Model Checking and Verification of Synchronisation Properties of Cobot Welding

Yvonne Murray
Pioneer Robotics AS
Dept. of Mechatronics, University of Agder
Norway
ym@pioneer-robotics.no

Henrik Nordlie
Robotics Group, Faculty of Science & Technology
Norwegian University of Life Sciences (NMBU)
Norway

David A. Anisi
Dept. of Mechatronics, University of Agder
Robotics Group, Faculty of Science & Technology
Norwegian University of Life Sciences (NMBU)
Norway

Pedro Ribeiro
Dept. of Computer Science
University of York
UK

Ana Cavalcanti
Dept. of Computer Science
University of York
UK

This paper describes use of model checking to verify synchronisation properties of an industrial welding system consisting of a cobot arm and an external turntable. The robots must move synchronously, but sometimes get out of synchronisation, giving rise to unsatisfactory weld qualities in problem areas, such as around corners. These mistakes are costly, since time is lost both in the robotic welding and in manual repairs needed to improve the weld. Verification of the synchronisation properties has shown that they are fulfilled as long as assumptions of correctness made about parts outside the scope of the model hold, indicating limitations in the hardware. These results have indicated the source of the problem, and motivated a re-calibration of the real-life system. This has drastically improved the welding results, and is a demonstration of how formal methods can be useful in an industrial setting.

1 Introduction

Robotic welding is commonly used in industrial workshops to increase efficiency and repeatability, and reduce dangerous and ergonomically straining work for human welders [12]. To address the needs of small and medium-sized enterprises (SMEs), which produce a large variety of products in small quantities, the welding system must be easy and fast to re-program, and highly flexible. To this end, Pioneer Robotics have developed the IntelliWelder M06 [17], a flexible and light-weight welding system consisting of a Universal Robots (UR) UR10e cobot [8] equipped with a welding torch and a Carpano FIVE MOT turntable serving as an external axis (EXAX). Fig. 1 shows the components of the IntelliWelder.

The main challenge in the operation of the IntelliWelder M06 has been to get a high quality, continuous weld in difficult areas, such as around corners. To get the best results, it is important that the welding robot and the turntable move continuously in a *synchronous* fashion. By using synchronous welding, it is possible to achieve a continuous weld of high quality, ensuring the weld is not jagged and interrupted. When the synchronisation does not work properly, the welding gun does not move forward in an even motion, and can move too fast or too slow. Moving too fast does not give the metal and filler enough time to heat up and weld together, while moving too slow results in build-up of filler material. Both of these problems can be seen in the weld depicted in Fig. 2.

This experience report describes how we have addressed some challenges faced when the UR robot and the EXAX move synchronously while welding. Relevant parts of the system have been modelled in

Matt Luckcuck and Mengwei Xu (Eds.):
Sixth International Workshop on Formal Methods
for Autonomous Systems (FMAS 2024)
EPTCS 411, 2024, pp. 91–108, doi:10.4204/EPTCS.411.6

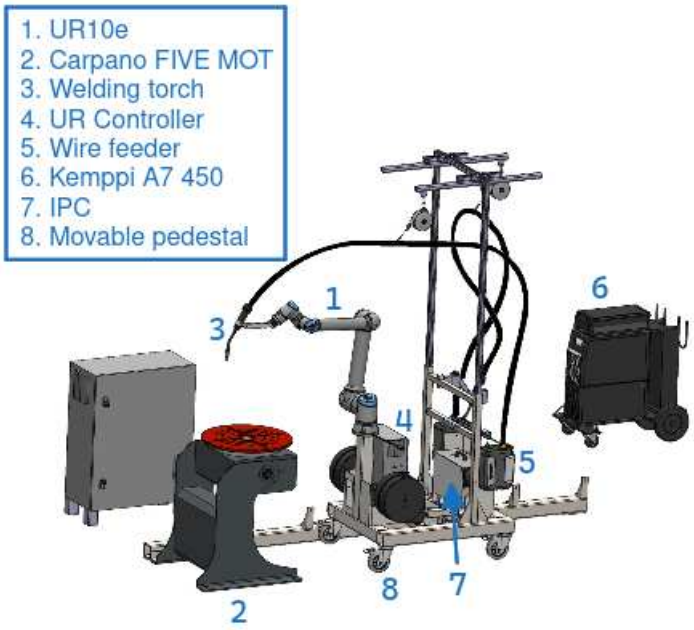

Figure 1: The IntelliWelder system with the different components marked by number [16].

Figure 2: Typical welding issue where there is buildup of filler material (A) and coverage is not sufficient (B), creating an irregular and weakened weld.

RoboChart [14], a domain-specific language for modelling and verification of robotic systems. Using our RoboChart model, key synchronisation properties have been verified using the refinement model checker FDR [25]. Details omitted here are in [16]; the work demonstrates how formal verification can be used in industry, and how the results can be used to localise the error source, leading to system improvements.

Previous research on multi-robot welding include [20], which focuses on nominal trajectory planning and self-coordination, and [27], which studies trajectory smoothing in a dual-robot collaborative welding system. Neither of these lines of work use formal methods or model checking. Closer to our research, the work in [19] combines graphic and formal methods to analyse collaborative behaviour such as deadlock and equivalence properties. None of these works, however, consider the issue of correct time synchronisation during multi-robot execution like we do in our case study.

The rest of this paper is organised as follows. In Section 2, we motivate our use of formal methods and model checking. In Section 3, we detail the system architecture and requirements, before the model is presented in Section 4. In Section 5, we present the verification results and their practical implications. Finally, in Section 6, we conclude, describe ongoing work, and suggest further work.

2 Formal Verification and Model Checking

To find and mitigate faults and undesired behaviour in robotic systems, they are traditionally subject to testing, including simulation before deployment. For real-world, complex robotic systems, however, it is impossible to test every possible scenario and input sequence. Moreover, even if a fault is discovered, error source localisation remains a challenge. In this setting, formal verification methods are a useful supplement. Model checking [1, 7] is a formal method to verify that given properties are fulfilled, regardless of inputs. If a property does not hold, model checking provides a counterexample that can pinpoint the cause of error. Adopting such methods is valuable in the design of real, industrial systems.

RoboTool [14] is a suite of plugins for the Eclipse IDE supporting use of the RoboStar framework [5]. Our previous work on verification of an industrial control system [15] using RoboStar has shown its proficiency and strengths. In RoboStar, a key artefact is a RoboChart [13] model that reflects the real system design. Once this model is created, assertions for the selected properties can be written and verified using the CSP process algebra and its model checker, FDR [10].

As our use case considers an already existing IntelliWelder system, we need to alter the idealised workflow of RoboStar [5] by effectively "reverse engineering" the RoboChart model from the existing system.

3 IntelliWelder and Synchronous Welding

In this section, we describe our case study: its architecture (Section 3.1), software (Section 3.2), and requirements (Section 3.3), identifying the problem we are addressing with model checking.

3.1 System Architecture

An illustration of the IntelliWelder's architecture can be seen in Fig. 3. To realise synchronous welding, Delfoi offline robot programming software from Visual Components [26] is used for creating waypoints and welds in a 3D layout consisting of the UR10e [22], the Carpano FIVE turntable, and the workpiece to be welded. Welds are created simply by selecting an edge on the workpiece CAD model. Delfoi then creates waypoints for both the UR robot and the Carpano turntable, so that each waypoint for the robot has a corresponding waypoint for the turntable, creating nominally synchronised movements.

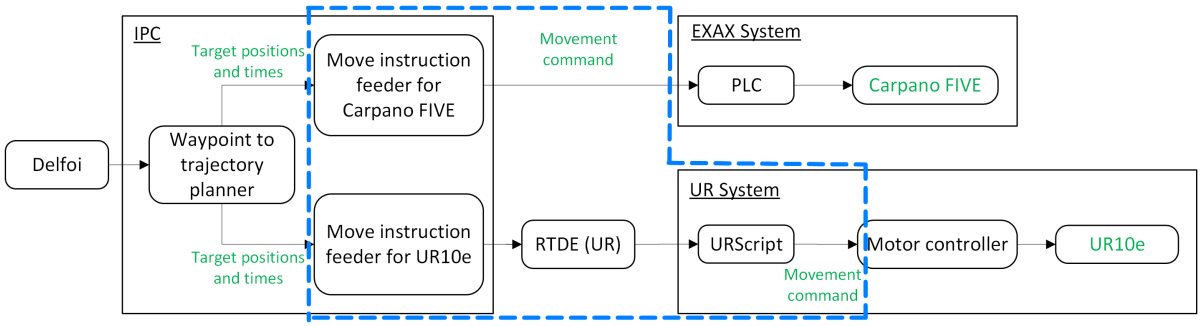

Figure 3: System architecture of the IntelliWelder. The blue dotted line indicates the scope of the RoboChart model: part of the Industrial PC (IPC), the Real-Time Data Exchange (RTDE) for the UR robot, and the URScript.

The waypoint paths generated in Delfoi are transferred to the Industrial PC (IPC). The waypoints are then processed to convert them into trajectories based on the desired forward welding speed and other

welding parameters. The waypoints for the external axis remain unchanged and are converted into a trajectory, while the waypoints for the UR robot are sampled at a higher resolution before being turned into its trajectory. Consequently, the UR robot has more waypoints to process than the Carpano FIVE. Once the trajectories are generated, the IPC sends them as movement requests for the system to execute.

The Carpano FIVE is controlled by a Programmable Logic Controller (PLC) that receives movement commands from the IPC. These commands can be based on position and velocity, or just velocity. The PLC then regulates movement using a PID. The Real-Time Data Exchange (RTDE) synchronises external applications with the UR robot [23]. It relays messages from the IPC to the UR robot via a TCP/IP connection. The UR robot controller executes URScript applications and manages movement via a PID.

3.2 Existing URScript code

In the current implementation, every time a movement request for the UR robot arrives, the URScript code runs on the UR controller. Required variables are read from the registers, updated by the RTDE, and the code checks if the target time for the next waypoint has already passed (that is, the robot is behind schedule). If so, the code logs a warning and continues to the next target.

Next, the URScript decides which movement type is preferable to reach the next waypoint, based on variables like blend radius, offset, joint velocities, and whether the next movement involves a sharp turn. The script selects between MoveJ, MoveL and MoveP, which are standard UR robot movements described in the user manual [24]. However, if none of the standard movement types are suitable, a custom-made function called MoveL_with_t is used for the movement. The custom URScript function MoveL_with_t uses the standard MoveL command with the next target pose and next target time as arguments. In that way, the UR robot can calculate the necessary velocity to reach the next target within the target time, using maximum acceleration. The reason for this being a fallback solution, only used when the standard movements are not feasible, is that it does not include a blend radius.

With a blend radius, we ensure that when the UR robot is within a given distance of the waypoint, it starts moving towards the next waypoint instead of completely finishing the move to the current waypoint. So instead of coming to a brief stop at the waypoint, it keeps moving towards the next, giving a smoother transition. Lack of a blend radius results in a jagged movement that is not ideal for welding.

The next section describes the requirements that the design just presented is expected to satisfy.

3.3 Requirements

We present here both system requirements (in Section 3.3.1), and requirements specifically for the components that we model as described in Figure 3 (in Section 3.3.2).

3.3.1 System-Wide Requirements

There are several requirements for the IntelliWelder system as a whole, discussed in detail in [16]. The most important of these requirements are the following two:

1. The welding torch must always stay in an area defined by a maximum deviation from the weld frame. This includes both position and orientation.

2. The welding torch must always move forward in the weld frame with a speed that is within a given maximum deviation of the desired forward weld speed.

These requirements need to be refined into specific requirements for Delfoi, the IPC planner, the calculation of arguments for the robot commands, and the execution of movements. Additionally, they expand to include requirements related to information communication and code execution time.

3.3.2 Model Requirements

With the system wide requirements in mind, the following requirements for the modelled component (see Figure 3) can be obtained, as described in detail in [16]:

R1 The component should detect events that imply that the system is out of sync.

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

The requirements R1 and R2 above are the properties we verify using model checking. If some of the assertions fail, it can help to pinpoint existing mistakes in the software. If all assertions pass, it indicates that some of the assumptions made on the component's context and the hardware are invalid.

To check the hardware, two different cases are evaluated: one where it is impossible for the robot to receive a waypoint that is already in the past (nominal case), and one where that is possible (realistic case). If the model checking results vary between the two, for example, if the assertions pass in the nominal case (which assumes the hardware is able to keep to the planned trajectory) but fail in the more realistic case, it is an indication that assumptions about velocities, accelerations, and perfect move execution, made on the real-life system, are inaccurate. We recall that the system does present a problem. So, if the problem is not present when the hardware executes the planned trajectory, then we can conclude that our assumptions about the hardware are not satisfied, and so, inaccurate.

In the next section, we present the RoboChart model we use to carry out our verification.

4 Modelling in RoboChart

Our RoboChart model reflects the system architecture already described, and the existing code and specifications. Any possible communication delays are assumed to be handled separately and are hence negligible for our purposes here. The components modelled receive movement requests as inputs. Thus, trajectory planning is outside of the model's scope and the feasibility of planned trajectories is assumed.

As previously noted, the number of waypoints differs for the UR robot and the Carpano FIVE, so both are expected to receive and execute commands concurrently and independently. In terms of control flow, the model's scope extends until the point where these movement commands are initiated for the Carpano FIVE and the UR robot. From that point, they handle the execution of movements. We expect and assume that the actual execution of movement commands by the UR robot and Carpano FIVE is correct and, therefore, that is also beyond the model's scope.

The definition of the model's scope reflects the fact that our goal is to check that our use of the Carpano FIVE and UR robot commands is appropriate. Therefore, in the RoboChart model, these commands are captured as services of the robotic platform, which we do not further specify.

The component modelled is responsible for selecting the most appropriate movement type for each request. It also detects if the system is out of sync, meaning the target time for a movement request has already passed, resulting in a negative time budget. In the model, this is indicated by the occurrence of an out-of-sync event, and is considered a critical failure.

Next, we justify the abstractions and simplifications made in the RoboChart model (Section 4.1), and then present the RoboChart model itself (Section 4.2).

4.1 Abstractions and Simplifications

It is well-known that model checking eventually encounters state-explosion problems. To keep the complexity and verification time at bay, the following abstractions and simplifications have been made.

Reduction of number of joints: The Carpano FIVE turntable has two joints, one for tilting and one for turning. The IntelliWelder, however, only uses the turning axis during the synchronous welding. Thus, this is the only axis that is considered in the model. The UR robot has six rotational joints, but it is modelled with only two. Although this selection can be made arbitrarily, selecting one from the first three joints (to represent position) and the other from the last three joints (to represent orientation) is advocated. By modelling two axes, the model still captures potential problems related to multiple joint values. Extending the model to include six axes affects the computational complexity of the model, as it would lead to additional parameters of operations (explained in the next section). These operations, however, are not further specified as they represent services that are out of the scope of our verification. So, additional parameters are not relevant for the verification, but just the fact that they are available.

Limited value ranges: To decrease the computation time and complexity, the value ranges of the variables of type real, recording distance, are limited. To cover both negative, positive, and zero-values, the integer range [-1..1] has been chosen. Similarly, the int variables recording discrete time are limited to the two ranges [0..2] and [-1..1]. With the first range, with only positive values, the assumption is that the UR robot and EXAX are never so late that the next waypoint is already in the past. When a negative value -1 is included, we can check whether the goal time for the next waypoint has passed. Lastly, the variables recording the current waypoint for the UR robot and the turntable, of datatype nat, are limited to the ranges [0..3] and [0..1], respectively. So, the maximum number of waypoints for the UR robot are 4 and for the turntable 2, capturing that the number of waypoints can vary in the real system.

Omitted variables: Some variables defined in the URScript are not used in the model to minimize the number of variables. For instance, current and target positions are not included if they are only used to calculate a distance. Instead, the distance is input directly. Adopting a similar approach, other variables are omitted or given a boolean rather than a numerical type to reduce the state space.

4.2 RoboChart Model

In writing a RoboChart model, a key decision is the definition of events and operations that capture services of the robotic platform. The previous section describes our assumptions, some of which are reflected in these definitions. These services are not further specified and establish the interface of the model. Properties are described in terms of interaction with the modelled component via these services.

Fig. 4 shows the robotic platform of our model (on the right), with three input events (start_system, next_UR_move and next_EXAX_move) and the five operations that can be called (four operations in ur_ops and one in exax_ops). The events, declared in the events interface, are used to initiate the system and send new move requests as previous moves are completed. The operations are defined in two separate interfaces: ur_ops and exax_ops, corresponding to the move commands that are executed by the UR robot and the Carpano FIVE. Although the model declares one robotic platform, it captures services of both the robot and the turntable used by the software.

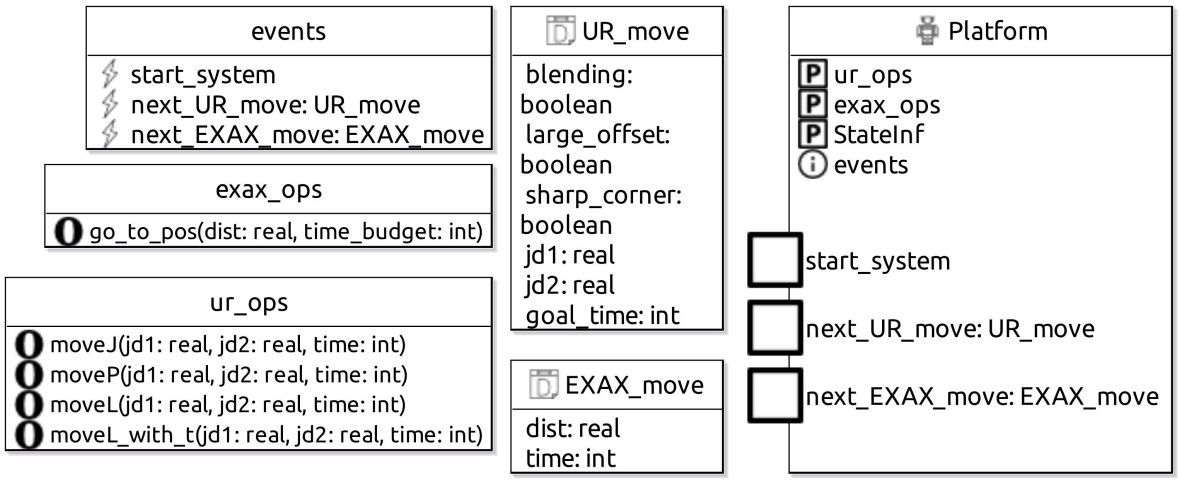

Figure 4: Robotic platform with its defined events, provided operations and custom record types to represent move commands.

Module A RoboChart model is defined by a module block, including the robotic platform and, in our case, one controller block, as shown in Fig. 5. Our module is called main, and the behaviour of our controller is defined by five parallel state machines, acting over the events and operations of the robotic platform as declared in three required interfaces and reflected in connections between the robotic platform and the controller (arrows between the blocks annotated with async).

A RoboChart controller defines how the events of the robotic platform connect to its state machines. In our example, start_system is used by the machine called System. The other two events are used by the machine state_check. A controller also defines how its state machines are connected to each other, via their events, to exchange information and synchronise their behaviour. In Fig. 5, the Controller block includes five blocks, each a reference to one of its state machines, as indicated by the keyword ref. In what follows, we present the definition of these machines.

The System **state machine** Its definition is shown in Fig. 6. In each state of System, a shared variable sys_state is updated to record the current state of the system. This variable is used in the state_check machine presented later to decide whether or not a movement request should be forwarded to the EXAX or to the UR state machine, that is to the turntable or to the UR robot.

The initial junction, a black circle with an i, indicates wait_for_start as the initial state of System, where it waits for the event start_system. When start_system happens, System moves to the working state, where it stays until either the UR robot or the EXAX finishes all of their waypoints, as indicated

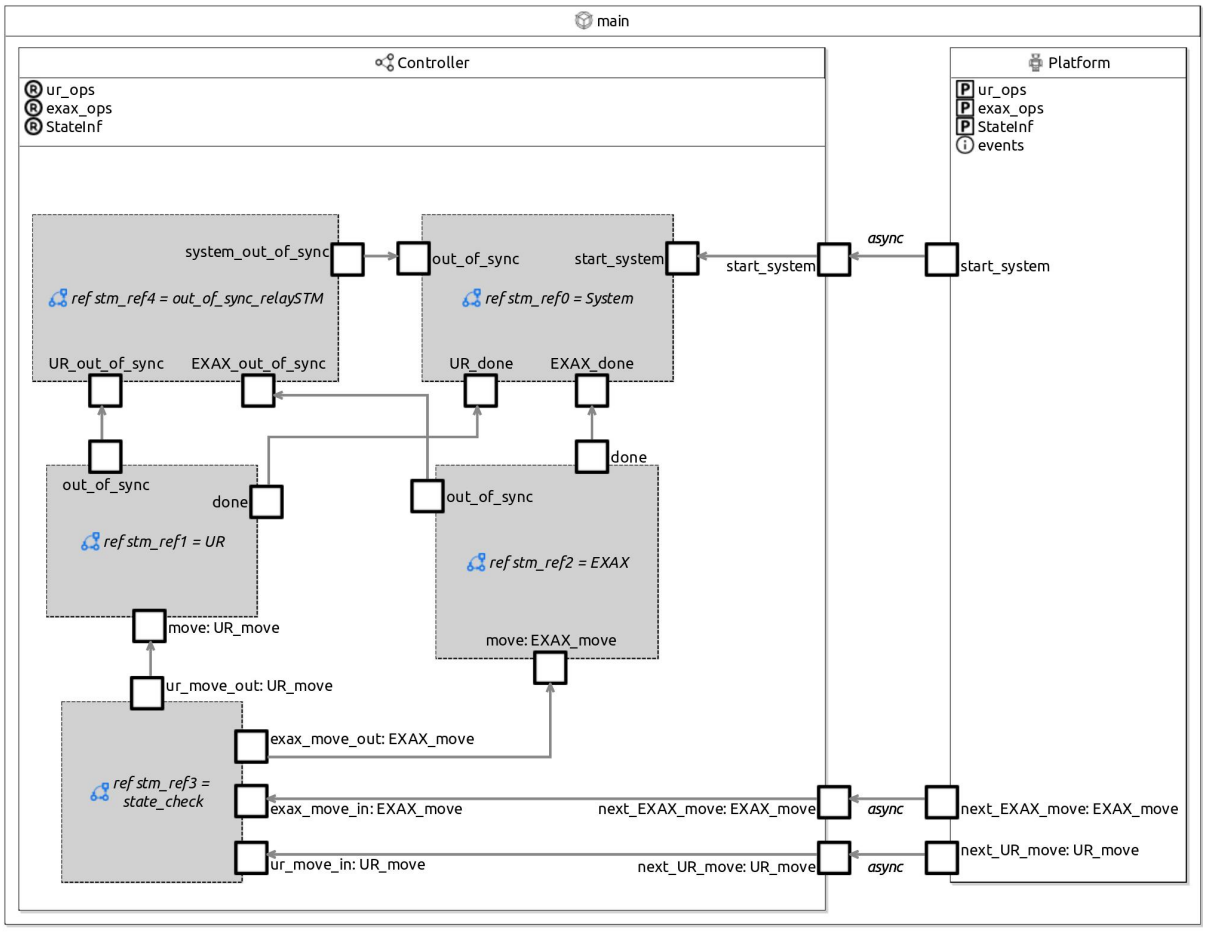

Figure 5: Main module including the Controller with all state machines and the robotic platform.

by events UR_done and EXAX_done, or an out_of_sync event occurs. An out_of_sync event, from any of the states working, UR_finished or EXAX_finished, results in a transition to the final state: white circle with an F. This means that the System state machine cannot progress further.

If both UR_done and EXAX_done occur, regardless of in which order, System goes through either the state UR_finished (if the UR robot finishes first) or EXAX_finished (if the EXAX finishes first), before going back to wait_for_start. This is the end of a welding operation.

The EXAX state machine Its definition, shown in Fig. 7, captures the behaviour of the turntable. EXAX starts in the wait_for_move state, waiting for a move command. The variable curr_waypoint is initialised to 0, and with the constant n_waypoints defined as 1, as in Fig. 7, the turntable goes through two waypoints. When EXAX receives a move event, it stores the requested distance and time to move in a variable exax_move. In the junction (dark circle), it is checked if the time of the movement request (exax_move.time) is strictly negative. If it is, an out_of_sync event is triggered and EXAX terminates. Otherwise, EXAX moves to a state by_position, where the operation go_to_pos is called using as arguments the values in exax_move. If curr_waypoint is greater or equal to n_waypoints, curr_waypoint is reset and the done event is triggered. This is then relayed, by the controller, to the

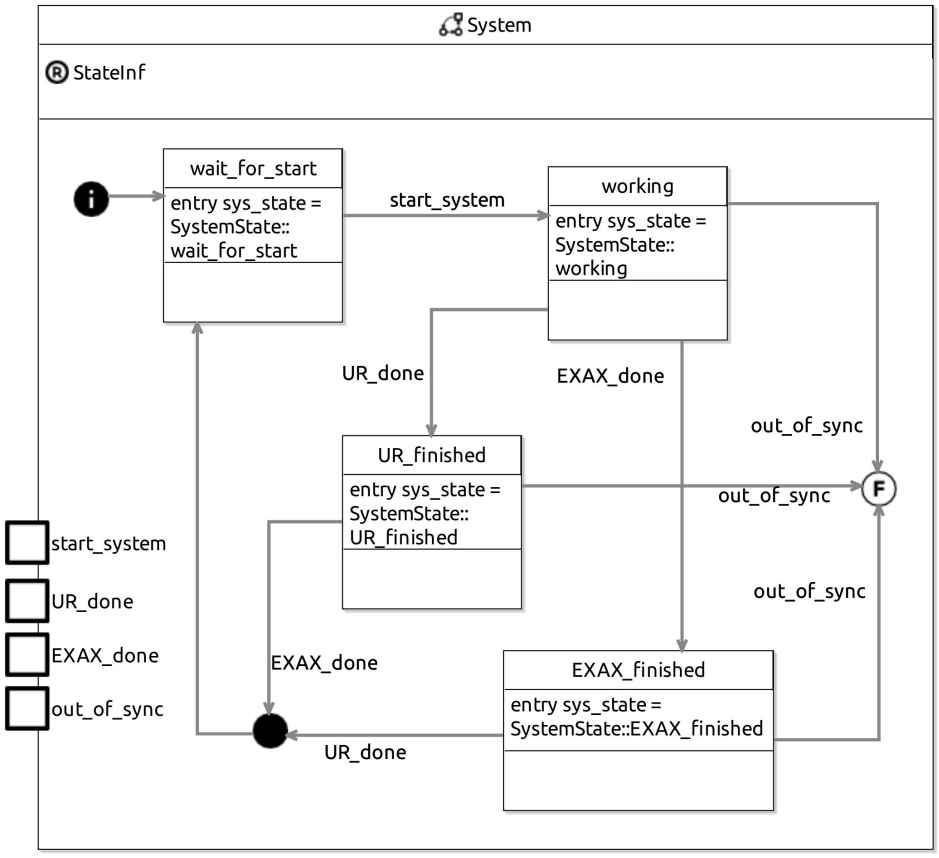

Figure 6: System state machine.

System state machine via its event EXAX_done. (System then transitions to its state EXAX_finished).

The UR state machine It is defined in Fig. 8 to model the behaviour of the UR robot, and is similar to EXAX (Fig. 7). The same method of counting and incrementing waypoints is used, and the done and out_of_sync events are used in the same way. The UR robot, however, chooses the most suitable movement type, so choose_cmd is more complex than the by_position state of EXAX.

Upon entering the choose_cmd state, the boolean variable choosing is set to true. This ensures that a move command must be chosen before leaving the state, since the only transition out of the state choose_cmd has a guard that requires the value of choosing to be false.

Which move command is chosen depends on whether or not the move request includes blending, a large offset from the ideal path (set to 0.8mm in our use case), or a sharp corner. The first junction checks whether the move request includes a blend radius or not, that is, whether ur_move.blending is true or false, where the variable ur_move records the data associated with the move request as defined in the transition out of wait_for_move. If it does include a blend radius, the next junction checks whether the move request includes a large offset (ur_move.large_offset).

If the offset is smaller than the threshold, moveJ is chosen: UR transitions to the moveJ state, where the operation of the same name is called and the variable choosing is set to false in the exit action. With

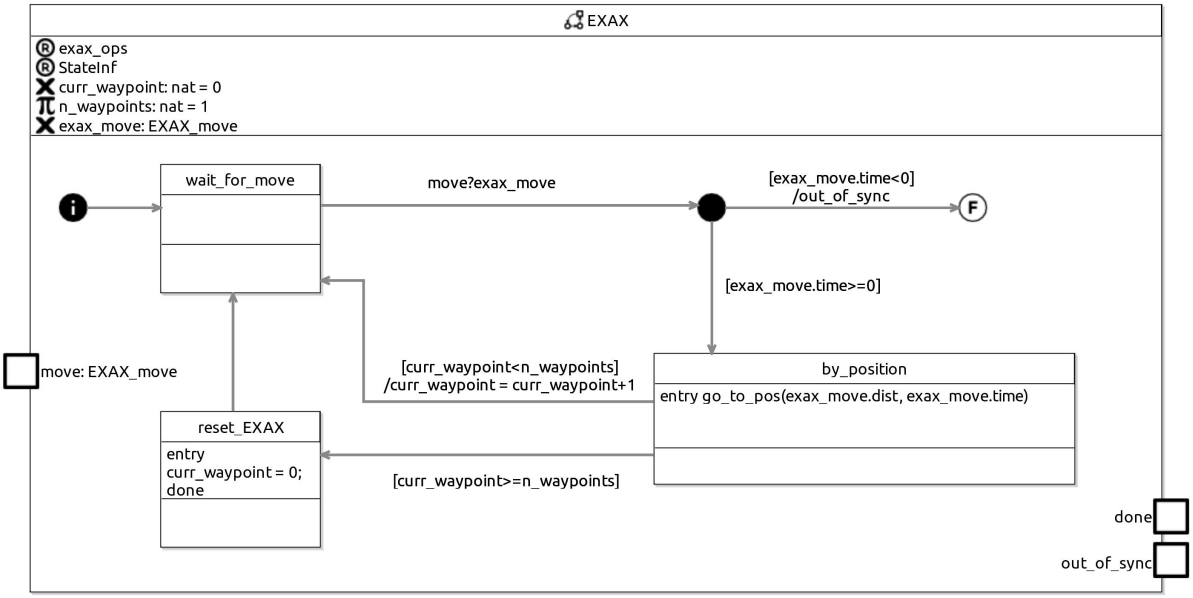

Figure 7: EXAX state machine

that choose_cmd is exited. If the offset is large, the next junction checks whether the move request contains a sharp corner or not (ur_move.sharp_corner). If it does not, moveP is suitable, but if it does, it is necessary to use moveL_with_t. In each case, like for moveJ, the entry action of a state calls the right operation and the exit action updates choosing. If the move command does not include blending, the system enters the big_dist_check state after the very first junction. In the entry action of that state, the check_big_dist function checks if the absolute value of either of the joint distances is larger than a given value (in the example, 1), since the distance determines if moveL is sufficient. If the distance is short, a moveL_with_t command is issued in the state of the same name.

The out_of_sync Relay **state machine** It is simple and omitted here; it relays the out_of_sync event from the EXAX and UR to System. This is necessary just because RoboChart prohibits connecting two different events to the same input of another machine. Full details can be found in [16].

The state_check **state machine** It is defined in Fig. 9 and has a single state checker with two self-transitions. They are triggered by events that accept and record a move command in local variables ur_move or exax_move depending on whether the UR or the EXAX received a move request (whether an input event ur_move_in or exax_move_in happens).

The guards of the transitions ensure that these inputs are accepted only if the system is in a state where the move command should be forwarded to the UR or EXAX machines. There are only two states where they should move: states working or EXAX_finished, for the event ur_move_in, and working or UR_finished, for exax_move_in. In the actions of the transitions, if a move request for the UR robot arrives, it is forwarded to the UR state machine. Similarly, a move request for EXAX is forwarded to the EXAX state machine. With the guards in the transitions, the state_check machine ensures that no move operations can be executed before the system has started, and that once a robot has reached all its waypoints, no further move operations can be executed until the system is reset and restarted.

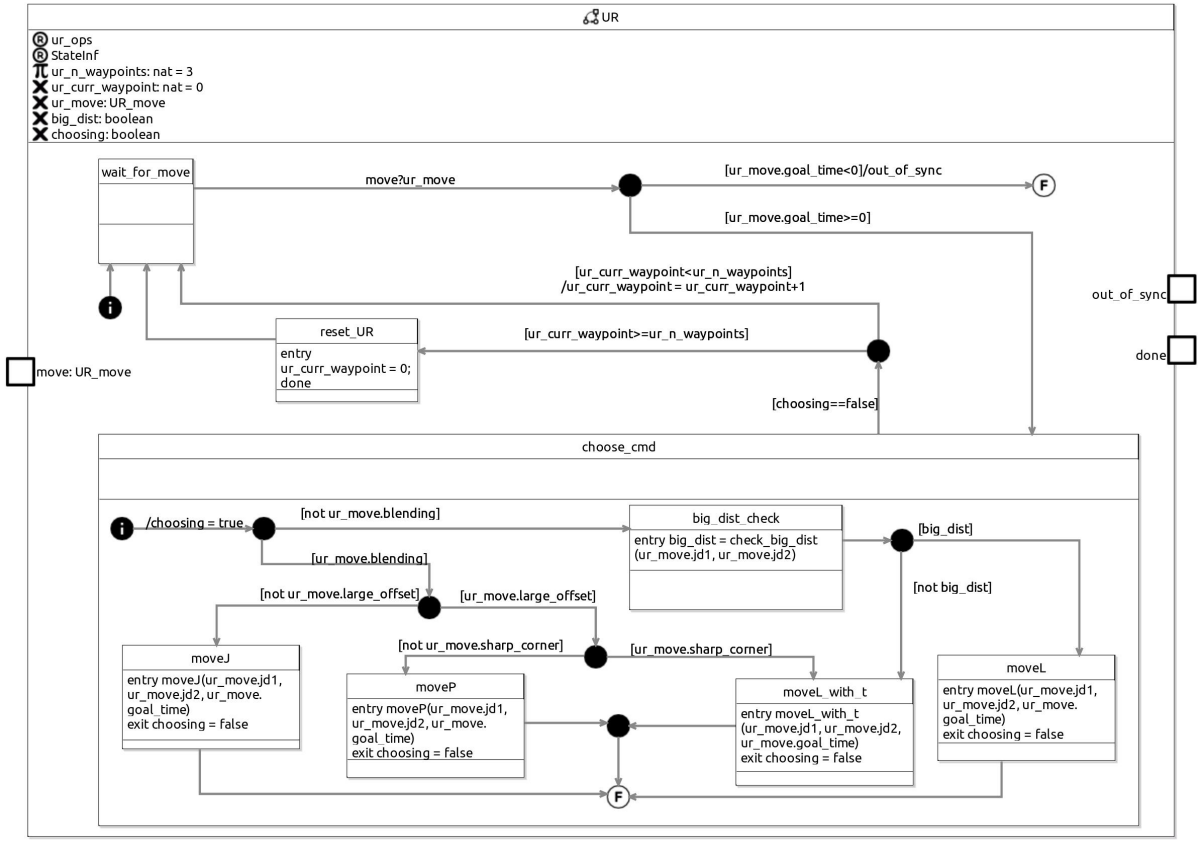

Figure 8: UR state machine.

5 Model Checking

In this section, we describe the model checking and its results: the defined properties and assertions (Section 5.1), the results from FDR (Section 5.2), and their implications for the real-life system (Section 5.3).

5.1 Verification of Selected Properties

Based on the requirements from Section 3.3, assertions can be formulated in natural language, and later defined in *tock*-CSP, which is a dialect of the process algebra CSP where the event *tock* marks the passage of discrete time. To this end, the following properties are to be validated through model checking [16]:

- Every time an EXAX_move or UR_move input event is triggered by the robotic platform, the corresponding movement operation for EXAX or UR, respectively, is called. This is captured in **assertion A1** and **A2** for EXAX , and in **assertion A3** and **A4** for UR.
- The EXAX state machine and the UR state machine do not terminate. This is captured in **assertion A5** and in **assertion A6**, respectively.
- If no out_of_sync event occurs in the System state machine, the state machine does not terminate. This is captured in **assertion A7**.

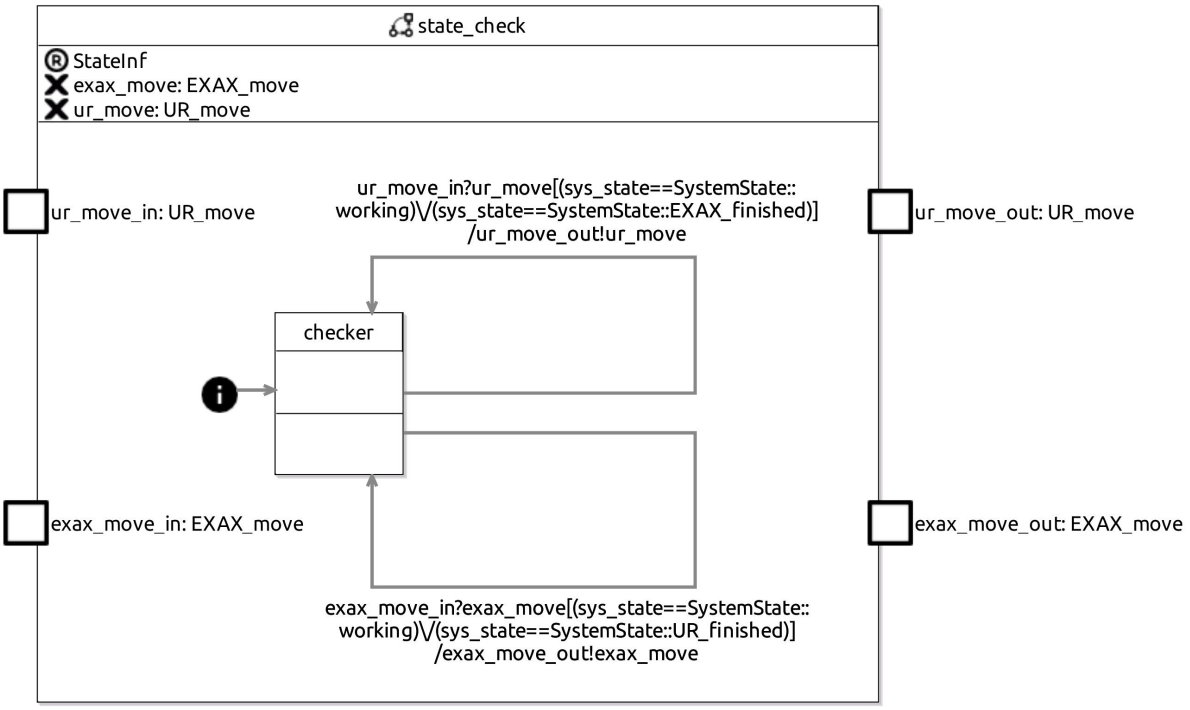

Figure 9: State machine responsible for relaying the move commands of the UR and EXAX only if System is in a state where those state machines should receive commands.

All the assertions described above are detailed next.

Assertion A1 We present in Listing 1 the RoboTool script defining `assertion A1`. With that, RoboTool can use FDR to check whether the assertion holds or not. FDR uses *tock*-CSP processes that define the semantics of the state machines presented in the previous section. These *tock*-CSP processes are automatically calculated by RoboTool. The scripts are written in a mixture of natural language and CSP.

As defined in Listing 1, `assertion A1` requires that `EXAX refines SpecA1 in the traces model`, on line 8. This means that `assertion A1` requires the traces of the process `EXAX` for the machine of the same name to be also traces of the process `SpecA1` defined in lines 1-7.

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

Listing 1: Definition of `SpecA1` and `assertion A1`.

`SpecA1` is defined directly in CSP, as indicated in lines 1 and 7. Moreover, it is defined within a `Timed` section (line 2). So, it is a *tock*-CSP process. It is given by the equation in line 3.

The definition of **SpecA1** uses a **let-within** construct. In the **let** clause, a process **Def** (lines 4-5) is defined. In the **within** clause, it is used to define **SpecA1** using a **timed_priority** function. This is just a technicality of FDR: **timed_priority** enforces the understanding of *tock* as a special event that marks the passage of time. So, the behaviour of **SpecA1** is really that of **Def**.

The behaviour of **Def** is initially that of **CHAOS(Events)**, a process that allows any event to occur. It can, however, be interrupted (operator **[| ... |>]**) by the CSP event **EXAX::move.in**, which represents the input move. Upon interruption, the behaviour of **Def** is give by **ADeadline**. This is a parameterised CSP process defined in the RoboTool *tock*-CSP mechanisation [2] that takes a set of events and a deadline, given as a number of *tock* events, as arguments. It requires that one of the events in the provided set, in this case only **EXAX::go_to_posCall**, the CSP process representing a a call to go_to_pos, occurs within the deadline, which here is set to 0. Thus, the call to the go_to_pos operation is required to happen immediately when the **EXAX::move_in** event occurs.

Assertion A2 For **assertion A1** to be meaningful, it is necessary to ensure that the EXAX state machine is timelock-free, due to the trivial case where the process refuses the event *tock*. This is checked with **assertion A2**. Listing 2 shows the definition of **A2**, where **EXAX2** on line 3 is defined as a version of **EXAX** in whose traces the events **EXAX::go_to_posCall** are ignored (using the hidden operator: ****). This is done because the machine can timelock in the call to that operation, that is, refuse the *tock* event. This is because that call, being in an entry action, is urgent, and deadlines create potential timelocks. In *tock*-CSP, when a deadline is reached, *tock* is refused. For the **EXAX::D__** process (line 3), two arguments (0, 1) are needed due to technicalities of the CSP model of RoboChart. The first argument is an ID-value, and the second is the value of n_waypoints, which is not fixed in the model of a machine.

```
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 2: Definition of **EXAX2** and **assertion A2**.

Assertion A3 The definition of **assertion A3** and **SpecA3** can be seen in Listing 3. This is the UR equivalent to **assertion A1** and **SpecA1**. This assertion ensures that for each move event, one of the four move operation calls must be made before any time is allowed to pass.

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

Listing 3: Definition of **SpecA3** and **assertion A3**

Assertion A4 Also for the UR STM it is important to ensure timelock-freedom, and this is done in `assertion A4`, which is the UR equivalent to `assertion A2` and omitted here.

Assertions A5 and A6 They are defined in Listing 4; they require that the EXAX and UR state machines, respectively, do not terminate. These assertions are expected to pass given that no out_of_sync occurs.

```
1 assertion A5: EXAX does not terminate.
2 assertion A6: UR does not terminate.
```

Listing 4: Definition of `assertion A5` and `assertion A6`.

Assertion A7 Listing 5 shows the definition of `SysTerminates`, as well as the definition of a process `Stop`, and `assertion A7`. The process `SysTerminates` (line 3) is based on another process (`SysConstrained` from line 2) which is a version of the system where out_of_sync events are skipped. So, the `SysTerminates` process on line 3 only takes into account the termination event of the System state machine. This is done by hiding all events except `System::terminate` using the `|\` operator. If System terminates despite the out_of_sync event being ignored, the assertion should fail, and this is captured by comparing `SysTerminates` to the process `Stop` (line 6-8). `Stop` is equivalent 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 does 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 5: Definition of `SysTerminates`, `Stop` and `assertion A7`.

5.2 Results from Checking the Assertions

As previously mentioned, the assertions have been run with two different value ranges for the time variable, core_int. The range [0..2] implies that it is not possible to receive negative time budgets for the movements, meaning that the movements are always performed in accordance with nominal plans. The range [-1..1] implies that negative time budgets can occur, signifying either an infeasible plan from Delfoi or incorrect execution of movements by either the UR robot or the turntable. The assertions have been checked on a computer with an AMD Dual EPYC 7501 (2*32 cores) processor and 2TiB of RAM.

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 1: Results of model-checking all assertions in FDR with core_int = [0..2].

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 2: Results of model-checking all assertions in FDR with core_int = [-1..1].

Nominal case with only positive time budgets, [0..2] As summarized in Table 1, all assertions pass. This outcome is desirable for verifying the synchronisation properties. Since A1 and A3 pass, it can be concluded that a movement operation is always called for each movement request received. Relating back to the requirements in Section 3.3, it indicates that **R2** is satisfied.

Realistic case with possibility of negative time budget, [-1..1] As summarised in Table 2, the only assertion that passes is A7. Fig. 10 shows an example of a trace related to the failed assertion A5. It shows an out_of_sync event, which should lead to termination due to EXAX_move having a negative value for the time variable. The counterexample, given at the bottom in Fig. 10, shows that the system does not terminate. Since out_of_sync events are possible, it shows that **R1** from Section 3.3 is satisfied.

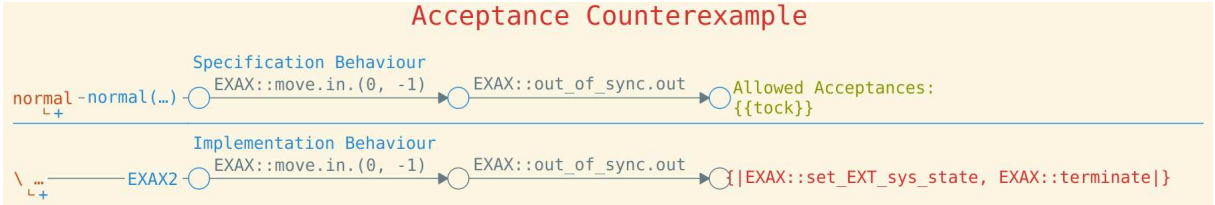

Figure 10: Trace showing a counterexample to assertion A5 - **EXAX does not terminate**.

5.3 Implications and Real-Life System Improvements

The fact that all assertions in the nominal case ($core\_int = [0..2]$) pass, and all assertions apart from one in the realistic case ($core\_int = [-1..1]$) fail, indicates that the robots are unable to follow the nominal plans. This could be, for instance, due to hardware limitations, like insufficient maximum speed or acceleration, or inaccuracy in their trajectory following. To minimise errors, a full re-calibration of the real-life system has been done. Since the programming is offline in Delfoi, it is crucial that the physical system is calibrated precisely so the digital CAD model is correctly positioned and orientated. The resulting weld after the calibration, as seen in Fig. 11, shows a significant improvement in quality.

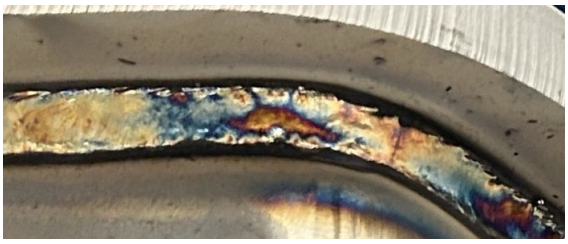

Figure 11: A corner of the workpiece showing significantly improved welding quality after system re-calibration.

6 Conclusion and Further Work

Applying model checking to an already existing industrial robotic system with known weaknesses has proved to be both challenging and useful. The main challenge lies in ensuring that the model catches the essential characteristics of the real-life system. Abstractions and assumptions need to be made to keep the computational complexity at a reasonable level. However, it is crucial that they are not so limiting that the model fails to capture the behaviour and possible errors. Keeping an eye on this so-called "reality gap" between the model and the real system is vital.

Even though improvements have been made on the real-life system based on the findings from the model checking, there is still room for improvement. An interesting path for further work is to verify the assumptions made on the hardware, to achieve a co-verification similar to that in [15]. It is also beneficial to verify the offline programming in Delfoi, and set requirements for the generation of waypoints. This will ensure the feasibility of the planned trajectories.

Further work will use the model further along the RoboStar workflow in [5]. The next step is to automatically generate a simulation model RoboSim [3]. The RoboStar team is also working on model-based testing, so we can generate forbidden traces from RoboChart models. They are specifications of tests that can then be run against the real-life software. This is an interesting way to address the reality gap.

Acknowledgements

David Anisi has received partial funding from the Norwegian Research Council (RCN) RoboFarmer, project number 336712. Ana Cavalcanti and Pedro Ribeiro are funded by the Royal Academy of Engineering (Grant No CiET1718/45), and the UKRI (UK Research and Innovation Council), Grants No EP/R025479/1 and EP/V026801/1.

References

[1] Christel Baier & Joost-Pieter Katoen (2008): *Principles of model checking*. MIT Press.

[2] J. Baxter, P. Ribeiro & A. L. C. Cavalcanti (2022): *Sound reasoning in tock-CSP*. Acta Informatica 59, pp. 125–162, doi:10.1007/s00236-020-00394-3.

[3] A. L. C. Cavalcanti, A. C. A. Sampaio, A. Miyazawa, P. Ribeiro, M. Conserva Filho, A. Didier, W. Li & J. Timmis (2019): *Verified simulation for robotics*. Science of Computer Programming 174, pp. 1–37, doi:10.1016/j.scico.2019.01.004. Available at papers/CSMRCD19.pdf.

[4] Ana Cavalcanti, Will Barnett, James Baxter, Gustavo Carvalho, Madiel Conserva Filho, Alvaro Miyazawa, Pedro Ribeiro & Augusto Sampaio (2021): *RoboStar Technology: A Roboticist's Toolbox for Combined Proof, Simulation, and Testing*. Springer International Publishing, doi:10.1007/978-3-030-66494-7_9.

[5] Ana Cavalcanti, Will Barnett, James Baxter, Gustavo Carvalho, Madiel Conserva Filho, Alvaro Miyazawa, Pedro Ribeiro & Augusto Sampaio (2021): *RoboStar Technology: A Roboticist's Toolbox for Combined Proof, Simulation, and Testing*. In: Software Engineering for Robotics, Springer, doi:10.1007/978-3-030-66494-7_9. Available at https://link.springer.com/10.1007/978-3-030-66494-7_9.

[6] HeeSun Choi, Cindy Crump, Christian Duriez, Asher Elmquist, Gregory Hager, David Han, Frank Hearl, Jessica Hodgins, Abhinandan Jain, Frederick Leve et al. (2021): *On the use of simulation in robotics: Opportunities, challenges, and suggestions for moving forward*. Proceedings of the National Academy of Sciences 118, doi:10.1073/pnas.1907856118.

[7] Edmund M. Clarke (1997): *Model checking*. In: Foundations of Software Technology and Theoretical Computer Science, 1346, Springer Berlin Heidelberg, doi:10.1007/BFb0058022. Available at http://link.springer.com/10.1007/BFb0058022.

[8] J Edward Colgate, Witaya Wannasuphoprasit & Michael A Peshkin (1996): *Cobots: Robots for collaboration with human operators*. In: ASME international mechanical engineering congress and exposition, 15281, American Society of Mechanical Engineers, doi:10.1115/IMECE1996-0367.

[9] Eclipse Foundation (visited August 5, 2024): *Eclipse website*. Available at http://www.eclipse.org/.

[10] Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov & Andrew W. Roscoe (2014): *FDR3 — A Modern Refinement Checker for CSP*. In: Tools and Algorithms for the Construction and Analysis of Systems, Springer Berlin Heidelberg, doi:10.1007/978-3-642-54862-8_13.

[11] C. A. R. Hoare (1978): *Communicating sequential processes*. Communications of the ACM 21, doi:10.1145/359576.359585. Available at https://dl.acm.org/doi/10.1145/359576.359585.

[12] P Kah, M Shrestha, E Hiltunen & J Martikainen (2015): *Robotic arc welding sensors and programming in industrial applications*. International journal of mechanical and materials engineering 10, doi:10.1186/s40712-015-0042-y.

[13] A. Miyazawa, P. Ribeiro, W. Li, A. L. C. Cavalcanti, J. Timmis & J. C. P. Woodcock (2019): *RoboChart: modelling and verification of the functional behaviour of robotic applications*. Software & Systems Modeling 18(5), pp. 3097–3149, doi:10.1007/s10270-018-00710-z.

[14] Alvaro Miyazawa, Pedro Ribeiro, Wei Li, Ana Cavalcanti, Jon Timmis & Jim Woodcock (2019): *RoboChart: modelling and verification of the functional behaviour of robotic applications*. Software & Systems Modeling 18, doi:10.1007/s10270-018-00710-z. Available at http://link.springer.com/10.1007/s10270-018-00710-z.

[15] Yvonne Murray, Martin Sirevåg, Pedro Ribeiro, David A. Anisi & Morten Mossige (2022): *Safety assurance of an industrial robotic control system using hardware/software co-verification*. Science of Computer Programming 216, doi:10.1016/j.scico.2021.102766. Available at https://linkinghub.elsevier.com/retrieve/pii/S0167642321001593.

[16] Henrik Nordlie (2024): *Formal verification of synchronization properties of a multi-robot welding system*. Master's thesis, Norwegian University of Life Sciences, Ås, Norway.

[17] Pioneer Robotics AS (visited August 15, 2024): *IntelliWelder - UR+ certified product*. Available at https://www.pioneer-robotics.no/cobot/intelliwelder/.

[18] J Norberto Pires, Altino Loureiro & Gunnar Bölmsjo (2006): *Welding robots: technology, system issues and application*. Springer Science & Business Media, doi:10.1007/1-84628-191-1.

[19] Gang Ren, Qingsong Hua, Pan Deng, Chao Yang & Jianwei Zhang (2017): *A Multi-Perspective Method for Analysis of Cooperative Behaviors Among Industrial Devices of Smart Factory*. IEEE Access 5, doi:10.1109/ACCESS.2017.2708127.

[20] Günther Starke, Daniel Hahn, Diana G. Pedroza Yanez & Luz M. Ugalde Leal (2016): *Self-organization and self-coordination in welding automation with collaborating teams of industrial robots*. Machines (Basel) 4, doi:10.3390/machines4040023.

[21] THG Automation (visited August 19, 2024): *In Sync: The Benefits of Coordinated Motion*. Available at https://thgautomation.com/2024/06/27/in-sync-the-benefits-of-coordinated-motion/.

[22] Universal Robots (visited August 5, 2024): *Universal Robots - UR10e Website*. Available at https://www.universal-robots.com/products/ur10-robot/.

[23] Universal Robots (visited August 8, 2024): *Real-Time Data Exchange Guide*. Available at https://www.universal-robots.com/articles/ur/interface-communication/real-time-data-exchange-rtde-guide/.

[24] Universal Robots (visited July 23, 2024): *Universal Robots e-Series User Manual*. Available at https://www.universal-robots.com/download/manuals-e-seriesur20ur30/user/ur10e/59/user-manual-ur10e-e-series-sw-59-english-international-en/.

[25] University of Oxford (visited August 15, 2024): *FDR4 - The CSP Refinement Checker*. https://cocotec.io/fdr/. Available at https://cocotec.io/fdr/.

[26] Visual Components (visited August 15, 2024): *Robot Offline Programming*. Available at https://www.visualcomponents.com/products/robot-offline-programming/.

[27] Jiahao Xiong, Zhongtao Fu, Miao Li, Zhicheng Gao, Xiaozhi Zhang & Xubing Chen (2021): *Trajectory-Smooth Optimization and Simulation of Dual-Robot Collaborative Welding*. In: Intelligent Robotics and Applications, 13014, Springer, doi:10.1007/978-3-030-89098-8_66.