Monte Carlo Tree Search and GR(1) Synthesis for Robot Tasks Planning in Automotive Production Lines

1st Eric Wete
Leibniz Universität Hannover — Volkswagen AG
Hannover — Wolfsburg, Germany
eric.roslin.wete.poaka@stud.uni-hannover.de

2nd Joel Greenyer
FHDW Hannover
Hannover, Germany
joel.greenyer@fhdw-hannover.de

3rd Andreas Wortmann
Universität Stuttgart
Stuttgart, Germany
andreas.wortmann@isw.uni-stuttgart.de

4th Oliver Flegel
Volkswagen AG
Wolfsburg, Germany

5th Martin Klein
Volkswagen AG
Wolfsburg, Germany

***Abstract*—- In automotive production cells, complex processes involving multiple robots must be optimized for cycle time. We investigated using symbolic GR(1) controller synthesis for automating multi-robot task planning. Given a specification of the order of tasks and states to avoid, often multiple valid strategies can be computed; in many states there are multiple choices to satisfy the specification, such as choosing different robots to perform a certain task. To determine the best choices under the consideration of movement times and probabilities that robots may be interrupted for repairs or corrections, we combine the execution of the synthesized controller with Monte Carlo Tree Search (MCTS), a heuristic AI-planning technique. The result is a model-at-run-time approach that we present by the example of a multi-robot spot welding cell. We report on experiments showing that the approach (1) can reduce cycle times by choosing time-efficient movement sequences and (2) can choose executions that react efficiently to interruptions by choosing to delay tasks that, if an interruption of one robot should occur later, can be reallocated to another robot. Most interestingly, we found, however, that (3) in some cases there is a conflict between time-efficient movement sequences and ones that may react efficiently to probable future interruptions—and when interruption probabilities are low, increasing the time allocated for MCTS, i.e., increasing the number of sample simulations made by MCTS, does not improve cycle time.**

***Index Terms*—Robot tasks planning, Reactive systems, Monte Carlo Tree Search**

I. INTRODUCTION

Programming multi-robot choreographies in automotive production cells, such as spot welding cells, is a challenge. The point-to-point movement trajectories and the order of robot tasks must be optimized for cycle times while avoiding collisions. Moreover, it can happen that interruptions may occur due to damages of the robots' tools or because other manual corrections to the process are necessary.

Every time a production cell is set up, due to spatial/architectural particularities on the shop floor, the multi-robot choreographies must be adapted or re-programmed, which is a manual and time-consuming process. Due to the complexity of the task, capabilities such as reacting dynamically to interruptions, for example by re-allocating open tasks to other robots, is usually an unattainable goal, as the space of different states explodes that must be considered in this manual process.

We investigated how reactive controller synthesis from requirement specifications can be applied for automating the task of efficient multi-robot task planning. Especially, since efficient algorithms were developed for synthesizing controllers from GR(1) specifications [1]–[5], a subset of LTL, applying such techniques appeared promising. However, current synthesis tools, such as SPECTRA [4], [5], do not support real-time constraints and stochastic processes. Moreover reactive controller synthesis is obviously not suited for finding efficient point-to-point movement trajectories. For the latter problem various trajectory planning approaches exist that shall not be the focus of this paper. However, point-to-point movement times and possible collisions that may occur, are relevant for planning safe and time-efficient choreographies.

We investigated solving the multi-robot choreography synthesis problem in three stages:

(1) We apply automatic trajectory planning algorithms for all the possible point-to-point movements of each robot. Trajectory combinations where robots are at the risk of colliding, are identified, and movement times are stored for later use.

(2) We apply SPECTRA's synthesis algorithm to compute a controller from a GR(1) specification that, in a nutshell, requires all tasks to completed in each cycle, while adhering to certain constraints in the order of tasks, and avoiding robot collisions as identified in the previous stage. The specification also considers the occurrence of interruptions. However, we neither regard interruption probabilities nor interruption and movement times. This GR(1) specification, in fact, need not be written by hand, but can be generated from tasks and constraints that robot programmers can enter in ABB

RobotStudio. If the specification is realizable, SPECTRA is able to synthesize a controller, i.e., one that contains all valid control strategies. The controller represents many strategies (in a rather compact way, thanks to its symbolic representation), but not necessarily all of the possible strategies. SPECTRA also ships a controller execution engine. We connected this engine via ABB's Robot Web Services to the robot cell, in order to receive the necessary sensor inputs and control the movements of the (real or simulated) robots. I.e., if after the fulfillment of a task, the synthesized controller dictates a particular task to be performed next, the corresponding trajectory, as calculated in stage (1), is executed.

(3) Since movement and interruption times as well as interruption probabilities are not considered by the controller produced at stage (2), we integrated the controller execution engine with a heuristic sample-based planning technique, namely Monte Carlo Tree Search (MCTS). The goal of MCTS is to find, at run-time, time-efficient task execution sequences, which includes finding sequences that are able to react time-efficiently to interruptions by re-allocating tasks to other robots. MCTS is a heuristic planning technique that gained traction in AI (cf. [6]); by continuously performing forward-looking simulations, it can often determine beneficial actions even in complex settings. Moreover, it can often find good choices even if the planning time is limited, and can gradually improve the quality of its predictions the more processing time it is granted.

In order to apply MCTS, we built a simple simulation engine that implements the assumptions of the interruption probabilities and durations, and reflects the movement times as determined in stage (1). Coupled with the synthesized controller, executed by SPECTRA's execution engine, this forms a model-at-run-time that is used by MCTS to generate and evaluate sample forward simulations of the system.

Results: We evaluated our approach using a real industrial multi-robot spot welding cell. We found that we can successfully reduce cycle times, as MCTS is able to choose time-efficient movement sequences and can choose executions that react efficiently to interruptions by choosing to delay such tasks that, if an interruption of one robot should occur later, can be re-allocated to another robot. Most interestingly, we found, however, that in some cases there is a conflict between time-efficient movement sequences and sequences that may react efficiently to probable future interruptions—and when interruption probabilities are low, increasing the number of iterations in MCTS may not allow MCTS to perform better.

Regarding the scalability of this approach, we found that SPECTRA struggles to synthesize controllers in cases of more than three robots, but it seems to be due to problems in an internal pre-processing step that can be overcome in the future.

Contribution: Although there is work to support or solve controller synthesis using planning algorithms (cf. [7], [8]), these methods are not used to optimize the controller towards other, nonfunctional requirements; to the best of our knowledge, our approach is the first to demonstrate a beneficial combination of controller synthesis and AI planning techniques for efficiently controlling a timed and probabilistic system. At run-time our approach is able to handle interruptions by choosing among different scheduling strategies and considers task re-allocation. The results we obtain in the case of low interruption probabilities motivate us to investigate techniques to handle rare events more reliably, or learn a model of the controlled system including interruptions probabilities.

The technique presented in this paper is an MDE technique that, by utilizing trajectory planning, formal controller synthesis, and on-line planning, significantly lifts the abstraction level compared to how multi-robot production cells are programmed in practice today. Offering engineers this level of abstraction is not a limitation; in fact, within the abstract solution space defined by the specification, synthesis and planning algorithms can find correct and optimized solutions for multi-robot choreographies. We show patterns of specifying robot cell requirements in SPECTRA's GR(1) language that can be applied similarly for other robot cells with different numbers of robots, different tasks, different assumptions, and different safety and liveness requirements.

Structure: We cover foundations in Sect. II, overview the methodology in Sect. III, and describe the multi-robot cell GR(1) specification and synthesis in Sect. IV. Sect. V illustrates the integration of MCTS for online planning, and Sect. VI describe our industrial case example. Sect. VII shows the results of our method, which are discussed in Sect. VIII. Sect. IX overviews related work and we conclude in Sect. X.

II. FOUNDATIONS

A. Multi-robot tasks planning

We define a *task* to be an activity to be performed by a robot and which takes certain period of time. A task is *reallocatable* when it can be performed by more than one robot.

Task planning is the activity to define for a robot the order of tasks to perform (*task plan*). **Multi-robot tasks planning** is the task planning for a multi-robot cell.

The task plan can be *fixed* or *flexible*. In a fixed task plan, the order of tasks is always the same in all cycles. In a flexible plan, the task order can change or tasks can be re-allocated, i.e. carried out by different robots, in each cycle.

B. Reactive synthesis

(The content of this subsection strongly follows [5])

1) Linear Temporal Logic (LTL): Linear Temporal Logic (LTL) is defined as a modal temporal logic with modalities referring to time [9], [10].

LTL is made up of a finite set of atomic propositions AP, Boolean connectors $\neg$, $\wedge$, $\vee$, and $\rightarrow$, and temporal operators $\boldsymbol{X}$ (next), $\boldsymbol{G}$ (globally), $\boldsymbol{F}$ (in the future), $\boldsymbol{U}$ (until), $\boldsymbol{Y}$ (previous) and $\boldsymbol{S}$ (since).

a) LTL Syntax: The syntax of an LTL formula φ is defined by the following grammar:

$\varphi := p \mid \neg\varphi \mid \varphi \vee \varphi \mid \boldsymbol{X}\varphi \mid \boldsymbol{G}\varphi \mid \boldsymbol{F}\varphi \mid \varphi\, \boldsymbol{U}\, \varphi \mid \boldsymbol{Y}\varphi \mid \varphi\, \boldsymbol{S}\, \varphi$ where $p \in AP$.

Let $p \in AP$, φ and ψ be LTL formulas. The following Boolean operators can be derived:

- ***true*** $= p \vee \neg p$
- ***false*** $= \neg$ ***true***
- $\varphi \wedge \psi = \neg(\neg \varphi \vee \neg \psi)$
- $\varphi \rightarrow \psi = \neg \varphi \vee \psi$

b) Computation: A computation $\pi = \pi_0\pi_1\pi_2\cdots$ is an infinite sequence of truth assignments to propositions, where $\pi_i \in 2^{AP}$ denotes a computation π at time instant i. π_i is a set of propositions at position i.

c) Satisfiability: A formula φ *holds on a computation* π iff $\pi, 0 \models \varphi$, which is denoted by $\pi \models \varphi$. We say that the computation π *satisfies* φ.

A set of computations M *satisfies* φ, written as $M \models \varphi$, if every computation in M satisfies φ.

d) Additional operators and equivalences: There are others operators and some equivalences of LTL operators:

- $\boldsymbol{F}\,\varphi \equiv$ ***true*** $\boldsymbol{U}\,\varphi$
- $\boldsymbol{G}\,\varphi \equiv \neg \boldsymbol{F} \neg \varphi$
- $\varphi\, \boldsymbol{W}\, \psi \equiv (\varphi\, \boldsymbol{U}\, \psi) \vee \boldsymbol{G}\,\varphi$ (weak until)
- $\boldsymbol{H}\,\varphi \equiv \neg$ (***true*** $\boldsymbol{S} \neg \varphi$) (historically)

2) GR(1) Synthesis: GR(1) synthesis considers a subset of LTL where the specifications are made up of initial assumptions and guarantees defining initial states, safety assumptions and guarantees referring to the current and next state, and justice assumptions and guarantees. which denote assertions about what should hold infinitely often during a computation. A GR(1) specification is made up of the following elements [10]:

1) $\mathcal{X}$ input variables controlled by the environment
2) $\mathcal{Y}$ output variables controlled by the system
3) $\mathcal{X}'$ and $\mathcal{Y}'$ copies of input and output variables at the next step
4) θ^e assertion over $\mathcal{X}$ characterizing initial environment states
5) θ^s assertion over $\mathcal{X} \cup \mathcal{Y}$ characterizing initial system states
6) $\rho^e(\mathcal{X} \cup \mathcal{Y} \cup \mathcal{X}')$ transition relation of the environment
7) $\rho^s(\mathcal{X} \cup \mathcal{Y} \cup \mathcal{X}' \cup \mathcal{Y}')$ transition relation of the system
8) $\mathcal{J}^e_{i \in 1..n}$ justice goals of the environment
9) $\mathcal{J}^s_{j \in 1..m}$ justice goals of the system

The tuple $\langle \theta^e, \rho^e, \mathcal{J}^e \rangle$ defines environment specifications and $\langle \theta^s, \rho^s, \mathcal{J}^s \rangle$ system specifications. A GR(1) specification is *realizable* if the following formula is realizable:

$$\varphi^{sr} = (\theta^e \rightarrow \theta^s) \wedge (\theta^e \rightarrow \boldsymbol{G}((\boldsymbol{H}\rho^e) \rightarrow \rho^s)) \wedge (\theta^e \wedge \boldsymbol{G}\rho^e \rightarrow (\bigwedge_{i \in 1..n} \boldsymbol{GF}\mathcal{J}^e_i \rightarrow \bigwedge_{j \in 1..m} \boldsymbol{GF}\mathcal{J}^s_j))$$

For GR(1), the specifications must be expressible in the structure defined above and therefore it does not cover complete LTL. GR(1) [10] considers a subset of LTL for which realizability and synthesis is solved in time exponential in the size of the LTL formula and polynomial in the resulting controller. [3], [10] proposed efficient symbolic algorithms for GR(1) realizability checking and controller synthesis.

3) Reactive systems specification: SPECTRA is a specification language for reactive systems with the expressive power of GR(1), and particularly appropriated for reactive synthesis [5]. The specification language comes with a set of tools. Among others, a synthesizer to obtain a correct-by-construction implementation, and several means for executing the resulting controller.

A SPECTRA specification contains variables, assumptions and guarantees. The variable names are unique and a variable can be referenced from anywhere inside the specification. However, assumptions and guarantees can not be referenced inside the specification.

Variables have a type and are controlled by the environment (declared with the keyword **`env`**) or by the system (declared with the keyword **`sys`**).

Assertions over the input (resp. the output) variables, transitions relation of the environment (resp. the system) and justice goals of the environment (resp. the system) are declared as initial, safety ($\boldsymbol{G}$) and justice ($\boldsymbol{GF}$) assumptions with the keyword **`asm`** (resp. **`gar`**).

C. Monte Carlo Tree Search

MCTS has received considerable interest due to its success in the problem of computer Go [6], [11]–[13].

The main idea behind MCTS is that generating a number of sample forward simulations is an effective and efficient way to estimate the value of an action, and the action's values can be used to optimize the control strategy [11].

The MCTS algorithm builds possible future game states (a search tree) according to the outcomes of simulated playouts. As long as the resources allow it, the MCTS repeats the four following phases: (1) Selection, (2) Expansion, (3) Simulation and (4) Backpropagation. The steps are illustrated in Fig. 1.

1) **Selection**: Starting at the root node ***R***, select optimal child nodes recursively until a leaf node (i.e., not fully expanded) ***L*** is reached. The selection is based on the tree policy.
2) **Expansion**: If the node ***L*** does not end the game with a win/loss for either player, create one (or more) child nodes according to the available actions and randomly select one of them, ***C***. Add ***C*** to the tree.
3) **Simulation**: Run a simulation from ***C*** until a result is achieved. Default Policy determines how simulation is run, e.g., make random moves repeatedly until the robot cell cycle ends.
4) **Backpropagation**: Update the move sequence from ***C*** to ***R*** with the simulation result, i.e., the outcome is backpropagated through the selected nodes to update their statistics. The visit counts are increased and the node value is updated according to the outcome (cycle time).

The tree policy is used to determine how children are selected and the default policy is used to determine how simulations are run (e.g. randomized).

MCTS maintains the balance for selection between exploration and exploitation. The tree policy is based on the UCT (Upper Confidence Bound 1) value computed with [13]. Since the goal of the system is to make best decisions, the intuition behind that is the more the system makes an action, the more this action will be taken (exploitation). But this strategy may

make the system to not find a better action (exploration). Moreover, if the system tries often new action (exploration) then it may probably discover often worst decisions. The system action or node chosen is the one with the highest value:

$$\frac{w_i}{n_i} + c.\sqrt{\frac{\ln n}{n_i}}$$

- $w_i \equiv$ number of wins (i.e., simulations leading to cycle time improvement) after the i-th move
- $n_i \equiv$ number of simulations after the i-th move
- $c \equiv$ balance between exploration and exploitation (typical: $\sqrt{2}$).
- $n \equiv$ total number of simulation for the node considered ($n = \sum_i n_i$).

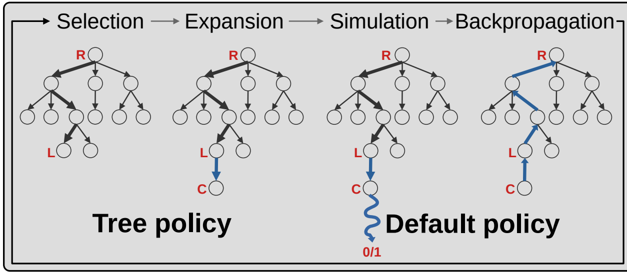


Fig. 1. Monte Carlo Tree Search iterations overview

When all the allocated resource time is consumed, the move played by the program is the most promising child node of the root node.

III. Methodology overview

We assume that a spatial model of the production cell was created, for example using standard tools such as ABB RobotStudio. Moreover, we assume that the robot tasks, especially re-allocatable tasks, along with working point coordinates are defined, efficient movement trajectories between the task locations were calculated for each robot, and trajectory and task allocation combinations with the danger of robots colliding were identified.

Now, our technique is based on the process shown in Fig. 2. The process is made up of the following steps:

(1) Reactive system specification: Based on the robot's task definition, collision constraints, and other requirements on the allowed order of task fulfillment, we derive a Spectra GR(1) multi-robot cell specification. The specification also considers robots interruptions and specifies how the robot cell shall react to them. For example, if a robot's tool is damaged, it may be required to move to a location where it shall be replaced, and then continue processing its tasks. The specification also includes assumptions, for example that a robot ordered to perform a task will eventually be finished with the task. We use the Spectra analysis tools to check whether the specification is realizable. It could be unrealizable, for example, if there is a conflict among task order or collision constraints.

(2) Controller synthesis: We perform controller synthesis from the system specification. The resulting outputs is a controller that describes valid control strategies that satisfy the specification.

(3) Strategy validation and execution: We integrate the resulting controller with a model at run-time to control the robot cell at run time and allow the controller to make time-optimal decisions. The synthesized controller execution is therefore tuned by our model at run-time. In order to simulate interruptions, we developed robot interruption models that also reflect interruption probabilities from our experience.

An overview of the architecture of the robot cell control loop is shown in Fig. 3. We developed adapters that connect Spectra's controller executor with the robot cell through a robot specific API (ABB Robot Web Services).

(1) Our adapter reads the robots' output signals to determine if any robot is interrupted (due to a mechanical failure) and can not perform a task. In addition, the robot cell gives information about the completion of all tasks.
(2) The status of the robots and tasks are transformed to inputs for the controller executor. The inputs consist of
    - the robots' status (interrupted or not), and
    - the status about each task (completed or not).

    The controller executor, based on the synthesized controller, processes the inputs and produces outputs in the form of next task assignments for each robot. The execution of the controller can be combined with an MCTS-based method as described in Sect. V.
(3) Our tool parses the controller's outputs and
(4) converts the task assignments to robot-specific function calls. The we repeat again from step (1).

IV. Robot Cell Controller Specification

The controller specification is derived from the robot cell requirements and constraints. First, the specification defines the robot cell variables. Environment variables represent the status of the robot cell: the robots' status and tasks' status; system variables represent the robot task assignments. Second, the specification defines the environment behavior as assumptions and the system behaviour as guarantees.

The assumptions are defined as follows:

(A1) At cycle start, no tasks are completed, no robots are interrupted.
(A2) When a cycle ends, a new cycle eventually starts.
(A3) Each robot will complete its assigned task after a finite number of steps.
(A4) At any time a robot can be interrupted (e.g. due to mechanical failure).
(A5) A task is completed if a robot performs it successfully.
(A6) An interrupted robot must not be able to complete a task.

The system behaviour is defined by the following rules:

(R1) At cycle start, robots must be at their home location.
(R2) Tasks assignments are restricted to the robots declared for this tasks.

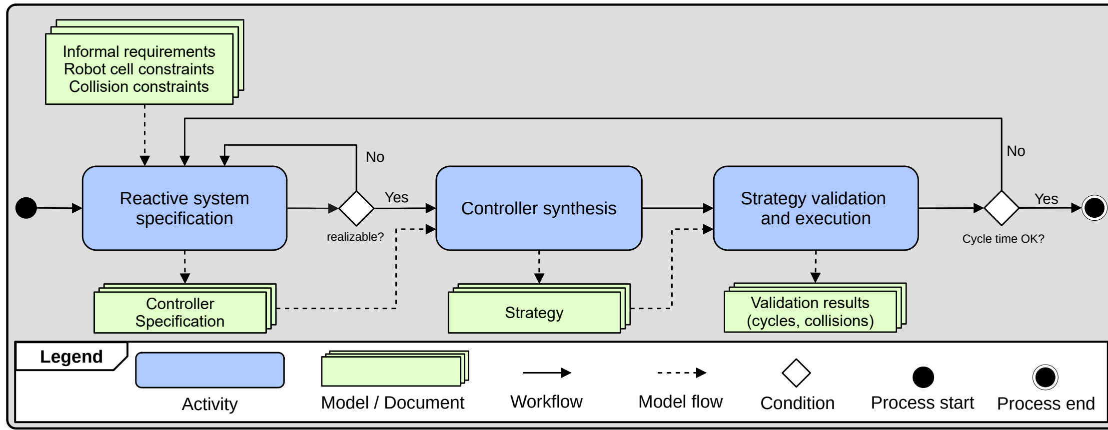


Fig. 2. Reactive robot task planning process

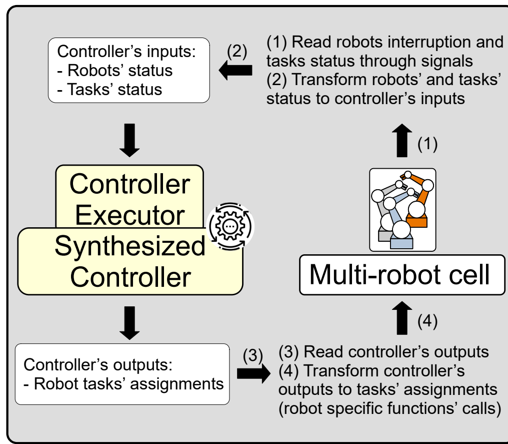


Fig. 3. Strategy execution in a multi-robot cell system

- (R3) When a task is assigned to a robot, the assignment must not change before robot completes the task, unless the robot is interrupted.
- (R4) Inside a cycle, each task is completed once.
- (R5) To prevent collisions, a task must not be assigned at same time to more than one robot.
- (R6) If a robot cannot perform a task, it must move to its home location (e.g., for repair) and its task can be re-assigned to another robot.
- (R7) A robot must not go back to the home location if any of its non-reallocatable tasks have not been completed, except if the robot is interrupted.
- (R8) A task must not start if one of its dependent tasks is not completed.
- (R9) Tasks assignments that lead to collisions must not occur.

V. Monte Carlo Tree Search Integration

In some states, the resulting synthesized controller has many possibilities to assign the next robot tasks. The default controller executor of SPECTRA chooses a random assignment among these possibilities. However, this choice mechanism is not optimal.

We apply an MCTS algorithm that evaluates each of the possibilities by simulating multiple steps into the future, until the end of the current cycle. The most promising task assignment is then chosen.

Our MCTS algorithm simulates a game in which the system behavior is determined by the synthesized controller and the environment behavior by our Java-based environment simulation, which embodies the environment assumptions as in Sect. IV, real-valued movement times, and interruption times and probabilities.

The resulting game tree consists of nodes. A node's outgoing transition is a node *action*, which corresponds to a synthesized controller system action. Our environment simulation sets the inputs of the controller (robot cell status) whereas the synthesized controller executor aims to compute the output of the controller (robot task assignments). The game tree starts with a node, in which the inputs are set and a system action has to be taken, i.e., an assignment of controller outputs. The available actions of a system node are computed by querying the synthesized controller through the controller executor. The assumptions of the environment are realized by our Java-based environment simulation.

As default policy we used a random-based approach. At the end of the simulation phase of an MCTS iteration, the cycle time is computed, since we always execute the simulation until the cycle ends. The best choice at the end of the MCTS simulations is the *action* which leads to the most time-efficient *actions*' sequence, i.e., robot task assignments.

VI. Running Example

A. Robot cell description

We illustrate our method with a spot welding cell with two spot welding robots. The workpiece is a vehicle body that must be welded on eight points. Figure 4 shows an illustration.

The robot cell consists of:

- Two robots (R0) and (R1). Their base position are (0) and (5), respectively.

- Eight welding spots located at positions (1), (2), (3), (4), (6), (7), (8), (9).

Each welding spot represents a task.

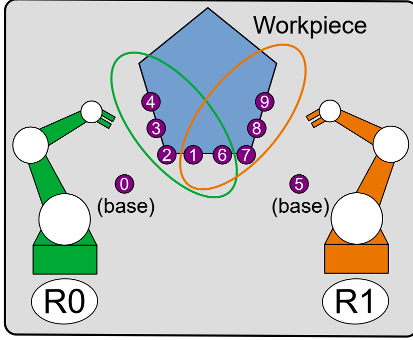


Fig. 4. Use case: A spot welding robot cell illustration

1) Robots constraints: Due to their kinematic constraints, each robot has a limited workspace (ellipses in Fig. 4) and thus, has a set of tasks it can perform:

C1: (R0) can perform tasks at (1), (2), (3), (4), (6).

C2: (R1) can perform tasks at (1), (6), (7), (8), (9).

We can see that this robot cell has two reallocatable tasks at (1) and (6). The other tasks are non-reallocatable.

2) User requirements: We have some requirements about the corners of the part. The corners must be welded before its adjacent points.

D1: (2) must be weld before (1). (3) and (4).

D2: (7) must be weld before (6), (8) and (9).

3) Collision constraints: A collision will occurs if (R0) is at location (6) and (R1) at location (1).

B. Reactive system specification

This section describes the reactive system specification that can be derived from these constraints, based on the properties described in Sect. IV and Sect. VI-A.

1) Constants and variables: All line numbers in this section are related to the Listing 1 of the specification.

The specification starts by declaring its name in line 1, the constants regarding the number of the robots, the welding points and the home position of robots in lines 3-22. The constant `basePositions` holds base location and `robotTasks` holds the tasks of the robots. Lines 28-31 define the types `Robot` and `Location` to model the robots and locations. The specification models the robot cell status consisting of task and robot status in lines 33-36 with the environment variables `isCompleted` and `isInterrupted` respectively. Our model specifies task assignment with the system variable `taskAssignement` in lines 38-39.

2) Assumptions: All line numbers in this section are related to the Listing 2 of the specification.

The environment assumptions are described in Sect. IV and modelled in the specification as assumptions. Lines 3-9 of the specification model define (A1) with initial assumptions. (A2)

```
1  module Spec020802
2
3  // Number of robots
4  define NUM_OF_ROBOTS := 2;
5  define NUM_OF_ROBOTS_MINUS_ONE := NUM_OF_ROBOTS - 1;
6
7  // Number of point's locations
8  define NUM_OF_LOCATIONS := 10;
9  define NUM_OF_LOCATIONS_MINUS_ONE := NUM_OF_LOCATIONS -
       1;
10
11 // Maximum number of tasks
12 define MAX_NUM_OF_TASKS := NUM_OF_LOCATIONS -
       NUM_OF_ROBOTS;
13 define MAX_NUM_OF_TASKS_MINUS_ONE := MAX_NUM_OF_TASKS -
       1;
14
15 // Define robot base location constant
16 define basePositions[NUM_OF_ROBOTS] := { 0, 5 };
17
18 // Define robot tasks
19 define robotTasks[NUM_OF_ROBOTS][MAX_NUM_OF_TASKS] := {
20   { 1, 2, 3, 4, 6, -1, -1, -1 },
21   { 1, 6, 7, 8, 9, -1, -1, -1 }
22 };
23
24 // Type to iterate over the 2D arrays "robotTasks"
25 type TaskIndex = Int(0..MAX_NUM_OF_TASKS_MINUS_ONE);
26
27 // Robot cell robots
28 type Robot = Int(0..NUM_OF_ROBOTS_MINUS_ONE);
29
30 // Robot cell locations
31 type Location = Int(0..NUM_OF_LOCATIONS_MINUS_ONE);
32
33 // Tasks status
34 env boolean[NUM_OF_LOCATIONS] isCompleted;
35 // Robots status
36 env boolean[NUM_OF_ROBOTS] isInterrupted;
37
38 // Task assignment of each robot
39 sys Location[NUM_OF_ROBOTS] taskAssignment;
40
41 // Define predicate specifying the end of a welding
       cycle
42 predicate IsCycleCompleted():
43   ((forall loc in Location. isCompleted[loc] = true) & (
        forall rob in Robot. isInterrupted[rob] = false)
        & (forall rob in Robot. taskAssignment[rob] =
        basePositions[rob]));
44
45 // Define predicate specifying that the location is not
       a robot target
46 predicate IsLocationNotATarget(Location loc):
47   (forall rob in Robot. taskAssignment[rob] != loc);
48
49 // ends first part ...
```

Listing 1. Specification of multi-robot cell: Constants and variables

defines the restart of a cycle that represents the completion of all tasks. It is designed as safety assumption at lines 23-28. Lines 11-15 model the robot task completion with the corresponding task status and implements the assumption (A3) and (A5). Since robot interruptions can happen at any time (A4), there is no need to write assumption on environment variable `isInterruped`. The specification states in lines 30-34 the assumption (A6): when a robot is interrupted it must not complete any task.

3) Guarantees: All line numbers in this section are related to the Listing 3 of the specification.

The system rules are described in Sect. IV and modelled in the specification as guarantees. Lines 3-6 of the specification model (R1) state a rule about the start of a cycle with a safety guarantee. It specifies that the task assignement of a robot

```
1  // continues first part ...
2
3  // No task is initially completed
4  asm InitiallyNoTaskIsCompleted:
5  forall i in Location. isCompleted[i] = false;
6
7  // No robot is initially interrupted
8  asm InitiallyNoRobotIsInterrupted:
9  forall i in Robot. isInterrupted[i] = false;
10
11 // Each robot will complete its assigned task after a
       number of steps
12 asm RobotsWillCompleteATaskAfterXStep:
13 G forall r in Robot. forall t in Location.
14 (taskAssignment[r] = t & !IsCycleCompleted()) ->
15 ((next(isCompleted[t]) = isCompleted[t]) | (next(
       isCompleted[t]) = true));
16
17 // Task status must not change if robot does not
       complete it
18 asm NotCompletedTaskKeepsItsStatusUnchanged:
19 G forall l in Location.
20 (IsLocationNotATarget(l) & !IsCycleCompleted() ->
21 (isCompleted[l] = next(isCompleted[l]));
22
23 // Restart new cycle when the current cycle is completed
24 asm RestartCycle:
25 G IsCycleCompleted() ->
26 ((forall j in Location. next(isCompleted[j]) = false)
27   & (forall i in Robot. next(isInterrupted[i]) = false)
28 ) | (forall j in Location. next(isCompleted[j]) = true);
29
30 // Interrupted robots must not perform task
31 asm InterruptedRobotMustNotDoTask:
32 G forall r in Robot. forall t in Location.
33 (isInterrupted[r] & taskAssignment[r] = t & !
       IsCycleCompleted()) ->
34 next(isCompleted[t]) = isCompleted[t];
35
36 // ends second part ...
```

Listing 2. Specification of multi-robot cell: Assumptions

must be its home position at cycle start. The specification implements (R2) by modelling in lines 8-11 robot workspaces based on the matrix of tasks `robotTasks` defined in Listing 1. Lines 13-17 model that when a robot has an assigned task and while is is not interrupted, then this assignment must not change until the task is completed (R3). The system has to make sure that vehicle parts are not welded twice at the same point (R4). We model this rule with the guarantee in lines 19-22, which avoid to make task assignment on already completed tasks. The specification models (R7) in lines 32-39. For each robot, we model a guarantee to states that if all its non redundant tasks are completed then it can go back to its home position, unless it is interrupted. In our case, the robot (R0) must not go back home if the tasks at (2), (3) and (4) are not completed yet. The same rule applies for the robot (R1) with the tasks at (7), (8) and (9).

a) Reactivity to unforeseen events: (R6) specifies what must happen in case of robot interruption. The goal of this rule is to free the working space of the vehicle part so that a neighbored robot can take over the tasks of the failed robot. This rule is modelled in lines 28-30.

b) Specification of task dependencies: The rule (R8) is about task dependency. It makes sure that the system can assign a task only if it has no dependent task which is not completed yet. We specified for our robot cell tasks dependencies in Sect. VI-A2. (D1) is implemented in lines 41-44 and (D2) in lines 46-48.

c) Collision prevention: (R5) avoids that a task is assigned at the same time to different robots. This rule is implemented in lines 24-26.

Sect. VI-A3 defines collision constraints related to task assignment. All possible combinations of task assignment which lead to collision must be avoided (R9). The reactive specification models this rule in lines 50-52.

VII. Results

We implemented a MCTS-based controller executor, which adds a new functionality to the existing Spectra controller executor. Our approach is to use MCTS simulations to make a time-optimal choice of the system actions leading to an optimal cycle time. We made assumptions based on the rate of interruptions which occur in the production lines. We integrated the assumptions as a model of the environment in MCTS simulations. This model allows us to evaluate systems actions.

In each evaluation step of the MCTS algorithm we build the game tree starting from the current system state of the synthesized controller. The children of a system state are environment states. We call a node in which the system must take an action *system node*, while in an *environment node*, the inputs are updated based on the last system outputs. An environment node of our game tree consists of the current inputs and the controller state whereas a system node is made up of the controller outputs. In order to achieve simulations, we implemented a flexible executor which is able to jump from any already discovered state to another one without following a consecutive path starting from the initial state to the state where we would like to jump to.

From any system node in which a system action is needed, we build a game tree with the system and the environment as players. The controller next states aims to generate environment nodes. We design and develop a simulation environment based on the behaviour model of the robot interruptions. This allows us to generate system nodes. The MCTS-simulations are based on the interruptions' assumptions. Since the response time of this kind of system in production lines is important, we choose to use time as resource limitation for the MCTS-simulations instead of the iterations count.

A movement time corresponds to the time needed for a robot to move from a welding point to another. The task time is the time need for the robot at a welding point to perform a weld, i.e., close and open the welding gun. We used the simulation software RobotStudio[1] for our experiment. We first computed the execution time for all possible trajectories and for all tasks. A movement time is around $1.6\,s$ and a task time $0.5\,s$. The time for an interruption is set up to $60\,s$. This helps us to design a simulation environment of a robot cell with robot movements and tasks times. Secondly, based on the robot movements and tasks times we created a simulation of a virtual multi-robot cell. Spectra comes with an executor for

[1] https://new.abb.com/products/robotics/de/robotstudio

```
// continues second part ...

// Robots must start at their home position
gar RobotMustStartAtHome:
G forall r in Robot. forall l in Location. (l = basePositions[r] & isCompleted[l] = false) ->
(taskAssignment[r] = basePositions[r]);

// Each robot can perform only the tasks in its workspace including its home location
gar RobotsWorkspaces:
G forall r in Robot. (taskAssignment[r] = basePositions[r]) |
(exists t in TaskIndex. robotTasks[r][t] > -1 & taskAssignment[r] = robotTasks[r][t]);

// When target is set, target must not change before robot gets there
gar TargetMustNotChangeUntilReached:
G forall r in Robot. (isInterrupted[r] = false) &
(exists l in Location. (isCompleted[l] = false) & (l = taskAssignment[r]) & (l != basePositions[r])) ->
taskAssignment[r] = next(taskAssignment[r]);

// A task must not to be completed twice
gar NoTaskIsCompletedTwice:
G forall l in Location. ((isCompleted[l] = true) & (forall u in Robot. l != basePositions[u])) ->
(forall r in Robot. next(taskAssignment[r] != l));

// 2 robots must not have the same task
gar RobotMustNotHaveSameTask:
G forall r1 in Robot. forall r2 in Robot. (r1 != r2) -> (taskAssignment[r1] != taskAssignment[r2]);

// Interrupted robot must go back home to release its task for re-allocation
gar RobotGoHomeIfInterrupted:
G forall r in Robot. (isInterrupted[r] = true) -> next(taskAssignment[r] = basePositions[r]);

// Robots may go back home if all tasks which can only achieved by them are done
gar Robot_0_MightBackHomeAfterOnlyTasks:
G (isCompleted[2] = false | isCompleted[3] = false | isCompleted[4] = false) & (isCompleted[0] = true) &
(isInterrupted[0] = false) -> next(taskAssignment[0] != 0);

gar Robot_1_MightBackHomeAfterOnlyTasks:
G (isCompleted[7] = false | isCompleted[8] = false | isCompleted[9] = false) & (isCompleted[5] = true) &
(isInterrupted[1] = false) -> next(taskAssignment[1] != 5);

// Tasks dependencies guarantees
gar Dependency_for_2:
G forall r in Robot. (isCompleted[2] = false) ->
(taskAssignment[r] != 1 & taskAssignment[r] != 3 & taskAssignment[r] != 4);

gar Dependency_for_7:
G forall r in Robot. (isCompleted[7] = false) ->
(taskAssignment[r] != 6 & taskAssignment[r] != 8 & taskAssignment[r] != 9);

// Collision constraints
gar CollisionConstraint_0_6_1_1:
G !(taskAssignment[0] = 6 & taskAssignment[1] = 1);

// end third part and specification.
```

Listing 3. Specification of multi-robot cell: Guarantees

the synthesized controller (Default/Random-based algorithm). We made a benchmark to compare the algorithms for 1000 cycles. In our experiments, MCTS simulations end at the end of a cycle.

The time allocated to MCTS impacts the system response time. We find a setup which permits time-efficient planning and an acceptable system response time. We compare the following algorithm executions:

1) Default/Random-based which comes with SPECTRA
2) MCTS-based with time limitation of 500 ms (MCTS_{500})
3) MCTS-based with time limitation of 1000 ms (MCTS_{1000})

The idea is that during run-time there is, on average, a time of 500 ms until the next task assignment must be computed. However, MCTS simulations can also be run in parallel, so comparing MCTS with more computation time is interesting as well.

A. Hardware setup

For our experiments we use an ordinary Personal Computer (PC), Intel (R) Core (TM) i5-8250U CPU @ 1.60GHz, 1800 MHz, 4 core (s), 8 logical processor (s), RAM 16.0 GB, x64-based processor, Windows 10.

B. Experiment 1: No interruptions

Our first experiment investigates the cycle time with the three algorithms (Default, MCTS_{500} and MCTS_{1000}), when no interruption occurs. This builds a baseline for comparison when we experiment with interruptions. Tab. I shows that the average cycle time is $11.06\,s$ for Default. MCTS outperforms significantly with $9.37\,s$ (improvement of $1.69\,s$ with $15.3\,\%$) and $9.27\,s$ (improvement of $1.79\,s$ with $16.2\,\%$) for MCTS_{500} and MCTS_{1000} respectively. We have on average 51 and 103 simulations for MCTS_{500} and MCTS_{1000} respectively.

TABLE I
NO INTERRUPTIONS: CYCLE TIMES

Algorithm	Avg. cycle time (s)	Improvement cycle time (s)
Default (Random)	11.06	
$MCTS_{500}$	9.37	1.69 (15.3 %)
$MCTS_{1000}$	9.27	1.79 (16.2 %)

C. Experiment 2: interruption probability of 0.0125 / new task assignment

In the second experiment we use a probability model for the interruptions and assume that an interruption occurs on a new task assignment with the probability of 0.0125.

The improvements reflect that MCTS outperforms even with this interruption's probability. For instance, Tab. II shows that $MCTS_{500}$ outperforms significantly with a cycle time of $15.55\,s$ whereas Default's cycle time is $17.26\,s$. This corresponds to an improvement of $1.71\,s$ (9.9%). In addition, $MCTS_{1000}$'s cycle time reaches $15.36\,s$ corresponding to an improvement of $1.90\,s$ (11.0%). The small difference between the $MCTS_{500}$'s and $MCTS_{1000}$'s cycle times can be explained by the interruption count which is smaller for $MCTS_{1000}$.

Furthermore, when we analyse the standstill time, we also find out some improvements. The standstill time corresponds to the time where no robots are working due to the interruptions. Even with the same number of interruptions, MCTS optimizes the standstill time. It shows that there are more tasks re-allocations which happened and therefore the robots have less time without working. The results in Tab. II show that the Default has the highest standstill time of $5.72\,s$. $MCTS_{500}$ outperforms with $5.64\,s$ showing an improvement of $0.08\,s$ (1.4%). Moreover, $MCTS_{1000}$ with more efficient task re-allocations outperforms with standstill time of $5.53\,s$ corresponding to an improvement of $0.19\,s$ (3.3%).

D. Experiment 3: interruption probability of 0.0025 / new task assignment

In this experiment we assume that an interruption occurs on a new task assignment with the probability of 0.0025.

Also in this case, MCTS outperforms. Regarding Tab. III, the cycle times of MCTS-based algorithms are much better than Default ($12.39\,s$). The $MCTS_{500}$ has a cycle time value of $10.55\,s$ corresponding to an improvement of $1.84\,s$ ($14.85\,\%$) and the $MCTS_{1000}$'s cycle time reaches $10.65\,s$ showing an improvement of $1.79s$ ($14.45\,\%$).

The same behaviour is observed when we compare the algorithms in terms of standstill time. Our observations reveal that MCTS outperforms the Default algorithm. Tab. III shows that Default has a standstill time of $1.20\,s$. The $MCTS_{500}$'s standstill time of $1.16\,s$ show an improvement of $0.04\,s$, i.e., 3.3%, whereas $MCTS_{1000}$ has a standstill time of $1.14\,s$ corresponding to an improvement of $0.06\,s$, i.e., 5.0%.

Furthermore, we expect to get more improvements by increasing the time allocated to MCTS. However, comparing $MCTS_{500}$ and $MCTS_{1000}$ on the cycle time we do not observe an improvement as in the standstill time.

E. Scalability analysis

We notice scalability issues with some SPECTRA specifications. The synthesis takes $608\,ms$ to compute our case with 2 robots. A specifications with 3 robots takes $15049\,ms$ for synthesis computation, and with 4 robots the process freezes during the SPECTRA specification is being translated to Binary Decision Diagram (BDD). We are investigating to overcome this limitation to any number of robots.

VIII. DISCUSSION

The results of our experiments show that it is possible for MCTS to reduce the cycle time and standstill time because it can predict and exploit interruptions. MCTS can find and apply a strategy which is time efficient and allows re-allocation of robot tasks efficiently. However, an efficient task planning which minimizes the movement time of robots may conflict with the strategies that maintain the potential for a time-efficient re-allocation of task when interruptions occur. This conflict can, but does not have to occur in all cases and we observed it in Sect. VII-D (in rare interruptions).

We illustrate this conflict in Fig. 5. Based on our specification (R0) must first perform task at (2). Therefore our MCTS-based method has to plan the task and must choose one of the 3 schedules showed in Fig. 5. Based on the analysis of movement times of the robot, the least time efficient planning as shown in Fig. 5(a) is (0) → (2) → (4) → (1) → (3) → (0). Fig. 5(b) shows a better planning: (0) → (2) → (3) → (4) → (1) → (0) and Fig. 5(c) illustrates the best time efficient task scheduling (0) → (2) → (1) → (3) → (4) → (0). However, this best planning (Fig. 5(c)) becomes the worst planning if an interruption occurs. At this point, MCTS must deal with this conflict to find a close to time optimal strategy based on its predictions. Since the interruptions are rare, it may happen that the estimated optimal strategy is not the really optimal (as predicted). This phenomenon explains the behaviour of our MCTS algorithms in Sect. VII-D where we do not notice an improvement on cycle time comparing the MCTSs each other by increased time allocation.

We find out that MCTS-based algorithm performs better than Random. When interruptions are rare, increasing the number of MCTS iterations may not improve the performance of the cycle time, but still improving standstill time. The standstill time improvement shows a more efficient task re-allocation.

IX. RELATED WORK

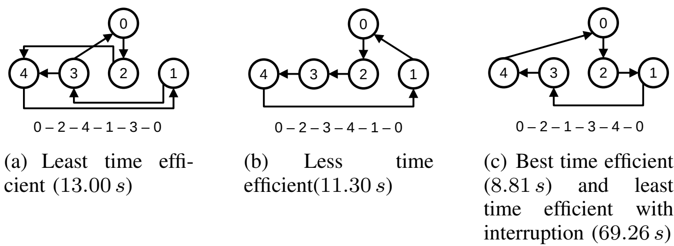
There exist approaches that combine controller synthesis and planning in the sense that controller synthesis is solved using planning algorithms [7], [8]. By contrast, our approach combines synthesized controllers from LTL specifications with a planning algorithm to optimize the controller execution in a timed an stochastic setting.

TABLE II
CYCLE TIMES AND IMPROVEMENTS COMPARISON: INTERRUPTIONS: 0.0125 / NEW TASK ASSIGNMENT

Algorithm	# Interruptions	Avg. cycle time	Avg. standstill time	Improvement cycle time	Improvement standstill time
Default (Random)	107	17.26	5.72		
MCTS_{500}	107	15.55	5.64	1.71 (9.9 %)	0.08 (1.4 %)
MCTS_{1000}	103	15.36	5.53	1.90 (11.0 %)	0.19 (3.3 %)

TABLE III
CYCLE TIMES AND IMPROVEMENTS COMPARISON: INTERRUPTIONS: 0.0025 / NEW TASK ASSIGNMENT

Algorithm	# Interruptions	Avg. cycle time	Avg. standstill time	Improvement cycle time	Improvement standstill time
Default (Random)	25	12.39	1.20		
MCTS_{500}	22	10.55	1.16	1.84 (14.85 %)	0.04 (3.3 %)
MCTS_{1000}	21	10.60	1.14	1.79 (14.45 %)	0.06 (5.0 %)



(a) Least time efficient (13.00 s)

(b) Less time efficient(11.30 s)

(c) Best time efficient (8.81 s) and least time efficient with interruption (69.26 s)

Fig. 5. Conflicts in tasks planning by rare interruptions

Moreover, our approach also is able to handle interruptions at run-time by choosing among different (provably correct) multi-robot task scheduling strategies that also consider task re-allocation. The related works addressing the problem of robot task and motion planning with camera-based approaches [14]–[21] does not offer this capability: run-time task re-scheduling and re-allocation.

Our approach, during the reactive system specification phase aims to eliminate potentials collisions beforehand. We propose a solution which addresses the problem of unforeseen events, which may happen at runtime and offer more flexibility for robot cell planing and programming since there is no fixed robot tasks planing.

X. CONCLUSION AND FUTURE WORK

We presented an approach for robot tasks planning in multi-robot cells based on MCTS and GR(1) synthesis. The integration of MCTS aims to optimize the task planning at run-time. Whereas GR(1) permits a task planning with the abstraction of movement times and the task re-allocation in case of interruptions. We integrate our approach with a multi-robot cell and experiment our method to compare it with a default or random based method in a simulation environment.

Our methodology can be used for other application domains. We showed that it is possible to make task planning of a multi-robot cell beforehand and handle unforeseen events which can not be anticipated during the robot programming phase. Furthermore, MCTS makes optimal run-time planning since it determines the best choices for the reactive system based on unforeseen events model. We showed that controller synthesis techniques and MCTS techniques can be combined successfully and they complement each other excellently. With our experiments we showed that integrating MCTS-based method reduces up to 5% idle times and disturbance ultimately. Moreover our approach can significantly up to 14.9% optimize the cycle time.

In our future work, we want to include online optimization methods in the domain of reinforcement learning such as Q-Learning. In addition, we want to overcome the issue on the scalability of SPECTRA specifications and investigate on improvements of MCTS to address the topic of rare events to improve our MCTS-based approach (e.g. parallelization). Another topic for the future is to automatically and formally analyse the requirements and relevant information of the multi-robot cell and generate the corresponding system specification, in which a non-realizable specification highlights the requirements at the level of the user and not in the GR(1)-Specification. We may offer a DSL to robot engineers, so they will not have to learn temporal logics. In addition, our approach can be extended to other optimization goals, e.g. energy-efficiency.

Acknowledgement: We thank Shahar Maoz and Ilia Shevrin for their help with using and integrating SPECTRA.

REFERENCES

[1] J. Greenyer, D. Gritzner, T. Gutjahr, F. König, N. Glade, A. Marron, and G. Katz, “Scenariotools – a tool suite for the scenario-based modeling and analysis of reactive systems,” *Science of Computer Programming*, vol. 149, pp. 15 – 27, 2017, special Issue on MODELS’16. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167642317301442

[2] J. Greenyer, L. Chazette, D. Gritzner, and E. Wete, “A scenario-based MDE process for dynamic topology collaborative reactive systems - early virtual prototyping of car-to-x system specifications,” in *Joint Proceedings of the Workshops at Modellierung 2018 co-located with Modellierung 2018, Braunschweig, Germany, February 21, 2018.*, 2018, pp. 111–120. [Online]. Available: http://ceur-ws.org/Vol-2060/mekes8.pdf

[3] N. Piterman, A. Pnueli, and Y. Sa'ar, “Synthesis of reactive(1) designs,” in *Verification, Model Checking, and Abstract Interpretation*, E. A. Emerson and K. S. Namjoshi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 364–380.

[4] S. Maoz and J. O. Ringert, “On the software engineering challenges of applying reactive synthesis to robotics,” in *Proceedings of the 1st International Workshop on Robotics Software Engineering, RoSE@ICSE 2018, Gothenburg, Sweden, May 28, 2018*, F. Ciccozzi, D. D. Ruscio, I. Malavolta, P. Pelliccione, and A. Wortmann, Eds. ACM, 2018, pp. 17–22. [Online]. Available: https://doi.org/10.1145/3196558.3196561

[5] ——, “Spectra: a specification language for reactive systems,” *Software and Systems Modeling*, Apr 2021. [Online]. Available: https://doi.org/10.1007/s10270-021-00868-z

[6] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, “Mastering the game of Go with deep neural networks and tree search,” *Nature*, vol. 529, no. 7587, pp. 484–489, Jan. 2016.

[7] A. Camacho, J. A. Baier, C. J. Muise, and S. A. McIlraith, “Bridging the gap between LTL synthesis and automated planning,” in *Workshop on Generalized Planning (GenPlan'17)*, 2017. [Online]. Available: http://www.cs.toronto.edu/~sheila/publications/cam-etal-genplan17.pdf

[8] N. D'Ippolito, N. Rodríguez, and S. Sardina, “Fully observable non-deterministic planning as assumption-based reactive synthesis,” *Journal of Artificial Intelligence Research*, vol. 61, pp. 593–621, 2018.

[9] K. Y. Rozier, “Linear temporal logic symbolic model checking,” *Computer Science Review*, vol. 5, no. 2, pp. 163 – 203, 2011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1574013710000407

[10] R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Sa'ar, “Synthesis of reactive(1) designs,” *Journal of Computer and System Sciences*, vol. 78, no. 3, pp. 911 – 938, 2012, in Commemoration of Amir Pnueli. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0022000011000869

[11] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, “A survey of monte carlo tree search methods,” *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 1–43, March 2012.

[12] G. Chaslot, J.-T. Saito, B. Bouzy, J. Uiterwijk, and H. van den Herik, “Monte-carlo strategies for computer go,” in *BNAIC'06*, P. Schobbens, W. Vanhoof, and G. Schwanen, Eds., vol. 18. University of Namur, 2006, pp. 83–90, pagination: 8.

[13] L. Kocsis and C. Szepesvári, “Bandit based monte-carlo planning,” in *Proceedings of the Seventeenth European Conference on Machine Learning (ECML 2006)*, ser. Lecture Notes in Computer Science, J. Fürnkranz, T. Scheffer, and M. Spiliopoulou, Eds., vol. 4212. Berlin/Heidelberg, Germany: Springer, 2006, pp. 282–293. [Online]. Available: http://www.sztaki.hu/~szcsaba/papers/ecml06.pdf

[14] S. Murray, W. Floyd-Jones, Y. Qi, D. Sorin, and G. Konidaris, “Robot motion planning on a chip,” in *Robotics: Science and Systems*, 06 2016.

[15] D. Sorin and G. Konidaris. (2018) Enabling faster, more capable robots with real-time motion planning. [Online]. Available: https://spectrum.ieee.org/automaton/robotics/robotics-software/enabling-faster-more-capable-robots-with-real-time-motion-planning

[16] M. Hanheide, M. Göbelbecker, G. S. Horn, A. Pronobis, K. Sjöö, A. Aydemir, P. Jensfelt, C. Gretton, R. Dearden, M. Janicek, H. Zender, G.-J. Kruijff, N. Hawes, and J. L. Wyatt, “Robot task planning and explanation in open and uncertain worlds,” *Artificial Intelligence*, vol. 247, pp. 119 – 150, 2017, special Issue on AI and Robotics. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S000437021500123X

[17] R. Meyes, H. Tercan, S. Roggendorf, T. Thiele, C. Büscher, M. Obdenbusch, C. Brecher, S. Jeschke, and T. Meisen, “Motion planning for industrial robots using reinforcement learning,” *Procedia CIRP*, vol. 63, pp. 107 – 112, 2017, manufacturing Systems 4.0 – Proceedings of the 50th CIRP Conference on Manufacturing Systems. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S221282711730241X

[18] A. H. Qureshi, A. Simeonov, M. J. Bency, and M. C. Yip, “Motion planning networks,” in *2019 International Conference on Robotics and Automation (ICRA)*. IEEE, 2019, pp. 2118–2124.

[19] X. Xu, Y. Hu, J. Zhai, L. Li, and P. Guo, “A novel non-collision trajectory planning algorithm based on velocity potential field for robotic manipulator,” *International Journal of Advanced Robotic Systems*, vol. 15, p. 172988141878707, 07 2018.

[20] M. Nazarahari, E. Khanmirza, and S. Doostie, “Multi-objective multi-robot path planning in continuous environment using an enhanced genetic algorithm,” *Expert Systems with Applications*, vol. 115, pp. 106 – 120, 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0957417418305165

[21] O. Arslan, K. Berntorp, and P. Tsiotras, “Sampling-based algorithms for optimal motion planning using closed-loop prediction,” in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, May 2017, pp. 4991–4996.