Agile Tasking of Robotic Systems with Explicit Autonomy*†

Rafael C. Cardoso[1], John L. Michaloski[2], Craig Schlenoff[2], Angelo Ferrando[3], Louise A. Dennis[1], Michael Fisher[1]
[1] The University of Manchester, Manchester M13 9PL, UK
{rafael.cardoso, louise.dennis, michael.fisher}@manchester.ac.uk
[2] National Institute of Standards and Technology (NIST), Gaithersburg, MD 20899, USA
{john.michaloski, craig.schlenoff}@nist.gov
[3] University of Genova, Genova 16145, Italy
angelo.ferrando@dibris.unige.it

Abstract

Task agility is an increasingly desirable feature for robots in application domains such as manufacturing. The Canonical Robot Command Language (CRCL) is a lightweight information model built for agile tasking of robotic systems. CRCL replaces the underlying complex proprietary robot programming interface with a standard interface. In this paper, we exchange the automated planning component that CRCL used in the past for a rational agent in the GWENDOLEN agent programming language, thus providing greater possibilities for formal verification and explicit autonomy. We evaluate our approach by performing agile tasking in a kitting case study.

Introduction

With the advent of Industry 4.0, new industrial standards have emerged for distributed and intelligent systems. At the same time, sensors and processing capability are gradually becoming cheaper. These factors have contributed to a significant growth in the adoption of robotic systems in domains such as manufacturing (Antzoulatos et al. 2017). In such systems, it is crucial for robots to be quickly tasked to perform an operation. Task agility is not only limited to the speed of tasking robots, but also includes other features such as handling task failure, planning for new goals, interchangeability of data and task plans between different robots, and adapting to dynamic environments (Harrison, Downs, and Schlenoff 2018).

To tackle the problem of agility performance across many robotic systems, the Canonical Robot Command Language (CRCL) (Proctor et al. 2016) was developed at the National Institute of Standards and Technology (NIST) in the USA. CRCL is a messaging specification that focuses on the domain of industrial robotics as a special case of an overall ontology for automation (Fiorini et al. 2017) and was chosen as the first test case to validate the concept of a core ontology with domain-specific extensions. It works as an information model for tasks that uses XML Schema to abstract semantic explanations of the behaviour of common robotic commands and their execution status. These commands are independent of the kinematics used by the robot and can be reused in robots from different manufacturers.

Interoperability of data between robots is also explored by middleware such as the Robot Operating System (ROS)[1]. The main difference between CRCL and ROS is that CRCL is a communication standard for both joint and Cartesian control of industrial robots, while ROS, is a framework providing a joint message interface. It is also possible to use CRCL with ROS using the packages available from the ROS-Industrial Consortium, however, in this paper we focus on the interaction of CRCL with GWENDOLEN.

In the agility demonstration at NIST (Kootbally et al. 2018), CRCL has been implemented for a 7 DOF Motoman SIA20D and a 6 DOF spherical wrist Fanuc LR Mate 200iD industrial robots. CRCL is a lightweight, modular component where messages provide robot motion and control with grasping as an abstraction on top of the proprietary robot controller interface. Significantly to manufacturing, these industrial robot controllers provide high-speed, smooth motion, and robust operation tuned to the proprietary robot architecture while optimising cycle time for manufacturing applications. Research partners, such as Georgia Tech and the University of Maryland in conjunction with the Advanced Robotics Manufacturing Institute (ARM), as well as industrial partners, such as Siemens, use or are exploring CRCL for industrial robot applications.

Classical task planning is an effective technique for generating plans to achieve a goal or a set of goals in complex problems (Nau, Ghallab, and Traverso 2004). The Planning Domain Definition Language (PDDL) (McDermott et al. 1998) is a well-known formalism for representing planning domains and problems and it is often the default input language for many automated planners.

*Work supported by UK Research and Innovation, and EPSRC Hubs for "Robotics and AI in Hazardous Environments": EP/R026092 (FAIR-SPACE), EP/R026173 (ORCA), and EP/R026084 (RAIN). Fisher's work also supported by Royal Academy of Engineering.

†Commercial equipment and software, many of which are either registered or trademarked, are identified in order to adequately specify certain procedures. In no case does such identification imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the materials or equipment identified are necessarily the best available for the purpose.

Copyright © 2021by the authors. All rights reserved.

[1] https://www.ros.org/

CRCL has used a subset of PDDL in the past to perform automated task planning (Proctor et al. 2016; Balakirsky 2015). In their work, the PDDL output plan is translated into CRCL commands that are then sent for execution to the robot controller. If something fails, then the current status of the execution can be translated back in order to update the problem definition and attempt to replan.

In this paper, we propose the use of the GWENDOLEN (Dennis 2017) agent programming language to reason about and execute tasks with CRCL, instead of the PDDL component that was integrated previously. One advantage of using rational agents over classical planning with PDDL is that failure handling is achieved at the execution level. Assuming that the failure handling plans are correct, then the agent should be able to adapt to the failure using its current plans and without resorting to full replanning. Furthermore, the explicit intentions in such agent languages and the possibility of formal verification in GWENDOLEN both provide benefits for future understanding and trust.

Related Work

The ROSPlan (Cashmore et al. 2015) framework brings classical task planning to ROS by embedding a PDDL task planner into robotic systems and linking it to existing ROS components. Their framework is composed of five main elements: the knowledge base contains the PDDL domain specification; the problem interface can generate the PDDL problem and publish it to a ROS topic; the planner interface calls the planner and publishes the solution to a topic; the parsing interface converts PDDL plans into ROS messages; and the plan dispatch sends the plan for execution. Even though ROSPlan can cope with action failures by calling the planner again to replan, we can save time by using a rational agent that can react to the failure by triggering the appropriate plans for failure handling.

An interface between the GWENDOLEN agent programming language and ROS has been developed to allow GWENDOLEN agents to publish and subscribe to ROS topics (Cardoso et al. 2020). Their interface is implemented in Java, and communicates with ROS through the *rosbridge* library; a library that offers communication via JSON messages to external code. The main advantage in their approach is twofold: first, publishing and subscribing to ROS nodes is completely transparent to the agent, as everything is dealt with at the interface level; and second, the agent code can be used with any version of ROS that has rosbridge. If we were to use CRCL with ROS, we would still need our GWENDOLEN integration to send commands through CRCL.

There are also several approaches (de Silva, Sardina, and Padgham 2009; Meneguzzi and Luck 2013; Colledanchise and Ögren 2017; Cardoso and Bordini 2019; Patra et al. 2020) that aim to integrate automated planning with rational agents. Such online planning can complement the reactive behaviour obtained from using agents and could be used to patch existing plans or create new plans at runtime. The main issue in using these approaches in practice is that their implementation is either domain specific or missing entirely. Such approaches would also have an impact in the verification of the agent programs, as new or modified plans can potentially violate existing properties.

The Canonical Robot Command Language

CRCL contains separate XML information models related to robot motion control and status reporting as well as underlying data types such as poses, speeds, and units. CRCL handles Cartesian and joint robot motion control, as well as parallel and vacuum gripper control, which allows it to target industrial robot applications. Because CRCL supports Cartesian motion control it is well-suited to handle pick and place robot tasks. CRCL requires an underlying robot controller to handle real-time motion trajectory planning.

CRCL addresses the myriad of Cartesian and joint level communication schemes inherent in proprietary commercial off the shelf robots. CRCL is a standard focused on the communication wire to the robot. The specification and software tools for CRCL are available from the ROS-Industrial Consortium. While CRCL is sufficient for many assembly and kitting tasks, extensions are needed in order to handle a wider variety of robot applications.

The standard CRCL interface insures software modularity between the application task software and the physical robot. The CRCL modularity should allow different commercial robots to be interchanged as long as a CRCL adaptor is available to communicate to the new robot and the new robots' functional specification is equivalent to the existing robot, such as for work volume, payload, etc. Although CRCL implementations are not pervasive yet, a single point of standardisation greatly assists in achieving modularity and avoiding the silo architecture common in industrial robotic applications.

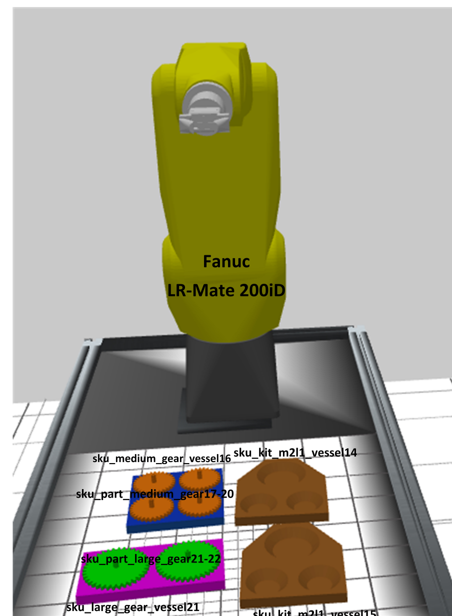

Figure 1: Example kitting models reported by CRCL.

Currently, CRCL does not include any sensor model re-

porting. For our purposes, we extend CRCL status XSD to include reporting of logical model information, similar to what would be returned by a camera. The augmented CRCL status model now reports all scene models name and pose. Such data can then be sent to the agent at runtime in order for it to maintain correct and up-to-date information about the environment. We further augment CRCL status by including properties that can be reported about each model. Using the CRCL status model properties, we are able to report inferences about the given scene. Figure 1 shows a planning scene that can be reported by a CRCL model status. In Figure 1 the model name is the type name appended with a numeric identifier within the scene.

Rational Agent Programming in GWENDOLEN

The GWENDOLEN agent programming language (Dennis 2017) is based on the Belief-Desire-Intention (BDI) model (Rao and Georgeff 1995): *beliefs* represent the knowledge that the agent has about the world, including itself and other agents; *desires* are the goals that the agent wants to achieve; and *intentions* are courses of action that can lead to the achievement of goals. For example, consider the GWENDOLEN syntax of a plan for sending a grasping command to grab an item with a robotic arm as shown in Listing 1.

```
+!take_part(Size) : { ~B gripper_grasped(_) }
                  <- take_part(Size);
```

Listing 1: GWENDOLEN Example Plan.

The plan will be triggered upon the addition (+) of the goal (!) take_part (Size). *Size* represents the size of the item to be grabbed and can be either a free variable or a unified term (following Prolog conventions, variables start with capital letters). After the colon and in between the curly brackets we have the guard of the plan. The plan will only be selected for execution if the guard is satisfied. In this case, there must not (∼) be a belief called gripper_grasped (_), where _ indicates variables which may match any value. Finally, the body of the plan is preceded by the left arrow and contains a sequence of steps (actions, belief operations, goal operations, etc.). In this example, the body includes the action take_part (Size) that will be sent to the robot. The semicolon at the end indicates the end of the body of the plan.

Verification of autonomous robots can contribute to the trustworthiness of the system and can be vital in application domains such as safety-critical scenarios (Farrell, Luckcuck, and Fisher 2018). What makes GWENDOLEN different from other agent programming languages is that it was built from the ground up with *formal* verification in mind. The language comes equipped with the Agent Java PathFinder (AJPF) (Dennis et al. 2012) model checker. Model checking (Clarke et al. 2018) is a formal technique that verifies properties (usually specified using some form of temporal logic) of a system by exhaustively exploring the state space from a model (an abstraction of the implementation that can be represented, for example, as finite-state machine). AJPF verifies the agent program directly, instead of relying in an abstract model.

GWENDOLEN-CRCL Integration

In our integration, the GWENDOLEN agent selects plans for execution that are then decomposed into CRCL commands to be given to the robot. The agent is a CRCL client that can send messages to the CRCL servers' front-end, which in our case is a robot, and then monitors execution status until a program step has completed. Any failure is reported to the client, which can then trigger failure handling plans in the agent and result in a modified course of action. Figure 2 illustrates the relationship between a Gwendolen Agent and CRCL communication to command and control either a real or virtual instance of the NIST Agility Laboratory (Piliptchak et al. 2019).

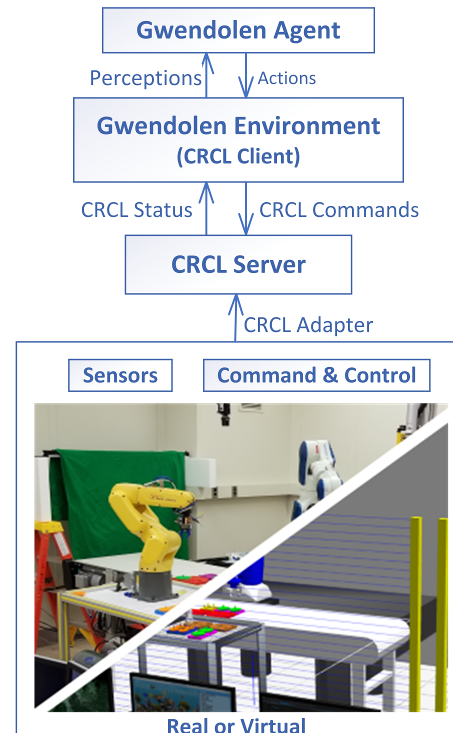

Figure 2: GWENDOLEN agent with CRCL communication to real/virtual world.

The environment in agent languages provides a bridge between the agent and the world. In an environment, the developer can implement the details of the agent's actions (either simulating or sending them to another node for execution) as well as implementing perceptions (either simulated or originated from another source) that will be sent to the agent. In this paper, the CRCL client is implemented as a GWENDOLEN environment. Communicating with CRCL is straightforward since GWENDOLEN environments are implemented in Java and there is a Java version of CRCL.

Information about the environment, such as sensor data and action results, is sent by the CRCL server to the CRCL client (in this case, a GWENDOLEN environment) in the form of CRCL status messages. By default, all data sent to the GWENDOLEN client is converted into perceptions and sent

to the agent. It is also possible to customise this function to revise the information coming in, for example, by applying a filter to send only information that is deemed relevant to the agent. Such customisation is domain dependent and should be added manually if the application requires it.

The high-level decision making of the robot is performed by the GWENDOLEN agent. This involves reasoning about events from the environment using a plan library containing pre-built plans for achieving tasks in a particular domain. Once the agent has decided upon the next course of action, the action is sent to the GWENDOLEN environment to be translated into a CRCL command and then sent to the CRCL server. Upon reception of the message containing an action to be executed, CRCL translates the action into a low-level command ready to be executed by the robot.

After the agent sends an action for execution, it waits for the result of that action before continuing with its current plan. This is achieved in GWENDOLEN by adding **action_result(Result)* to the step in the plan after said action, where * represents wait until *action_result(Result)* is true before continuing, and *Result* is a Boolean value indicating success (true) or failure (false). The wait in GWENDOLEN effectively suspends the intention related to the plan that it is a part of until its condition is satisfied. This information comes in from the CRCL server, which monitors the execution status of commands and reports it back to the agent. In case of an action failure, the appropriate failure handling plan for the action is triggered, which by default can be as simple as trying to execute the action again until it eventually succeeds, or more intricate to contain domain-specific information (such as attempting a different course of action).

Case Study: Kitting

Kit building or *kitting* is a common manufacturing process in which individually separate but related items are grouped, packaged, and supplied together as one unit (kit). It provides an industrial application for autonomous robots with planning, control and sensing challenges necessary to achieve task correctness as well as agility in dealing with errors.

Kitting is instructed in terms of task-level goals, such as "grasp part A and place it inside kit B" (Lozano-Perez et al. 1989). In our case study, different size gears (i.e., small, medium and large) are supplied in a supply vessel tray and empty kit trays arrive with open slots to be filled by a combination of sized gears. To perform kitting, either a small, medium or large gear is first grasped and then placed into a corresponding open slot of that size in a kit.

Planning involves studying relationship of the gears and trays where both kitting and supply tray models provide the occupancy of the contained slots. Tasks in this case study require filling a kit with the variety of gear sizes from the supply trays by tracking the tray slots state as either occupied by a gear or open with no gear.

For advanced reasoning, the necessary physical definition of all the types of objects is assumed to be provided. For example, Figure 3 shows the physical relationships exhibited by a kitting tray and gears. For kitting, we assume these definitions for the different size gears, each supply or kitting tray, as well as all slots within the trays have been provided.

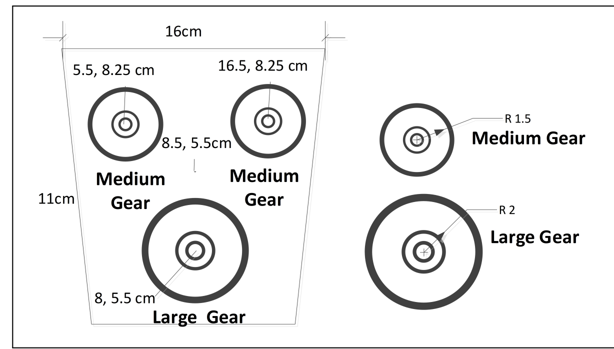

Figure 3: Kit tray and gears.

For each kit, supply vessel, and gear, the CRCL server produces knowledge inferences by studying the relationship of the gears and trays in the scene. For example, an inference could be established based on whether a tray slot is "open" or contains a gear with a model name (e.g., "sku_medium_gear_19"). This inference is based on the proximity of a gear to the position of a slot with a tray. If no gear in the scene is found in close proximity to the slot, it is "open".

With augmented CRCL model status reporting, the GWENDOLEN agent can reason about which decision to make using up-to-date information about the world. The basic kitting strategy used by the agent is as follows:

1. Find an open slot in a kit tray;
2. Search its belief base for a matching free gear of the same size as the open slot;
3. Move the arm to a position that would allow the gripper to grasp the gear;
4. Grasp the gear;
5. Move the arm to the open slot;
6. Release the grasp.

Now we discuss the tools used to implement this case study in a 3D simulation[2]. We use the *Fanuc LR Mate 200iD* robotic arm to perform kitting operations. This model has 6 axis, a reach of 717 mm, and load capacity of 7 kg. Gazebo with Open Dynamics Engine (ODE) physics engine provides the simulation tool for physics-based interaction of robots, kitting, and environment. Wire mesh models and physical properties of the NIST agility lab robots and kitting objects were incorporated into the Gazebo world. Modelling the physical properties of kitting elements in a physics engine can be challenging. For example, in order to properly use the simulated physics engine functionality, the weight, size and inertia of robot and kitting objects must be properly specified. The benefit is that when robots or kitting objects collide with each other, the object interaction is observable and if unintentional, adverse consequences are apparent.

At the top level, the agent is implemented in the GWENDOLEN agent programming language. The CRCL client is

[2] `https://github.com/autonomy-and-verification/gwendolen-crcl-kitting`

written as a GWENDOLEN environment (i.e., Java) and acts as a bridge between the agent and the robot. The Java version of the CRCL server is used to convert actions that are sent by the agent into CRCL commands that are communicated to the robot via a CRCL adaptor module.

The CRCL adaptor module uses CodeSynthesis[3] and the Xerces XML DOM parser[4] for translating CRCL XML messages into a C++ representation. Given a C++ representation, conversion to ROS or other robot platforms can be generated. The lower level software architecture relies on Gazebo for 3D physics-based simulation.

The GWENDOLEN agent uses the following beliefs that are updated by CRCL to match the current state of the world:

- $gear_tray(IdGearTray, Size, [Slot_1, \ldots, Slot_n])$: *IdGearTray* is the identifier name for the tray, *Size* is either *"small"*, *"medium"* or *"large"*, and for each slot in the list of slots $[Slot_1, \ldots, Slot_n]$ the value is assigned to either the identifier name for the gear in that slot or *"empty"*;
- $kit_tray(IdKitTray, [slot(Id_1, Size_1, Pos_1), \ldots, slot (Id_n, Size_n, Pos_n)])$: *IdKitTray* is the identifier name for the tray, and each slot predicate from the list of slots contains the identifier name for the slot in the first parameter (required for the agent to know where to move the arm), the size of the slot in the second parameter, and the identifier name for the gear in that slot or *"empty"* in the third parameter;
- $gripper(State)$: *State* represents the current grasping mode of the gripper, either *"open"* or *"closed"*;
- $grasped(Object)$: *Object* is the identifier name of the gear that is currently grasped by the gripper.

In Table 1, we show the high-level actions that the agent can decide to do, and their translation into low-level CRCL commands. Actions *open_gripper* and *close_gripper* are straightforward to translate. The agent should not have to worry about low-level path planning or pose information; it sends actions to move the arm to a particular location, which can be a kit tray, a gear tray, a particular gear in a gear tray, or a particular slot in a kit tray. The translation into low-level requires prior mapping of the coordinates of each possible location. While some of these positions may change at runtime (e.g., a gear is moved), we rely on CRCL to keep track of the new coordinates.

GWENDOLEN action	**CRCL command**
open_gripper	setGripper(1.0)
close_gripper	setGripper(0.0)
move(Location)	moveTo(Pose)

Table 1: Actions in GWENDOLEN and their respective counterparts in CRCL commands.

The main plan for performing a kitting operation is shown in Listing 2. This plan is triggered upon the detection that there is a kit tray *IdKitTray* with an empty slot Id for gears of size $Size$. The guard of the plan checks that the gripper is not currently grasping an object, and then consults the belief base for a gear tray with gears of $Size$. The first step in the body of the plan is an internal action that looks through a list of slots in the gear tray and then adds a perception containing the first non-empty position[5]. Next, the agent adds the relevant goals for each of the main steps in kitting. Each of these goal additions will trigger another plan, with its own guard and body.

```
+!kitting(Id,Size) :
  { ~B grasped(_), B gear_tray(IdGearTray,Size,Slots) }
  ← find_gear(Slots), *gear(Gear), +!move(Gear),
     +!close_gripper, +!move(Id), +!open_gripper;
```

Listing 2: GWENDOLEN agent plan for kitting.

As an example of action failure in this scenario, let us consider the failure of the action to open the gripper's grasp. In this setting, a failure is detected by CRCL if it receives an action to open the gripper while it is currently grasping a gear and the target position for releasing the grasp is either not an empty position of a kit tray or the size of the empty position does not match the size of the gear currently grasped.

For brevity, we only show the plan for handling the failure of trying to place a gear into an open slot of different size, shown in Listing 3. Since there are other plans that handle failures of the *open_grip* action, we need to ensure that the agent will select the correct plan. This is done by testing if the sizes (received from CRCL) do not match in the guard of the plan. To solve this failure, first the agent looks through its updated belief base for a new open slot of the appropriate size, moves to it, and then releases the grasp.

```
+!action_result(Size,Gear,SizeGear,false) :
  { ~ (Size == SizeGear) }
  ← +!find_open_slot(SizeGear),
     *new_target(NewIdKitTray,NewId), +!move(NewId),
     +!open_gripper;
```

Listing 3: One of the plans in GWENDOLEN for handling failure in the *open_gripper* action.

Another example of an error scenario is the mishandling and dropping of a free gear during the pick and place of the gear in an open kitting slot. Such a case could occur if the gear was not grasped properly as it was askew in the supply tray and undetected, or by sliding out of the supply tray if the gear is misoriented in the tray and falls out of a moving conveyor. Quality control deems dropped parts no longer to specification. In such a scenario, if the robot is able to pick up the gear then it should place it in the recycle bin. It can be difficult for PDDL planners to cope with this type of error scenario, since they can not achieve the same level of reactivity as in rational agents without constantly recalling the planner to perform replanning.

[3] www.codesynthesis.com/products/xsd
[4] xerces.apache.org/xerces-c

[5] For brevity, we omit the plans that deal with the case where all positions in the tray are empty.

Conclusions

Our integration of rational agents, developed in the GWENDOLEN agent programming language, with the CRCL information model allows robotic solutions to be developed and deployed for application domains that require agile tasking of robots. We have demonstrated the applicability of our approach in a case study of a robotic arm autonomously assembling kits. The reactivity of rational agents makes them a suitable approach for handling action failures without having to resort to full replanning. Even though its effectiveness in responding to failures depends on pre-existing and well made plans, these plans can be formally verified to provide assurances that they will behave as expected.

For future work, we want to extend our integration to incorporate a number of ontologies that have been designed for robotics and automation (Fiorini et al. 2017). These ontologies could be used to standardise the information being exchanged between GWENDOLEN agents and CRCL. This would make our approach more general and easier to apply in different application domains. We also plan to experiment with more complex case studies, possibly involving multiple robots/agents, and explore the use of formal verification to provide assurances about existing plans, in particular the plans for failure handling.

References

Antzoulatos, N.; Castro, E.; de Silva, L.; Rocha, A. D.; Ratchev, S.; and Barata, J. 2017. A multi-agent framework for capability-based reconfiguration of industrial assembly systems. *International Journal of Production Research* 55(10):2950–2960.

Balakirsky, S. 2015. Ontology based action planning and verification for agile manufacturing. *Robotics and Computer-Integrated Manufacturing* 33:21 – 28. Special Issue on Knowledge Driven Robotics and Manufacturing.

Cardoso, R. C., and Bordini, R. H. 2019. Decentralised planning for multi-agent programming platforms. In *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems*, AAMAS '19, 799–807.

Cardoso, R. C.; Ferrando, A.; Dennis, L. A.; and Fisher, M. 2020. An interface for programming verifiable autonomous agents in ROS. In *European Conference on Multi-Agent Systems (EUMAS)*.

Cashmore, M.; Fox, M.; Long, D.; Magazzeni, D.; Ridder, B.; Carreraa, A.; Palomeras, N.; Hurtós, N.; and Carrerasa, M. 2015. ROSPlan: Planning in the robot operating system. In *Proceedings of the 25th International Conference on International Conference on Automated Planning and Scheduling*, ICAPS'15, 333–341. AAAI Press.

Clarke, E. M.; Grumberg, O.; Kroening, D.; Peled, D.; and Veith, H. 2018. *Model checking*. MIT press.

Colledanchise, M., and Ögren, P. 2017. How behavior trees modularize hybrid control systems and generalize sequential behavior compositions, the subsumption architecture, and decision trees. *IEEE Transactions on Robotics* 33(2):372–389.

de Silva, L.; Sardina, S.; and Padgham, L. 2009. First principles planning in BDI systems. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems - Volume 2*, AAMAS '09, 1105–1112.

Dennis, L. A.; Fisher, M.; Webster, M. P.; and Bordini, R. H. 2012. Model checking agent programming languages. *Automated Software Engineering* 19(1):5–63.

Dennis, L. A. 2017. Gwendolen semantics: 2017. Technical Report ULCS-17-001, University of Liverpool, Department of Computer Science.

Farrell, M.; Luckcuck, M.; and Fisher, M. 2018. Robotics and integrated formal methods: Necessity meets opportunity. In *Integrated Formal Methods*, volume 11023 of *LNCS*, 161–171. Springer.

Fiorini, S. R.; Bermejo-Alonso, J.; Gonçalves, P.; de Freitas, E. P.; Alarcos, A. O.; Olszewska, J. I.; Prestes, E.; Schlenoff, C.; Ragavan, S. V.; Redfield, S.; et al. 2017. A suite of ontologies for robotics and automation [industrial activities]. *IEEE Robotics & Automation Magazine* 24(1):8–11.

Harrison, W.; Downs, A.; and Schlenoff, C. 2018. The agile robotics for industrial automation competition. *AI Mag.* 39(4):73–76.

Kootbally, Z.; Schlenoff, C.; Antonishek, B.; Proctor, F.; Kramer, T.; Harrison, W.; Downs, A.; and Gupta, S. 2018. Enabling robot agility in manufacturing kitting applications. *Integrated Computer-Aided Engineering* 25(2):193–212.

Lozano-Perez, T.; Jones, J. L.; Mazer, E.; and O'Donnell, P. A. 1989. Task-level planning of pick-and-place robot motions. *Computer* 22(3):21–29.

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL - the planning domain definition language. Technical Report TR-98-003, Yale Center for Computational Vision and Control.

Meneguzzi, F., and Luck, M. 2013. Declarative planning in procedural agent architectures. *Expert Systems with Applications* 40(16):6508 – 6520.

Nau, D.; Ghallab, M.; and Traverso, P. 2004. *Automated Planning: Theory & Practice*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Patra, S.; Mason, J.; Kumar, A.; Ghallab, M.; Traverso, P.; and Nau, D. 2020. Integrating acting, planning, and learning in hierarchical operational models. In *Proceedings of the 30th International Conference on Automated Planning and Scheduling (ICAPS)*, 478–487. AAAI Press.

Piliptchak, P.; Aksu, M.; Proctor, F. M.; and Michaloski, J. L. 2019. Physics-based simulation of agile robotic systems. In *ASME International Mechanical Engineering Congress and Exposition*, volume 59384, V02BT02A011. American Society of Mechanical Engineers.

Proctor, F.; Balakirsky, S.; Kootbally, Z.; Kramer, T.; Schlenoff, C.; and Shackleford, W. 2016. The canonical robot command language (CRCL). *Industrial Robot: An International Journal* 43:495–502.

Rao, A. S., and Georgeff, M. P. 1995. BDI agents: From theory to practice. In *Proceedings of the first International Conference on Multi-Agent Systems*, 312–319.