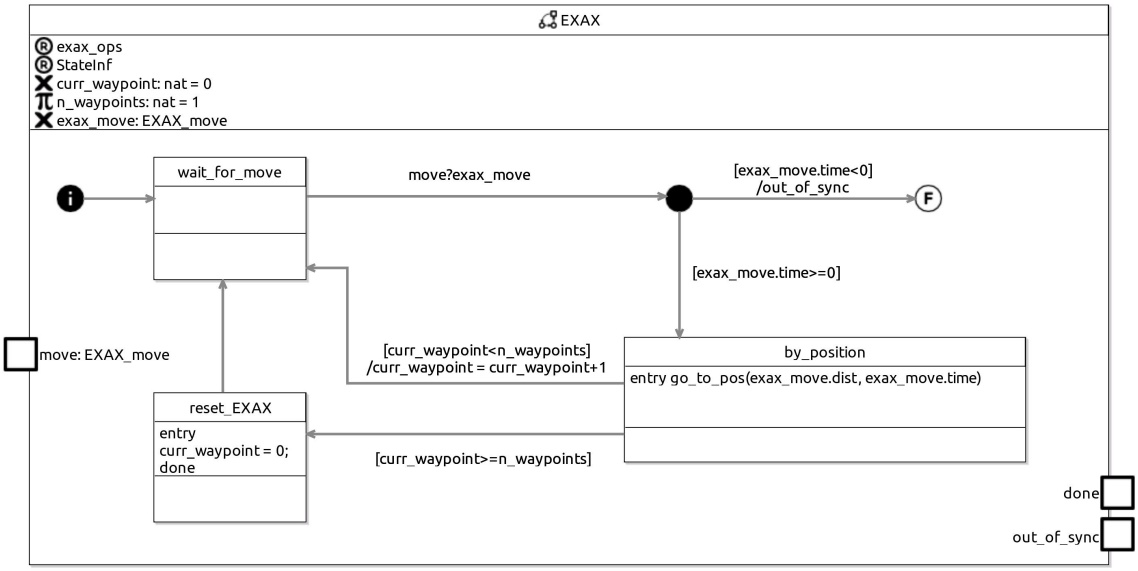


Figure 4.10: EXAX state machine

employed to choose move commands for the UR10e. This decision process was discussed in Section 3.3. In the `choose_cmd` state, the move command is selected based on the boolean values that are part of the `UR_move` record type and the joint distance values `jd1` and `jd2`. The distance values are used in the function `check_big_dist`. The definition of this function and the `abs` function that it calls can be seen in Figure 4.12. The `check_big_dist` function checks whether the absolute value of either of the joint distances is larger than a certain value, in this case the value 1. As discussed in Chapter 3, this is different from calculating the distance in the tool frame, but has been deemed an ample simplification. The calculation of Euclidean distances is not so simple due to the lack of support for certain mathematical operations in CSP and FDR. This check of maximum absolute distance can however be seen as taking the infinity norm of the vectors in joint space instead of the Euclidean (2-norm). The purpose of the check is that given a larger movement, the UR10e is expected to be capable of performing moveL according to the request, but the developers of the IntelliWelder experienced that a "normal" moveL command struggled to complete movements over short distances. Instead, a `moveL_with_t` command is used, which takes only goal position and goal time as the arguments. Based on the decision process in the composite state `choose_cmd`, a move command is selected and the corresponding operation is called. On entry to the `choose_cmd` state, the boolean variable `choosing` is set to true. Due to this, the state machine is trapped in the state until a move command has been selected. In

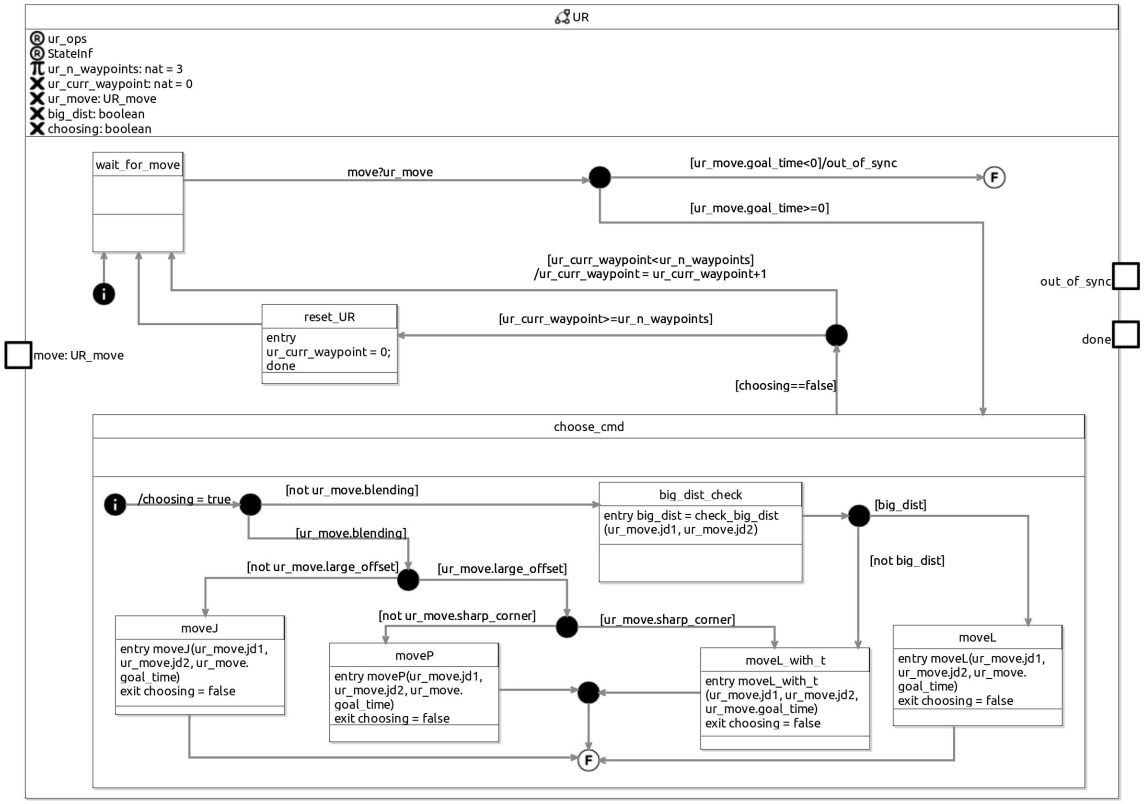


Figure 4.11: UR state machine.

the move command states, after a move operation has been called, the `choosing` variable is set to false on exit before transitioning to the final node of the composite state. Once the variable is set to false, the state machine can transition out. This requirement is enforced by the guard `[choosing == false]` on the transition out of `choose_cmd` state. The same system of incrementing the `curr_waypoint` variable and resetting the value upon completion is used as in the `EXAX` state machine. The same goes for the way the `done` and `out_of_sync` events are called.

4.3.5 out_of_sync relay

Figure 4.13 shows the state machine responsible for relaying the `out_of_sync` events from the `EXAX` and `UR` state machines. This exists because it is not currently possible to connect two different events from state machines to the same input of another state machine in RoboChart. Therefore, a relay state machine has been created that functions as an or-gate that triggers the `out_of_sync` event of the `System` state machine if either the `EXAX` or the `UR` is out of sync.

```
1 -- generate check_big_dist not
2 check_big_dist(jd1,jd2) =
3   let
4     jd1L = abs(jd1)
5     jd2L = abs(jd2)
6   within
7     jd1L >= 1 or jd2L >= 1
8
9 -- generate abs not
10 abs(num) = if (num >= 0) then num else -num
```

Figure 4.12: Definitions of functions in *instantiations.csp*.

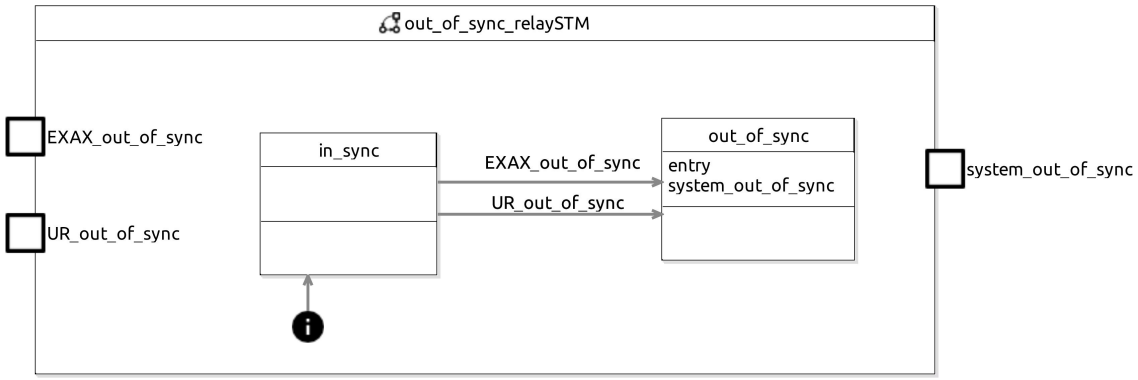


Figure 4.13: This is the state machine responsible for relaying any of the two `out_of_sync` events from the UR or EXAX to the `out_of_sync` event on the System state machine.

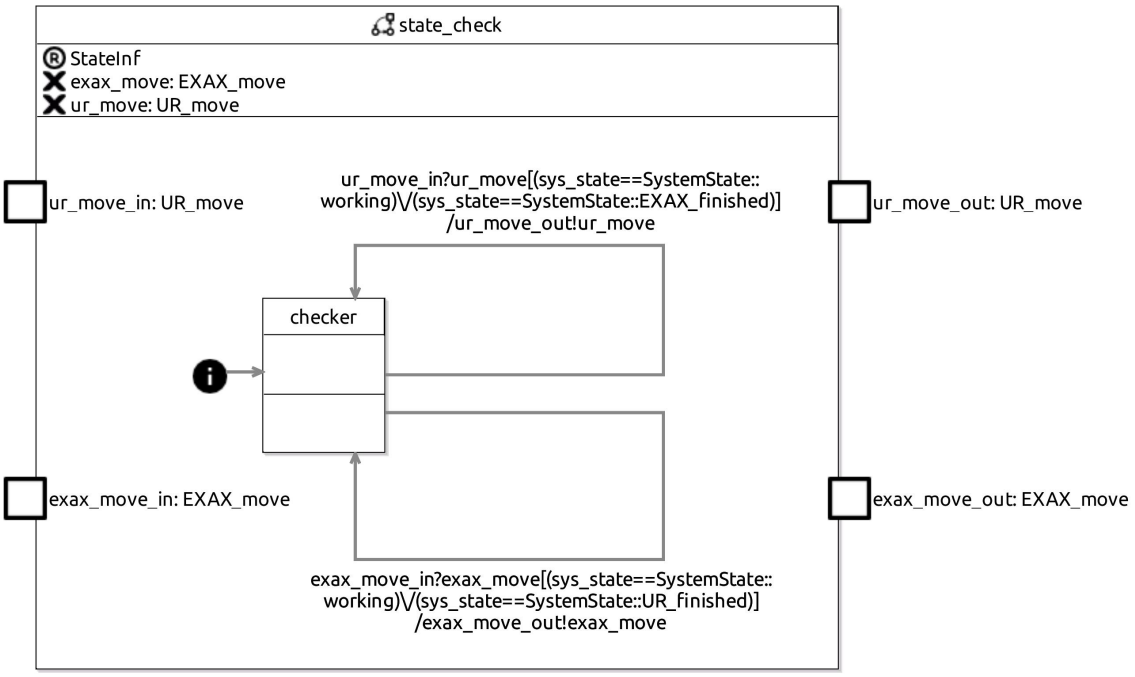


Figure 4.14: State machine responsible for only relaying the move commands of the `UR` and `EXAX` if the `System` is in a state where those state machines should receive commands.

4.3.6 state_check STM

In the `System` state machine, a variable is updated each time the system transitions to a different state. This is done to facilitate this state machine. As seen in Figure 4.14, it has only one state, which transitions to itself whenever it receives a move event. It writes the move command to a variable based on whether the `UR` or the `EXAX` received a move request. It uses a guard to check if the system is in a state that implies the move should be forwarded to the `UR` or `EXAX` state machines. If that is the case, the move is sent to the output event corresponding to the correct state machine. This state machine ensures that no move operations can be executed before the system has been started, and that no more move operations can be executed once all the waypoints of a robot are completed until the system resets and is restarted.

4.4 Properties for verification

Now that the model has been defined, assertions about the system can be made, and thus, certain aspects of the model's behavior can be investigated. The assertions are derived from the previously established requirements set for the selected section of the IntelliWelder system. Ideally, the assertions set for the model should align perfectly with the requirements set for the physical system. Below, these assertions are stated in natural language to later be defined in CSP so that their validity can be tested through model checking.

- Every time a `UR_move` event is triggered by the RP, the model responds by calling one of the defined UR movement operations.
- Every time an `EXAX_move` event is triggered by the RP, the model responds by calling the defined EXAX movement operation.
- The `UR` state machine does not terminate.
- The `EXAX` state machine does not terminate.
- If no `out_of_sync` event occurs in the `System` state machine, the state machine does not terminate.

4.5 Assertions

Since the coreassertions automatically generated by RoboChart do not cover the requirements set for the model, it is necessary to define some custom assertions. These assertions are defined in a file named `IntelliWelder.assertions`. In `.assertions` files, it is possible to define `CSP` blocks of code, as well as assertions in RoboChart's own DSL. This `.assertions` file is then translated to a `CSP` file which can be checked using FDR. RoboChart's DSL uses *tock*-CSP, which uses the event *tock* to mark the passage of discrete time. This enables the definition of timed assertions. For a thorough explanation of `CSP` and *tock*-CSP, refer to [34].

4.5.1 Assertion A1

The first assertion is defined in listing 2. This assertion requires that `EXAX refines SpecA1 in the traces model`. Searching the traces model means looking for safety through the notion of refinement. The definition of `SpecA1` can be seen in listing 3. Notice that it is within a **`Timed`** section, which means it will be treated as a timed process. In `SpecA1` the process `Def` is defined. It starts with `CHAOS(Events)`, which can only be interrupted by `EXAX::move.in`. This leads to a call to `ADeadline`. The definition of `ADeadline` can be seen in listing 4. The function `ADeadline` takes a set of events and a deadline, given as a number of *tock* events, as arguments. It then demands that one of the events in the provided set occur within the deadline. In `SpecA1`, `ADeadline` is called with only `EXAX::go_to_posCall` in the set of events with a zero-tock deadline. This means that the call to the `go_to_pos` operation has to happen immediately when the `EXAX::move_in` event is received.

```
1 timed assertion A1: EXAX refines SpecA1 in the traces model.
```

Listing 2: Definition of **`assertion`** `A1`

```
1 timed csp  SpecA1 csp-begin
2 Timed (OneStep) {
3  SpecA1 = let
4         Def = (CHAOS(Events) [| {|EXAX::move.in|} |>
5         ADeadline({|EXAX::go_to_posCall|}, 0)); Def
6  within timed_priority(Def) }
7 csp-end
```

Listing 3: Definition of `SpecA1` used in **`assertion`** `A1`.

```
1 timed csp  ADeadline csp-begin
2 Timed(OneStep) {
3  ADeadline(E,d) = (CHAOS(Events) ||| Deadline(SKIP,d)) [|E|> SKIP}
4 csp-end
```

Listing 4: Definition of the function `ADeadline` used in **`assertion`** `A1` and **`assertion`** `A3`.

4.5.2 Assertion A2

In order to make sure that **assertion** A1 provides meaningful insight, it is necessary to ensure that the EXAX state machine is timelock-free. This is because there is a trivial case where the process refuses the event tock. The definition of **assertion** A2 can be seen in listing 5. In the listing, EXAX2 is defined as a version of EXAX where EXAX::go_to_posCall is ignored. Note that when the EXAX::D__ process is initialized, it receives the arguments (0, 1). The first argument is an ID-value, and the second is a definition of the value of n_waypoints for the EXAX STM. This represents the number of waypoints the state machine iterates through. The constant n_waypoints being 1 corresponds to two waypoints because of zero-indexing. This is done because the state machine could get stuck in the call to the operation and, therefore, refuse the tock event. This happens since calls to the operations are expected to take no time, and are therefore urgent. This means the liveliness of the state machine was not preserved.

```
1 timed csp  EXAX2 csp-begin
2 Timed(OneStep) {
3   EXAX2 = EXAX::D__(0, 1) \ {| EXAX::go_to_posCall |}
4 }
5 csp-end
6 assertion A2: EXAX2 is timelock-free.
```

Listing 5: Definition of EXAX2 and **assertion** A2.

4.5.3 Assertion A3

Listing 6 shows the definition of **assertion** A3 and SpecA3. This is the equivalent of **assertion** A1 and SpecA1, but for the UR. Since the UR can choose between four different move operations, the set of events seen in SpecA3 now contains all four of those operation calls. The assertion ensures that for each UR_move.in event, one of the four move operation calls needs to be made before any time is allowed to pass.

4.5.4 Assertion A4

In the same manner as in **assertion** A2, it is necessary to ensure timelock-freedom for the UR STM for **assertion** A3 to be meaningful. The definition of **assertion** A4 can be seen in listing 7. The definition of UR2 is very similar to that of EXAX2, but instead of ignoring only EXAX::go_to_posCall it ignores all calls to all four move operations provided for the UR state machine.

```
timed csp  SpecA3 csp-begin
 Timed(OneStep) {
 SpecA3 = let
    Def = (CHAOS(Events) [| {|UR::move.in|} |> ADeadline({|UR::
   moveJCall, UR::movePCall, UR::moveLCall, UR::moveL_with_tCall|}, 0)
   ); Def
    within
     timed_priority(Def)
 }
csp-end
timed assertion A3: UR refines SpecA3 in the traces model.
```

Listing 6: Definition of `SpecA3` used in **`assertion A3`** as well as the definition of **`assertion A3`** itself.

```
timed csp  UR2 csp-begin
Timed(OneStep) {
  UR2 = UR::D__(0, 3) \ {| UR::moveJCall, UR::movePCall, UR::moveLCall
    , UR::moveL_with_tCall |}
}
csp-end
assertion A4: UR2 is timelock-free.
```

Listing 7: Definition of `UR2` and **`assertion A4`**.

4.5.5 Assertion A5

In listing 8, **`assertion A5`** is defined. It simply demands that the `EXAX` STM does not terminate. This assertion is expected to pass given that no `out_of_sync` event occurs.

```
assertion A5: EXAX does not terminate.
```

Listing 8: Definition of **`assertion A5`**.

4.5.6 Assertion A6

This assertion is exactly the same as **`assertion A5`**, but for the `UR` STM. Its definition can be seen in listing 9.

```
assertion A6: UR does not terminate.
```

Listing 9: Definition of **`assertion A6`**.

4.5.7 Assertion A7

To define an assertion that ensures that the `System` STM does not terminate, a modified version is necessary. Therefore, a constraint is placed on the `System::D__(0)` process. During model checking, all possible combinations of inputs are searched. Since the components are verified in isolation, the `System` STM is unaware that if `out_of_sync` never triggers in any of the other two state machines, it will not trigger here. In listing 10, the definition of `SysTerminates` can be seen, as well as the definition of a process `Stop`, and **`assertion A7`**. The process `SysTerminates` is based on another process named `SysConstrained`. This is a version of the system where `out_of_sync` events are skipped. Then, the `SysTerminates` process listens only for the termination event of the `System` state machine. This is done by hiding all events except `System::terminate` using the `|\` operator. If this happens despite the `out_of_sync` event being ignored, the assertion should fail. This is done by comparing `SysTerminates` to the process `Stop`. This process just corresponds to the CSP process **`STOP`**, which is a deadlock. This means that the `Stop` process can never perform any events before terminating, and so by demanding that **`SysTerminates refines Stop in the traces model`**, it can be ensured that this process never performs `System::terminate`, and thus never terminates. The assertion is expected to always pass since the `out_of_sync` event is being ignored. Still, it shows that in the cases where it does not occur, the `System` state machine will not terminate.

```
1  timed csp  Systerminates associated to System csp-begin
2   SysConstrained = (System::D__(0) [| {| System::out_of_sync |} |] SKIP
       )
3   SysTerminates = (SysConstrained ; System::terminate -> SKIP) |\ {|
      System::terminate |}
4  csp-end
5
6  csp Stop csp-begin
7      Stop = STOP
8  csp-end
9
10 assertion A7: SysTerminates refines Stop in the traces model.
```

Listing 10: Definition of `SysTerminates`, `Stop` and **`assertion A7`**.

4.6 Checking the assertions

In the next chapter the results from checking the assertions will be presented. They are checked for two different ranges for the type `core_int`. A sanity check is also performed to verify the correctness of some of the assertions.

4.6.1 `core_int` value ranges

When the range is defined as `[0..2]`, the results will show how the system reacts if it does not receive negative time budgets for the movement commands, meaning the movements are always performed as planned. When the range is defined as `[-1..1]`, the results will show how the system reacts to negative time budgets. These negative time budgets can occur due to an infeasible plan being received from Delfoi, or due to incorrect execution of movement operations by the UR10e or the Carpano FIVE. In general, this value range is meant to cover the case of erroneous input.

4.6.2 Purpose of sanity check

The reason for performing a sanity check is to ensure that certain assertions are defined correctly. Specifically if `assertion 1` and `assertion 3` pass, they can be validated by removing calls to operations in the model. This should cause them to fail, since there should be no traces where a movement operation is not immediately called after receiving a movement request.

4.6.3 Specifications of PC used for checking assertions

Because of the computational demands of checking some of the assertions, a dedicated computer was used. It has an AMD Dual EPYC 7501 (2*32 cores) processor and 2TiB of RAM. The RAM is especially relevant as a lot of memory is required to hold the information generated during assertion checks.

Chapter 5

Results

This chapter will present the results from all checks performed on the model as it was presented in Chapter 4. Initially, the assertions are checked with only positive time budgets. Then a sanity check is performed with a modified model, still with only positive time budgets. This is a way to ensure that the assertion catches flaws in the model as expected. The assertions are then checked with negative time budgets allowed. This, to represent the case when the input to the model is non-idealized or erroneous. Since RoboTool generates two `CSP` files, one for timed assertions and one for untimed assertions, the assertions that are untimed show up in both files. These assertions are therefore checked both as timed and untimed assertions so that the results can be compared.

5.1 Results with `core_int` in range `[0..2]`

Firstly, all assertions will be checked with only positive time budgets. `assertion 5-7` are checked both in the timed assertions `CSP` file, and in the untimed version.

5.1.1 Normal model

Table 5.1 shows the results when checking all of the assertions defined in `IntelliWelder.assertions` with the range for `core_int` set to `[0..2]`. The format for the table follows that of [11]. It can be seen that all assertions pass and that the number of states visited and transitions made is the same for all assertions associated with the same state machine.

Assertion	Result	Elapsed Time			Complexity	
		Compilation	Verification	Total	States	Transitions
A1	✓	10.79s	0.34s	11.13s	228	713
A2	✓	11.10s	0.32s	11.42s	228	713
A3	✓	14.42s	0.51s	14.93s	5,060	17,001
A4	✓	14.15s	0.57s	15.72s	5,060	17,001
A5	✓	0.12s	0.39s	0.51s	228	713
A6	✓	0.12s	0.60s	0.72s	5,060	17,001
A7	✓	0.64s	0.55s	1.19s	32	197

Table 5.1: Results of model-checking all assertions in FDR with only positive values for `core_int`.

Assertion	Result	Elapsed Time			Complexity	
		Compilation	Verification	Total	States	Transitions
A5	✓	0.98s	0.47s	1.45s	156	557
A6	✓	9670.82s	0.48s	9671.30s	3,380	13,621
A7	✓	768.59s	0.45s	769.04s	32	165

Table 5.2: Results of checking the untimed assertions with only positive values for `core_int`. Only `assertion 5-7` are untimed, so the others are unavailable here.

Table 5.2 shows the results of checking the untimed assertions. They all pass, but `assertion A6` and `assertion A7` take much longer than when checking them in the timed assertions. It can also be observed that the number of states and transitions are different when comparing the timed and untimed assertions.

5.1.2 Sanity check with modified model

As a sanity check to see if `assertion A1` and `assertion A3` work as they should, one of the calls to the operation in the model can be removed. Since the `assertion A1` and `assertion A3` demand that an operation from a defined set is called for each move event received, the assertions would be expected to fail. This should produce a trace that is logically connected to the missing operation. In Figure 5.1, the `choose_cmd` state of the `UR` state machine can be seen. It has been altered so that there is no call to the `moveJ` operation. When `assertion A3` is checked in FDR4 it fails and the counterexample shown in Figure 5.2 is produced. By clicking on the Specification Behaviour it is possible to further investigate the case that made the assertion fail. In Figure 5.3, the move request that was received to make the assertion fail can be seen. The boolean values of the movement request are `true, false, false`. If the decision process of the composite state is followed,

it is clear that if the first boolean is true and the second is false, the state machine will end up in the `moveJ` state. Since there is no call to a move operation in this state, the assertion fails.

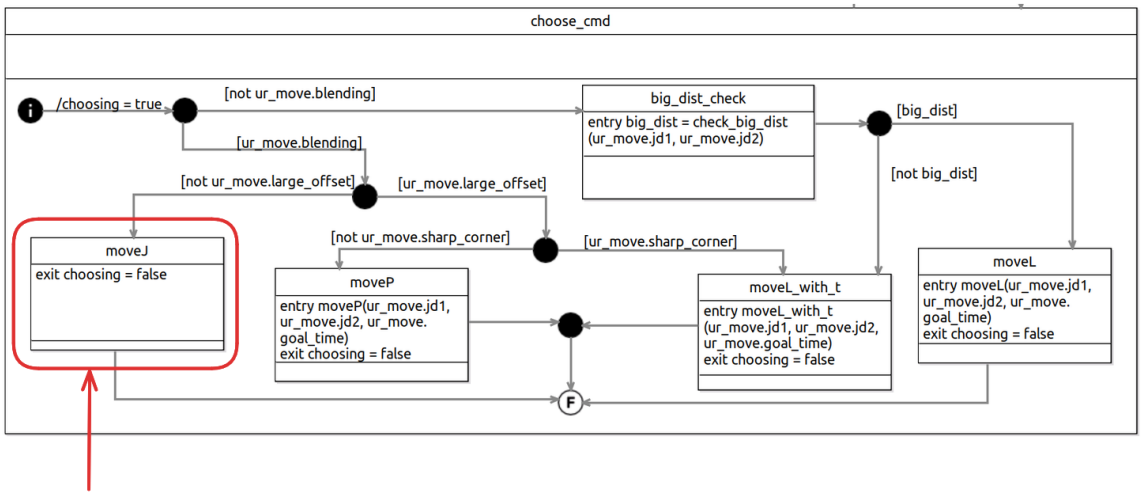


Figure 5.1: The composite state of the `UR` state machine with the call to `moveJ` removed.

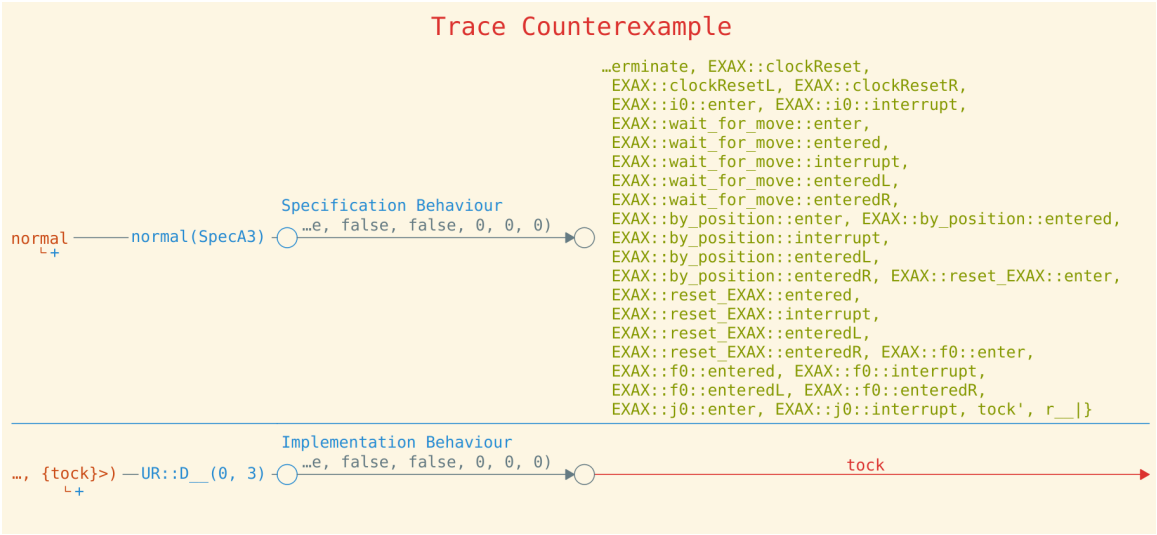


Figure 5.2: Counterexample produced with call to `moveJ` operation missing.

```
Trace to Behaviour
UR::move.in.(true, false, false, 0, 0, 0)
```

Figure 5.3: Trace to behaviour that causes `assertion A2` to fail.

Assertion	Result	Elapsed Time			Complexity	
		Compilation	Verification	Total	States	Transitions
A1	X	11.16s	0.26s	11.42s	11	87
A2	X	10.94s	0.28s	11.22s	99	388
A3	X	15.23s	0.26s	15.49s	75	1,235
A4	X	14.19s	0.35s	14.54s	931	4,412
A5	X	0.10s	0.40s	0.50s	153	554
A6	X	0.12s	0.48s	0.60s	1,497	6,329
A7	✓	0.70s	0.58s	1.28s	32	197

Table 5.3: Results of model-checking all assertions in FDR with both positive and negative values for `core_int`.

5.2 Results with `core_int` in range `[-1..1]`

Next, the assertions are checked with the type `core_int`, including negative values. Again, both the timed and untimed assertions are checked to compare the results.

5.2.1 Timed assertions

Table 5.3 shows the results when checking all of the assertions with the range of `core_int` set to `[-1..1]`. All of the timed assertions fail apart from `assertion A7`. This is not unexpected, since `assertion A7` has the constraint that it skips `out_of_sync` events. The elapsed times are similar when compared to checking the timed assertions for a range of the same size, but with only positive values as was done in Table 5.1. The number of states visited and the number of transitions is lower for all assertions apart from `assertion A7`. This is expected since the assertion check terminates upon the discovery of a counterexample.

5.2.2 Counterexample trace for `assertion A5`

An example of a trace to one of the failed assertions can be seen in Figure 5.4. It shows that the implementation behavior showing the `EXAX_move` has a negative value for the time-variable and that, as a consequence, the trace includes an `out_of_sync` event, which leads to termination.

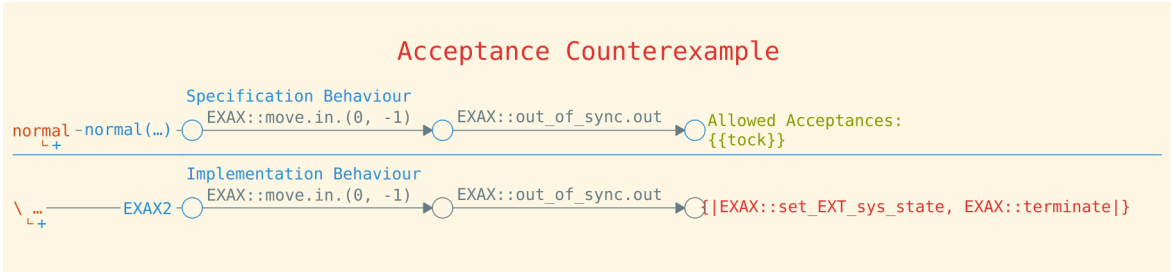


Figure 5.4: Trace showing a counterexample to assertion `A5 - EXAX does not terminate`.

5.2.3 Untimed assertions

Table 5.4 shows the results of checking the untimed assertions with negative time budgets included. It can be seen that **`assertion`** `A5` and **`assertion`** `A6` fail, and **`assertion`** `A7` pass. All of this is expected as the negative time budget will cause `out_of_sync` events and consequently, termination. Again, **`assertion`** `A7` passes because of the constraint, skipping the `out_of_sync` events.

Assertion	Result	Elapsed Time			Complexity	
		Compilation	Verification	Total	States	Transitions
A5	X	750.57s	0.40s	750.97s	123	466
A6	X	9736.80s	0.41s	9737.21s	1,147	5,409
A7	✓	751.78s	0.41s	752.19s	32	165

Table 5.4: Results of checking the untimed assertions with both positive and negative values for `core_int`

Chapter 6

Discussion and further work

This chapter discusses the results and their implications. It presents other approaches to the modeling, suggests potential improvements to the suite of assertions, and discusses the limitations of the current solution. It also includes recommendations for others performing similar work in addition to suggesting further work.

6.1 Implications of results

To begin with, the results obtained in the previous chapter will be further discussed. Their significance with respect to the requirements defined in Section 3.4.3 are discussed, and interesting findings are presented.

6.1.1 Timed assertion results with `core_int` in the range `[0..2]`

All assertions passed, given that the time budgets were always positive or had the value of zero. This was a desirable result for verifying the synchronization properties of the model. Since `assertion A1` and `assertion A3` passed, it can be determined that a movement operation is always called for each movement request received. This means that no movement requests are lost due to faulty logic in the model, serving as an example of a non-trivial result. To relate this back to the requirements set for the model in Section 3.4.3, it indicates that **R2** is satisfied. To make sure that these assertions were not trivially satisfied, `assertion A2` and `assertion A4` also needed to pass, which they did. As for the assurance against termination, `assertion A5-A7` show that the `UR`, `EXAX`, and `System` state machines do not

terminate. For the UR and EXAX state machines, this implicitly indicates that no out_of_sync event occurs. For the System state machine it shows that if no out_of_sync event occurs, it does not terminate. In the System state machine this event occurs either by the UR state machine or EXAX state machine triggering it through their out_of_sync events. Therefore, **assertion** A5-A6 passing implies that the out_of_sync event would not trigger in the System state machine if the entire module was used.

6.1.2 Sanity check with core_int in range [0..2]

The sanity check performed on a model without the call to the movement operation moveJ showed that **assertion** A5 failed. This strengthens the belief that the assertion works as intended and that it would catch cases where a movement request did not lead to a movement operation call. Again, this is related to ensuring that **R2** from Section 3.4.3 is met. It is, however, interesting to note the necessity of a sanity check. In the idealized workflow shown in 4.1, label 9 refers to the automatic generation of tests from RoboChart models [33]. The generation of test cases through mutation of RoboChart models has been explored in [35]. This approach could potentially produce a test that would serve the same purpose as this sanity check.

6.1.3 Untimed assertion results with core_int in range [0..2]

A surprising result when checking the untimed assertions was the compilation time. When checking untimed assertions, FDR does not have to include calls to the tock event, hence, model-checking should intuitively be simpler than the timed assertion checks. Despite this, the compilation of **assertion** A6 took 9670.82s for the untimed version, while it took merely 0.12s in the timed assertions file. This is a factor of almost 10^5. It was also shown that **assertion** A7 took much longer in the untimed version. The exact reason for this has not been pinpointed. In personal correspondence with one of the developers of RoboChart, Pedro Ribeiro (Departement of Computer Science, University of York, May 2024), it was suggested that it could be related to a difference in how the timed and untimed semantics are represented. The timed semantics may be more efficient for FDR to compile. His experience was that FDR could compile processes in a very inefficient way because of how they were constructed, despite them being logically equivalent. Another thing to note is the fact that the number of states and transitions visited is lower in

the untimed version. This is because extra states and transitions appear due to the inclusion of the *tock* event when checking timed assertions.

6.1.4 Timed assertion results with `core_int` in range `[-1..1]`

The timed assertion results were as expected with negative time budgets included. All assertions failed, apart from `assertion A7`. On inspection of the counterexamples, the traces included a call to the `out_of_sync` event followed by termination. The fact that `out_of_sync` events were caught is positive because it shows that **R1** from Section 3.4.3 is being met. The time spent was similar to the timed assertions with `core_int` in the range `[0..2]`. This is reasonable since the cardinality is equal. The time taken for verification was somewhat shorter than for checking the range `[0..2]`. This is also logical since the verification terminates upon the discovery of counterexamples. These failures also indicate that the model is input-dependent and that if there exist issues upstream in the system, the synchronization properties of the model are not maintained.

6.1.5 Untimed assertion results with `core_int` in range `[-1..1]`

The same phenomenon occurs in these untimed assertion results as in the untimed results with `core_int` in range `[0..2]`. The time taken is much longer for `assertion A6-A7`, but with this configuration, `assertion A5` also takes a much longer time to compile than for the timed assertion check.

6.2 Limitations

The following section will discuss the known limitations of the current solution. Those include the use of abstractions when modeling the system, the interpretation and translation of existing code, as well as the limitations that exist in the software used for modeling.

6.2.1 Abstractions

There are a number of abstractions involved in the process of turning the real system into a RoboChart model. Some of these include reducing the granularity of the angular values, decreasing the number of joints of the robots, and converting time values based on a clock to relative time budgets for each movement. Ideally, an

abstraction can reduce the complexity of a system without compromising the key properties in the process. It can, however, be difficult to be sure that this does not happen. Many of the functions in the URScript are also simplified through abstraction. Capturing these in the model would cover the full functionality of the URScript, which could uncover errors in the calculations implemented. This was not done in the thesis because it is challenging to capture complex mathematical operations in RoboChart while using abstractions to simplify the numerical values of the system. This has been discussed in a previous publication by the RoboStar team: "The issue with abstraction and arithmetic raises concerns regarding the use of model checking to deal with properties related to complex numeric calculations. For this class of properties, the use of theorem proving is likely to be more fruitful." [33].

6.2.2 Assumptions

The assumptions that are made can be split into assumptions about the outside of the model, and the assumptions made about the internal components of the model. For the outside of the model, we assume that the planned waypoints and goal times form a request that can feasibly be performed by the physical system, given that a suitable movement command is selected. This means that there are no issues due to singularities or conflicting configurations for the robotic arm, and that the accelerations and target velocities are within the physical capabilities of the robots. It is also assumed that the precision of the UR10e and Carpano FIVE is not a limiting factor. For the modeled section, an assumption is that the custom functions of the URScript are correct. These are simplified to boolean values in the model. The functions are used to make decisions about what move command to select. Since they are not modeled and verified, their correctness can not be ensured. Another assumption is that there are no issues regarding the time required to make computations within the modeled section of the system. One argument supporting this assumption is that the UR Controller, which arguably performs the most complex computations, uses a queuing mechanism. It always plans the following waypoint during the execution of the current one. This should ensure that it never pauses in order to process its next move command. There are other ways to avoid the issue of having to perform instantaneous calculations. A suggested solution is the Almost ASAP approach [36].

6.2.3 Interpretation of existing code

When modeling an already existing solution, part of the process is interpreting code and making a model that reflects the code as well as possible. Ideally, this translation would be done in such a way that there is no difference between the logic of the RoboChart model and the code, but this is not likely to be the case. Therefore, this potential gap between the source code and the created model should be acknowledged.

6.2.4 Limitations of software used

Model checking can be quite computationally expensive, and so checking assertions can take a long time. This can slow down the iteration speed of the development. It can also be a limiting factor because of the hardware specifications that are necessary for the checking. During this thesis, not all assertions could be checked on a laptop, and a dedicated computer had to be used. A limitation when learning to use RoboChart is that there is a limited amount of material available in the form of tutorials. Some can be found, but commonly the use cases found on the website had to be explored to find examples of how the software can be used. Luckily, help was easily available from the University of York throughout the process.

6.3 Further work

This section will consider natural further work based on the findings of this thesis. The main consideration is the further development of the solution presented in this thesis, but potential further work for the IntelliWelder system and some thoughts for further improvement of RoboChart are also mentioned.

6.3.1 Modeling

Modeling is a process involving making a number of decisions. When modeling using RoboChart, this involves deciding what the variables, operations and events of the RP should be, what abstractions are used to represent the real system, and what assertions should be defined for the model. Firstly, the events and operations that were defined also define the section of the system that will be modeled. The selected section captures a part of the system, but far from everything. "An important factor in the successful application of CSP and FDR has been a high

degree of selectivity in the choice of problem to tackle. The scope must be sufficiently well-defined to be able to isolate a portion of the system to treat, while being sufficiently complex that there is benefit to be gained from the investment of effort involved."[37]. There are still many components of the system that could be modeled and formally verified. For example, the execution of movement operations by the UR10e and Carpano FIVE could be modeled. The offline programming in Delfoi could also be modeled so that requirements could be set for the generation of waypoints. This could ensure properties related to the feasibility of the planned movements. If the requirements for the system defined in Section 3.4.2 are compared to the requirements for the model defined in Section 3.4.3, the connection cannot be made directly. The requirements for the model are defined based on what can be captured within the selected scope for the model. They state requirements that are expected to be necessary for the synchronized operation of the full system, but more work needs to be done to bridge the gap between these lists of requirements.

6.3.2 Assertions

While the current set of requirements ensures some important properties of the system, many more could be defined to verify other aspects. It could, for example, be checked that all states are reachable, which should be the case for this model. Assertions to ensure properties about the order of events could also be defined. For example, no calls to movement operations should happen before a movement request has been received. Also, the `System` state machine should not be able to reach the states `UR_finished` or `EXAX_finished` before it has been in the `working` state.

6.3.3 Continuation of RoboStar workflow

Eventually, a natural goal for this project would be to move the model further along in the RoboStar workflow described in [33] and depicted in Figure 4.1. This would mean creating a RoboSim block diagram to represent the platform. Here, the operations that are called in the RoboChart would be modeled along with the triggering of events. Eventually, the goal would be to bring the whole solution into a simulation for further validation.

6.3.4 Further work for the IntelliWelder product

There are also opportunities for further development of the IntelliWelder system. Hopefully, the insights gained during the analysis of the code and the modeling process can contribute to the improvement of the product. Perhaps the requirements defined for the full system, as well as the requirements for the modeled part of the system could prove useful for the development process. As described, the IntelliWelder plans a discrete set of waypoints along the edge that will be welded. It is also possible to use an entirely different approach. One could imagine a PID controller with feedback control for the current position, as well as feedforward control that looks at the projected positions of the UR10e and Carpano FIVE in the near future. This approach is perhaps less suited for model checking as it operates in a less discrete domain. Since the current solution uses sets of waypoints, it seems more natural to interpret it as a finite state system.

6.3.5 Further work for the development of RoboTool

The RoboTool software provides an intuitive way to model complex systems through the GUI. This philosophy of modeling through a graphical interface could be further developed by making even more functionality available in the graphical view. A few features are suggested to improve the user experience. The first is to add the capability of writing statements inside of states from the graphical view. Currently, the only way to define an entry action is to open the `.rct` file in the text editor and write the command there. It would be nice to have all capabilities available without leaving the graphical view. Another is that you can have diverging versions if both the text file and the graphical view are open, and both are edited. A suggested fix is that a warning could occur if the user tries to make changes to the text file when there are unsaved changes in the graphical view. The same warning could appear if there are changes in the text file that have not been updated in the model. Finally, it could be interesting to perform further experiments to pinpoint the reason for untimed assertions taking so long to compile for this model. This could very well be an issue with FDR4, which can also be investigated further.

Chapter 7

Conclusion

To conclude this thesis, the goals and objectives that were defined in the introduction will be restated and evaluated in retrospect of what has been achieved.

The first goal was: "Familiarization with the RoboStar workflow as well as previous case studies where RoboChart has been used". This goal was met by consulting the available case studies on the RoboStar website. The models used in these case studies were used as templates for the model presented in this paper. Since the source material is limited, these were the main sources of inspiration for ways to represent different concepts in RoboChart.

The second goal was: "Analysing the IntelliWelder solution to identify a suitable section for modeling as well as appropriate abstractions". This goal was met in the problem analysis chapter, where the different components of the IntelliWelder system were analyzed so that well-informed modeling decisions could be made. The abstractions were defined in the method chapter based on the insight gained from the problem analysis. The section of the system that was later modeled was also defined in the problem analysis chapter.

The third goal was: "Modeling a well defined section of the IntelliWelder in an adequate and representative manner in RoboChart". The model was defined in the method chapter based on the abstractions that were previously defined and the analysis of the system. This model was made to capture as many meaningful features of the real system as possible within the framework of RoboChart

modeling. The goal was to do this while maintaining the synchronization properties defined for the modeled section.

The fourth goal was: "Defining requirements for the model in a format that allows refinement checking in FDR4". The requirements were defined as assertions in the method chapter. These were custom assertions defined to capture key properties of the system such as movement operations being called for each movement request and the absence of undesired termination of the state machines in the model. The assertions were automatically compiled to a `CSP` file ready for model checking in FDR4.

The fifth goal was: "Evaluating the results of the model checking, as well as the model itself and the assertions defined". This goal was met by checking the assertions in FDR4. The results of the checks can be seen together with the statistics of the model checking in the results chapter. Some of the interesting traces that were obtained through FDR4 were also presented here. In the discussion chapter, these results were evaluated based on whether or not they lined up with the expected outcomes of the model checking. Generally, the assertion results were as expected, which means certain properties of the model have been formally verified. It was also shown that the model is input-dependent and that synchronization properties were not maintained if erroneous input was received. One surprising result was the time it took to compile some of the untimed assertions. The model and assertions were also evaluated in the discussion chapter.

The sixth and final goal was: "Stating recommendations and further work based on experiences and findings from the thesis". These recommendations were given in Chapter 7 along with suggestions for further work. The suggestions included the natural next steps for the model. Those included potential changes to the existing model as well as a plan for progressing the model in the RoboStar workflow. Further work was also suggested for the IntelliWelder system and for the further improvement of RoboChart.

Hopefully, the work done in this thesis has provided valuable insight for both the developers of the IntelliWelder system and the team behind RoboStar technology.

Bibliography

[1] C. Freschi, V. Ferrari, F. Melfi, M. Ferrari, F. Mosca, and A. Cuschieri, "Technical review of the da vinci surgical telemanipulator," *The International Journal of Medical Robotics and Computer Assisted Surgery*, vol. 9, no. 4, pp. 396–406, 2013.

[2] I. W. Muzan, T. Faisal, H. Al-Assadi, and M. Iwan, "Implementation of industrial robot for painting applications," *Procedia engineering*, vol. 41, pp. 1329–1335, 2012.

[3] J. N. Pires, A. Loureiro, and G. Bölmsjo, *Welding robots: technology, system issues and application*. Springer Science & Business Media, 2006.

[4] "Universal robots - ur10e website," Universal Robots. (), [Online]. Available: `https://www.universal-robots.com/products/ur10-robot/` (visited on 05/08/2024).

[5] "Carpano five - website," Carpano. (), [Online]. Available: `https://www.carpano.it/five-500-kg-welding-turntable/` (visited on 05/08/2024).

[6] A. Afzal, D. S. Katz, C. Le Goues, and C. S. Timperley, "Simulation for robotics test automation: Developer perspectives," in *2021 14th IEEE conference on software testing, verification and validation (ICST)*, IEEE, 2021, pp. 263–274.

[7] H. Choi, C. Crump, C. Duriez, *et al.*, "On the use of simulation in robotics: Opportunities, challenges, and suggestions for moving forward," *Proceedings of the National Academy of Sciences*, vol. 118, no. 1, e1907856118, 2021.

[8] P. Koopman and M. Wagner, "Challenges in autonomous vehicle testing and validation," *SAE International Journal of Transportation Safety*, vol. 4, no. 1, pp. 15–24, 2016.

[9] E. M. Clarke, "Model checking," in *Foundations of Software Technology and Theoretical Computer Science*, S. Ramesh and G. Sivakumar, Eds., red. by

G. Goos, J. Hartmanis, and J. Van Leeuwen, vol. 1346, Series Title: Lecture Notes in Computer Science, Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 54–56, ISBN: 978-3-540-69659-9. DOI: 10.1007/BFb0058022. [Online]. Available: http://link.springer.com/10.1007/BFb0058022 (visited on 04/10/2024).

[10] M. Ozkan, Z. Demirci, Ö. Aslan, and A. Yazıcı, "Safety verification of multiple industrial robot manipulators with path conflicts using model checking," *Machines*, vol. 11, no. 2, p. 282, Feb. 13, 2023, ISSN: 2075-1702. DOI: 10.3390/machines11020282. [Online]. Available: https://www.mdpi.com/2075-1702/11/2/282 (visited on 04/17/2024).

[11] Y. Murray, M. Sirevåg, P. Ribeiro, D. A. Anisi, and M. Mossige, "Safety assurance of an industrial robotic control system using hardware/software co-verification," *Science of Computer Programming*, vol. 216, Apr. 2022, ISSN: 01676423. DOI: 10.1016/j.scico.2021.102766. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0167642321001593.

[12] B. Rumbaugh Jacobson, *UML Reference Manual*. Addison-Wesley, 1999.

[13] A. Miyazawa, P. Ribeiro, W. Li, A. Cavalcanti, J. Timmis, and J. Woodcock, "Robochart: A state-machine notation for modelling and verification of mobile and autonomous robots," *University of York, Department of Computer Science, York, UK, Tech. Rep*, 2016.

[14] "Eclipse website," Eclipse Foundation. (), [Online]. Available: http://www.eclipse.org/ (visited on 05/10/2024).

[15] A. Miyazawa, P. Ribeiro, W. Li, A. Cavalcanti, J. Timmis, and J. Woodcock, "RoboChart: Modelling and verification of the functional behaviour of robotic applications," *Software & Systems Modeling*, vol. 18, no. 5, pp. 3097–3149, Oct. 2019, ISSN: 1619-1366, 1619-1374. DOI: 10.1007/s10270-018-00710-z. [Online]. Available: http://link.springer.com/10.1007/s10270-018-00710-z (visited on 05/06/2024).

[16] C. A. R. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, Aug. 1978, ISSN: 0001-0782, 1557-7317. DOI: 10.1145/359576.359585. [Online]. Available: https://dl.acm.org/doi/10.1145/359576.359585 (visited on 03/18/2024).

[17] T. Gibson-Robinson, P. Armstrong, A. Boulgakov, and A. W. Roscoe, "FDR3 — a modern refinement checker for CSP," in *Tools and Algorithms for the Construction and Analysis of Systems*, E. Ábrahám and K. Havelund, Eds.,

Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 187–201, ISBN: 978-3-642-54862-8.

[18] K. J. Waldron and J. Schmiedeler, "Kinematics," in *Springer Handbook of Robotics*, B. Siciliano and O. Khatib, Eds. Cham: Springer International Publishing, 2016, pp. 11–36, ISBN: 978-3-319-32552-1. DOI: 10.1007/978-3-319-32552-1_2. [Online]. Available: https://doi.org/10.1007/978-3-319-32552-1_2.

[19] "Ur-script manual for e-series," Universal Robots. (), [Online]. Available: https://www.universal-robots.com/download/manuals-e-seriesur20ur30/script/script-manual-e-series-sw-511/ (visited on 05/07/2024).

[20] K. Weman, *Welding processes handbook*. Elsevier, 2011.

[21] A. M. Turing *et al.*, "On computable numbers, with an application to the entscheidungsproblem," *J. of Math*, vol. 58, no. 345-363, p. 5, 1936.

[22] J. E. Savage, *Models of computation*. Addison-Wesley Reading, 1998, vol. 136.

[23] C. Baier and J.-P. Katoen, *Principles of model checking*. MIT press, 2008, pp. 19–88.

[24] D. Kundu, D. Samanta, and R. Mall, "Automatic code generation from unified modelling language sequence diagrams," *IET Software*, vol. 7, no. 1, pp. 12–28, Feb. 2013, ISSN: 1751-8806, 1751-8814. DOI: 10.1049/iet-sen.2011.0080. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1049/iet-sen.2011.0080 (visited on 05/06/2024).

[25] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Communications of the ACM*, vol. 18, no. 8, pp. 453–457, 1975.

[26] G. M. Reed and A. W. Roscoe, "A timed model for communicating sequential processes," in *International Colloquium on Automata, Languages, and Programming*, Springer, 1986, pp. 314–323.

[27] J. Baxter, P. Ribeiro, and A. Cavalcanti, "Sound reasoning in tock-csp," *Acta Informatica*, vol. 59, no. 1, pp. 125–162, 2022.

[28] "Real-time data exchange guide," Universal Robots. (), [Online]. Available: https://www.universal-robots.com/articles/ur/interface-communication/real-time-data-exchange-rtde-guide/ (visited on 05/08/2024).

[29] J. E. Colgate, W. Wannasuphoprasit, and M. A. Peshkin, "Cobots: Robots for collaboration with human operators," in *ASME international mechanical*

engineering congress and exposition, American Society of Mechanical Engineers, vol. 15281, 1996, pp. 433–439.

[30] A. Miyazawa and A. L. C. Cavalcanti. "Autonomous chemical detector - robochart case study," RoboStar. (), [Online]. Available: https://robostar.cs.york.ac.uk/case_studies/autonomous-chemical-detector/autonomous-chemical-detector.html (visited on 05/12/2024).

[31] A. Miyazawa, P. Ribeiro, and A. L. C. Cavalcanti. "Alpha algorithm," RoboStar. (), [Online]. Available: https://robostar.cs.york.ac.uk/case_studies/alpha-algorithm/alpha-algorithm.html (visited on 05/12/2024).

[32] K. Ye and J. Woodcock. "Uvc-treatment in agriculture," RoboStar. (), [Online]. Available: https://robostar.cs.york.ac.uk/prob_case_studies/uvc/index.html (visited on 05/12/2024).

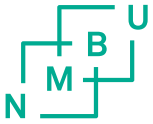
[33] A. Cavalcanti, W. Barnett, J. Baxter, *et al.*, "RoboStar technology: A roboticist's toolbox for combined proof, simulation, and testing," in *Software Engineering for Robotics*, A. Cavalcanti, B. Dongol, R. Hierons, J. Timmis, and J. Woodcock, Eds., Cham: Springer International Publishing, 2021, pp. 249–293, ISBN: 978-3-030-66493-0. DOI: 10.1007/978-3-030-66494-7_9. [Online]. Available: https://link.springer.com/10.1007/978-3-030-66494-7_9 (visited on 02/20/2024).

[34] A. Roscoe, "The theory and practice of concurrency," 1998.

[35] R. M. Hierons, M. Gazda, P. Gómez-Abajo, R. Lefticaru, and M. G. Merayo, "Mutation testing for robochart," *Software Engineering for Robotics*, pp. 345–375, 2021.

[36] M. De Wulf, L. Doyen, and J.-F. Raskin, "Almost asap semantics: From timed models to timed implementations," in *International Workshop on Hybrid Systems: Computation and Control*, Springer, 2004, pp. 296–310.

[37] J. Lawrence, "Practical application of CSP and FDR to software design," in *Communicating Sequential Processes. The First 25 Years: Symposium on the Occasion of 25 Years of CSP, London, UK, July 7-8, 2004. Revised Invited Papers*, A. E. Abdallah, C. B. Jones, and J. W. Sanders, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 151–174, ISBN: 978-3-540-32265-8. DOI: 10.1007/11423348_9. [Online]. Available: https://doi.org/10.1007/11423348_9.



Norges miljø- og biovitenskapelige universitet
Noregs miljø- og biovitskapelege universitet
Norwegian University of Life Sciences

Postboks 5003
NO-1432 Ås
Norway