Proceedings of The ASME 2021
International Mechanical Engineering Congress and Exposition
IMECE2021
November 1-4, 2021, Virtual, World

IMECE2021-73683

AGILE TASKING OF ROBOTIC KITTING

John Michaloski*, Murat Aksu and Craig Schlenoff
National Institute of Standards and Technology
Gaithersburg, MD, USA

Rafael C. Cardoso and Michael Fisher
The University of Manchester†
Manchester, UK

ABSTRACT

Manufacturers are looking for intelligent solutions to increase quality and productivity. Smart manufacturing envisions production empowered by autonomous robots that can complete tasks intelligently, with the focus on adaptability, flexibility, and versatility. In such systems, agile tasking plays an important role, as it is critical for robots to be quickly tasked to perform an operation. However, task agility is not 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. Because robot task agility requires sophisticated dynamic and continuous planning and replanning, the Gwendolen agent programming language was chosen to evaluate as the agile robot planner. In this paper, we develop a manufacturing kitting case study and provide a list of kitting performance metrics to evaluate performance. The case study uses Gwendolen, Canonical Robot Command Language (CRCL), Robot Operating System (ROS), and Gazebo software components in combination to simulate and evaluate kitting. We explore the strengths of Gwendolen agile tasking to assess the operation against the kitting performance metrics.

Keywords: robots, planning, agent, agility, simulation, kitting

*Contact author: john.michaloski@nist.gov

†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.

Nomenclature

API	Application Programming Interface
APRS	Agility Performance of Robotic Systems
BDI	Belief–desire–intention
CRCL	Canonical Robot Control Language
IEEE	Institute of Electrical and Electronics Engineers
KDL	Kinematics and Dynamics Library
KRL	Kuka Robot Language
NIST	National Institute of Standards and Technology
PDDL	Planning Domain Definition Language
ROS	Robot Operating System
ROS-I	ROS-Industrial
SUT	System Under Test
XJC	XML Java Compiler
XML	eXtensible Markup Language
XSD	XML Schema Definition

BACKGROUND

Agile manufacturing can be defined as the capability of surviving and prospering in a competitive environment of continuous and unpredictable change by reacting quickly and effectively [1, 2]. For manufacturing robots, the IEEE Robotics and Automation Standards Working Group on Robot Agility [3] provides a list of desirable traits of robotic systems under the umbrella of agility — a complex notion of reconfigurability and autonomy that contrasts to the more ubiquitous pre-programmed robot tasks. Aspects of robot agility include hardware reconfig-

urability, software reconfigurability, communications, task representation, sensing, perception, reasoning, planning, tasking, and execution [4]. Robot agility is by nature reactive involving a changing environment for its measurement [5].

To tackle the problem of agility across many different robotic systems, the Canonical Robot Command Language (CRCL) [6] was developed at the National Institute of Standards and Technology (NIST) in the USA. CRCL addresses the myriad of Cartesian and joint level communication schemes inherent in commercial off–the–shelf robots. For example, a recent sampling of commercial robots shows each with a different robot programming language such as Karel for Fanuc, INFORM for Motoman, RAPID for ABB, KRL for Kuka, VAL3 for Stäubli, and URScript for Universal Robots. To handle industrial applications, CRCL contains separate XML information models related to robot motion control and status reporting as well as underlying data types for 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 many industrial robot applications. Because CRCL supports Cartesian motion and gripper control, it is well-suited to handle pick and place robot agile planning operations.

CRCL is an interface to an underlying robot controller which must provide a real-time robot kinematic solver and handle motion trajectory planning. For our research purposes, CRCL relied on the open-source frameworks (Robot Operating System and Gazebo) for robot control and simulation. The Robot Operating System (ROS) is an open source software framework that provides libraries and tools to help create robot systems. For example, ROS can be used to build a customized solution or a turnkey solution by assembling the best open source components that are integrated into ROS. For example, the Fanuc robot model is used and defined in the ROS-Industrial (ROS-I) `Fanuc` package [7] but could easily be replaced with another ROS robot package.

Gazebo is an open-source 3D physics-based simulator that can be used to design a virtual industrial robot world. Simulation is typically just a graphical visualization of the robot sequence of operations. In the case of Gazebo, physics-based models of the robots, kits, and environment provide a higher-fidelity approximation to the real-world. For example, the placing of a "gear" into a slot holder in visual simulation could overlay two images at the bottom of the slot (the gear and holder) without consequence. However, in the case of Gazebo physics-based simulation, the gear would "bounce" out of the slot as it is physically impossible for two solid object to combine.

In agile manufacturing, it is crucial for robots to be rapidly and intelligently tasked to perform an operation. We define a task to mean achieving a goal by selecting a series of actions based on the given state of the robot controller and environment. Tasks can be discrete events (e.g., place package into box) or continuous activity (e.g., monitor the robot for safe human distancing). Tasks can also vary in other ways, including timescale, difficulty, and detail. Given a task goal, an agile task planner consults the definitions of actions, preconditions, and postconditions in the problem domain and determines a feasible sequence of actions to achieve the goal. Of interest in this paper is task representation and associated reasoning to handle and recover from various random challenging events. Handling problems with a higher degree of intelligence results in smarter agile software.

The task planning in this paper is done using the Gwendolen agent programming language as it provides composable agile planning while offering greater possibilities for formal verification and explicit autonomy than was used in the past [8, 9]. Gwendolen is a Belief–desire–intention (BDI) agent programming language [10,11]. Beliefs provide information on the likely state of the environment. Desires include information about the objectives to be accomplished including priorities or payoffs that are associated with the various current objectives. Intentions define the currently chosen course of action. Intuitively, BDI represents a world that the agent believes to be possible, desires to bring about, and intends to bring about, respectively. Correspondingly, agile task agents are viewed as being rational and acting in accordance with their beliefs and goals [12].

Gwendolen uses intentions to store the mechanism for achieving goals that generally include actions, belief updates, and the commitment to goals. Gwendolen then defines a plan with the syntax:

trigger : *guard* ← *body*

In a plan, the trigger event may match the top event of an intention. The guard is checked against the agent's state for plan applicability. A body contains the plan actions for execution. For example, the Gwendolen syntax for robot command pick and place plan is shown in Listing 1.

```
+! pick_and_place(Part, Destination) :
{ ~B grasped (_) }
<- +!move(Part), +!close_gripper,
+!move(Destination), +!open_gripper;
```

Listing 1: Gwendolen Pick n Place Listing

Gwendolen uses BDI notation where: '+' denotes the addition of a belief, '+!' denotes the addition of a goal, '{ }' encloses the guard (context/precondition) of the plan, '∼' denotes negation in this case of a belief (represented by B), and *'(_)'* denotes a universal variable (which can match with any value). Thus, the *pick_and_place* plan is triggered by a goal intention while providing a Part and Destination variable binding. The guard $\{\sim B\ grasped(\_)\}$ verifies that the gripper is not grasping any item before executing the plan body. Finally, the body performs a series of actions including a *move* to the *Part* location, *close_gripper* on the *Part*, *move* to the *Destination* location, and then an *open_gripper* freeing the *Part*.

To evaluate task agility, a case study using kitting is developed. Kitting is a method to feed parts to workstations in assembly lines, where the different parts required to assemble one unit of an end item, or the specific set of parts to be assembled at a workstation, are included in a kit which is then transferred to the assembly line [13]. In industrial assembly of manufactured products, kitting is often performed prior to final assembly. The reasons for implementing such systems usually involve parallelized assembly systems, product structures with many part numbers, quality assurance of the assembly, and high value components [14].

In order to assess agile tasking, performance metrics are used to judge differing systems and approaches to planning. For this paper, agility performance metrics for manufacturing kitting have been previously studied in Downs et al. [15] and were adapted for our purposes. Such performance metrics were derived from test methods that assess system agility giving quantitative and qualitative metrics for judging robot system efficiency and effectiveness. Although the performance metrics defined in [15] were for robot system agility, they also apply to judging task agility. In tandem, agile tasking with the Gwendolen agent-based planner is performed with the results to be judged against the kitting agility performance metrics.

In the following sections, a case study involving agile task programming for simulation of robot sensing, planning, and control of manufacturing kitting is presented. We start with a background on related work in agile task programming. Next, a case study is described presenting the software elements required to deploy a kitting simulation, including Gwendolen agile tasking planning, CRCL, ROS, and Gazebo software components. Then performance metrics for kitting are presented with assessment of Gwendolen planning against these metrics. Finally, a conclusion summarizes the results and future work to be explored.

RELATED WORK

The objective of this paper is to evaluate the Gwendolen agile agent-based planner as applied to robot kitting against a list of kitting performance metrics. This assessment includes the study of Gwendolen with regard to robot planning, sensing, and control necessary for the overall kitting operation. Although there are many examples of research on software agents and the use of BDI as an agile task planning strategy, this review of related work focuses on the application of BDI to robotics and considers the implication of performance metrics on performance. Of note, conventional BDI does not perform global plan optimization which may indeed be preferable to a composable belief system. For example, Optaplanner [16] is a lightweight, embeddable planning engine for Java programming that handles planning optimization problems that is being deployed in the NIST agility laboratory [9]. However, the composability of agent plans that handle agile tasking including error handling, recovery from errors, and adaptive planning is the primary focus of the research. For future work we can look into adding support for Gwendolen agents to call an external optimisation planner when/if necessary.

Classical task planning is exemplified by the Planning Domain Definition Language (PDDL) and has been deployed in the NIST agility laboratory [17]. Kit building applications were described with Web Ontology Language (OWL) [18] and then generating PDDL files from the models in order to generate a plan or replan. Another PDDL system is found with the ROSPlan [19] framework which embeds a PDDL task planner into robotic systems and links it to existing ROS components. Even though PDDL systems 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 [20]. Their interface is implemented in Java and communicates with ROS through the ROSBridge library [21]. 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. There are also several approaches [22–26] that aim to integrate automated planning with rational agents. Such online planning can complement the reactive behavior 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 no domain is specified. Such approaches would also have an impact in the verification of the agent programs, as new or modified plans can potentially violate existing properties. Finally, robot planning solutions that rely on ROS can connect to many commercial robots, but preclude the ability to leverage the real-time industrial robot controllers. Overall, advantages of Gwendolen planning agents include:

- failure handling since Gwendolen allows plans to be formally verified;
- Gwendolen verification can verify directly the agent programs that will run the decision-making code;
- the plans in Gwendolen are developed explicitly and the resultant decisions follow a practical reasoning architecture, such that, it is easier to explain how and why an agent came about to a decision (i.e., executed an action) as opposed to trying to figure out how a symbolic planner generated a plan.

Further, Gwendolen is a good fit for the agile kitting case study that will be evaluated mainly due to the reactive nature of agent-oriented programming. If an event happens during the execution that is not normal but has been pre-emptively identified (failures, decrease in performance, etc.), then this event will trig-

ger the appropriate plan to react to it. It is imperative for the decision-making software to be able to adapt on the fly to these situations in order to remain agile.

Upon first examination, formal verification may not appear to be as significant as in other planning cases due to the logical steps within the kitting case study world. However, in general, formal verification is highly beneficial in finding problems pertaining to the model of the system (or directly in the program) as well as finding counterexamples. In our approach, we will show the utility to provide formal guarantees about the plans that handle failures, since these are one of the main advantages of using Gwendolen in the kitting case study. To be deemed as agile, multiple layers of establishing correctness improve overall planning.

KITTING CASE STUDY

In manufacturing, kitting is a process in which individually separate but related items are grouped, packaged, and supplied together as one unit (kit). Kitting is a well-studied manufacturing problem. Kitting is a reasonably difficult robot task to test agile task planning. Kitting requires grasping, pick and place, and sorting/ordering of objects that can exhibit complexity with different grippers, gripper changes, dynamic supply from a conveyor, etc. However, our goal is to understand and examine agile tasking for basic kitting with one robot and one gripper manipulating simple gears, trays, and kits in order to measure agility based on performance metrics. In the case study, the kitting task is purposefully scoped to be understandable while clarifying the various elements required. The intent is to demonstrate and evaluate the Gwendolen agile task planner given a list of kitting performance metrics.

Architecture

The system is based on the NIST Agility Performance of Robotic Systems (APRS) agility laboratory that contains two industrial robots, a Fanuc LR-Mate 200iD and a Motoman SIA20F [27]. This case study focuses on Gwendolen agile tasking for the Fanuc LR-Mate 200iD. Figure 1 shows the architecture for simulated robot kitting. The kitting uses colored plastic test gears that are moved from their initial location in supply trays to a final placement in a kit tray's open slot. Supply trays contain gears of one type (small, medium, or large), while kits contain a combination of gear types. For simulation, instead of overhead cameras in the work volume, Gazebo reports on the model properties of gears, trays, and kits. The agile task case study system is composed of the following modules:

Gwendolen Planner — Gwendolen accepts kitting order and uses the logical world model as provided by the Model and Control Application Programming Interface (API) to reason about the kitting problem. Gwendolen selects a plan that produces kitting actions that are then translated into a

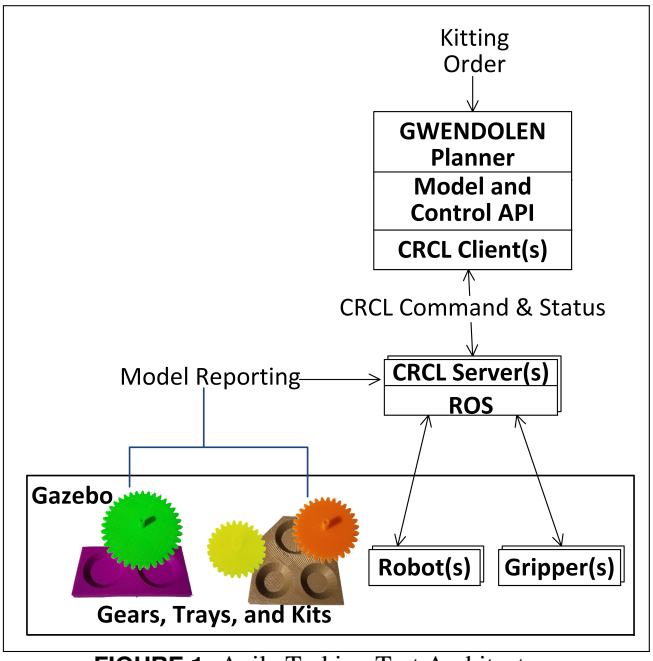

FIGURE 1: Agile Tasking Test Architecture

CRCL step to be transmitted by the CRCL client. Changes to the world model are reported through CRCL within the status report and echoed in the Model and Control API. These changes include physical properties and or logically derived properties from the physical properties. Any failures are reported by CRCL, which triggers replanning if possible.

Model and Control API — translates between a logical world model required for Gwendolen reasoning and a CRCL physical description of the world model. For each Gwendolen action, the Model and Control API maps variables from a logical description into the physical kitting object properties.

CRCL Client — receives the CRCL messages and sends the commands to a CRCL server and monitors reported status. The CRCL client simulator acts like a logical camera sensor but instead of extracting gear, kit, and tray model knowledge from a camera, it relies on a Gazebo plugin that reports on all models in the simulation world. To enable planning at a logical reasoning level about the kitting world, the existing CRCL status schema was extended to provide kitting model (gears, trays, kits, etc.) locations as well as inferences about the kitting models (e.g., a gear located in a supply tray slot).

Robot CRCL server — reads CRCL commands and provides CRCL status updates by forwarding knowledge to/from the CRCL robot interface and Gazebo simulation.

The CRCL server translates CRCL representation into ROS representation.

ROS — provides numerous robot control and sensing packages that were bundled using the ROS framework. Relevant to the Gwendolen agile planning simulation were the packages *moveit!* an open–source Cartesian and joint motion trajectory planner, the Kinematics and Dynamics Library (KDL) kinematic solver applied to the Fanuc LR-Mate 200id, the *tf* transform library for handling robot mathematics, and the ROS core to manage message communication between components. In addition, the *gazebo_ros_api* Gazebo plugin supplied by ROS were used to set or retrieve simulated robot positioning, as well as to retrieve kitting world model properties.

Gazebo — the agile kitting task simulation was done using Gazebo. Gazebo is an open-source 3D physics-based simulator that can be used to design a virtual robot world. In our simulation, Gazebo used the Open Dynamics Engine (ODE) physics engine to provide the physics-based interaction of robots, kitting, and environment. The simulated world provides a robust mechanism to validate and test system operation, and at the same time quantify performance using Gwendolen.

The highest-level task planning module is coded in Java. Gwendolen parses the BDI notation, and then compiles plans from BDI into concurrent Java code. Plans are mapped into threads which may block until a condition is satisfied. Each plan is executed every cycle. For our purpose, we used Java XML Java Compiler (XJC) to autogenerate Java classes corresponding to the CRCL XML Schema Definition (XSD) schema files. Using these Java classes, CRCL is parsed into Java representation. Likewise, XJC was used to extend CRCL XSD to enable improved status reporting of model properties as well as model inferences. The Java CRCL implementation also handles the serialization from Java into CRCL XML for transmittal to the CRCL server on the ROS side.

The ROS side is coded in C++ and uses CodeSynthesis to parse and serialize the CRCL XML. Once parsed, the CRCL is translated into ROS representation suitable for *moveit!* trajectory planning and kinematic solving. Joint positions are then updated to Gazebo using the *gazebo_ros_api* plugin.

Kitting World

At a high level of kitting operation, the robot is tasked with picking the appropriate gears from the supply trays, and then placing the gear into the kit. To keep the kitting scenario simple for discussion, we will limit the robot agility requirements and assume the following. First, the robot already has the correct part gripper and that all parts can use the same gripper. Second, we will assume the necessary combination of small, medium, and large part trays and kits will be available on the worktable. The small supply kits contain four small gears, the medium supply kit has four medium gears, and the large supply kit has two large gears. The kit of interest has storage for two medium gears and two large gears. Figure 2 illustrates the gears, trays, and kits used for kitting operation that exhibit a simple geometry, with a peg handle on top to make it possible to pick the part with relative ease. We assume supply trays are filled with the same gear as shown in Figure 2 where one medium supply tray contains up to four medium orange gears, and 1 large supply tray contains up to two green large gears. The goal is to fill the two kits to capacity using the robot to move a matching gear from the supply tray into an open slot in the kit. Despite the simplicity, the kitting task reduces to a pick and place problem with a myriad of problems and challenges.

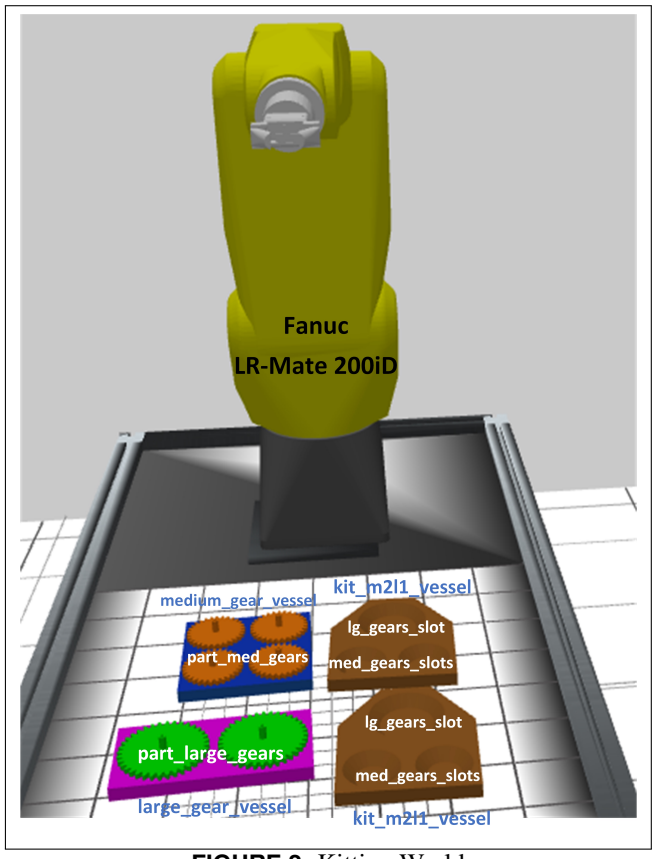

FIGURE 2: Kitting World

For our case study, we assume the robot has a two-finger gripper to grasp the gear peg and move from the supply tray to the kit. The goal of the kitting problem is to load the kit with two medium gears and one large gear. Clearly the problem can

Planner	Plan Model and Control API	CRCL
acquire_part	Beliefs 1) find "closest" empty kitting slot, find "closest" matching supply gear from supply tray 2) gripper empty	CRCL reports logical object properties and first order derived properties (slots and gears in trays) every cycle. Used here to determine open kitting slot and matching gear and location.
take_part	approach gear grasp gear retract	init (speeds, units) moveto pose (approach) moveto pose (grasping point) setgripper 1 (close) moveto pose (retract) end
place_part	approach open kit slot release gear retract	init(speeds, units) moveto pose (approach) moveto pose (slot position) setgripper 0 (open) moveto pose (retract) end

TABLE 1: Kit Planning Steps

exhibit a higher degree of complexity, with multiple robots, multiple end-effector types, tasking to include unloading trays and kits, to name a few potential task variants.

Table 1 shows the planning for kitting that includes the major plans for specifying the movement of objects from supply trays to assembly kits. The sequence of operations to be handled by the kitting planner depends on the constraints for the task. Four types of constraints are defined and examined: processing constraints (e.g., new or higher priority kitting order), feasibility constraints (e.g., sufficient parts in part tray to fill kit), completions constraints (for example dropped part or robot servo fault), and random fault constraints (for example dropped part or bad sensor/actuator.)

To fulfill a kitting task, first the robot must receive a kit order. Then the planning sequence outlined in Table 1 is undertaken.

First, the *acquire_part* plan is run where the goal is to find the closest open slot in a kit tray and a matching free gear of the same size in a supply tray. One of the beliefs that must be satisfied is that the gripper is not already holding a gear. For this planning step, CRCL provides logical status information used in determining kitting open slots, supply tray gears, and gear types. Next a *take_part* moves the robot arm to allow the gripper to grasp the gear and retract from the supply tray. This planning step translates the logical gear name into a sequence of CRCL commands to approach, move to, and retract from the gear physical pose location. Finally, the *place_part* moves the robot to place the gear in the open kitting slot. This planning step translates the kitting slot name into a sequence of CRCL commands to approach, move to, and retract from the kitting open slot physical pose location.

Gwendolen Agile Tasking

To enable higher-level plan reasoning, a logical kitting object model was developed. So, instead of using CRCL reporting Cartesian motion primitives such as physical location, objects in the kitting world were given logical names and properties while hiding lower level details. To enable this functionality, an abstract Plan Model and Control API was implemented between the planner and CRCL. This planning API layer required broadening the existing CRCL status reporting to include kitting object properties (object name, pose, location, type) for API reasoning. In addition broadening the existing CRCL status reporting included inferring gear and kitting tray slot properties that are not reported by CRCL but instead were derived from first order reasoning about the kitting objects. For example, each supply vessel has slots that may or may not contain a small/medium/large gear depending on the vessel type. By cycling through the gears, it can be established which gear slots contain a gear as well as its properties: name, type (should match vessel supply type), and state (open or contain a gear name). Likewise, a kit has slot properties similar to supply vessels, but can be a combination of small/medium/large gear slots.

As Gwendolen is an agent-based system, agents contain a set of plans that are selected if their guard is true and then executed according to their plan body. We explore using Gwendolen to

handle the kitting agile planning outlined previously. First, the main kitting algorithm is mapped into a Gwendolen plan. The Gwendolen plan for sending a grasping command to grab an item with a robotic arm is shown in Listing 2.

```
+! move_part(Id, Size) :
  { ~B grasped (_),
     B kitting_tray(IdGearTray, Size, Slots)
  }
  <- find_gear(Slots),
     *gear(Gear),
     +!move(Gear), +!close_gripper,
     +!move(Id), +!open_gripper;
```

Listing 2: Gwendolen Main Kitting Plan Listing

The Gwendolen plan is triggered upon the addition '+' of the goal '!'*move_part(Size)*. *Size* represents the size of the item to be grabbed and can be either a free variable or a unified term (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 *grasped(_)*, where '_' indicates variables which may match any potential gripped object. 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 multiple actions to pick and place that are sent to the robot. The semicolon at the end indicates the end of the body of the plan.

As an example of Gwendolen failure handling, let us consider the dropping of the grasped gear that initially did match the kitting tray open slot. In this setting, the grasp CRCL will use its underlying force control that causes both gripper fingers to touch, indicating the grasping of no part. Minimally, this will be reported as an incorrect size gear in the gripper. In Gwendolen the code to detect a 'dropped' gear by the gripper is given by Listing 3

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

Listing 3: Gwendolen Dropped Gear Plan Listing

First, the Gwendolen code *+! action_result(Size, Gear, SizeGear, false)* defines a plan with a trigger event that matches four parameters *Size, Gear, SizeGear,* and the *false* boolean state. Next, the guard of the plan { *(Size == SizeGear)* } triggers when the size of the kit tray slot is not equal to the size of the gear we are trying to place and the body of the *action_result* plan will be executed. The plan body follows '<−' where the first step *+!find_open_slot(SizeGear)* calls another subplan to look for a slot in a kit tray compatible with the size of the gear that we are currently grasping. The second step of the plan * *new_target(NewIdKitTray, NewId)* waits for the belief containing the new target location resulting from the execution of the above subplan. A timeout and failure would be handled by another plan. Assuming a matching gear is found from the previous step, the plan performs *+!move(NewId), +! open_gripper;* which calls a move command to move to the new slot, and then issues a command to open the gripper and release the gear within its grasp into the open kitting slot.

For brevity, we only show the plan for handling the failure of a belief perceived dropped part which corresponds to trying to place a gear into an open slot of different size. Since there are other plans that handle failures of the *open_gripper* 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. Most likely, this will not be the case and the plan will recover by attempting to find a matching gear, be it the dropped gear or another gear.

ANALYSIS

In the application of agile planning to robotics, the metrics of interest are performance and correctness. Thus, the goal of assessing performance is to achieve an accurate assessment of the performance and reliability of the system by defining appropriate metrics with the following properties [28].

- A metric should have the property that it is repeatable: if the methodology is used multiple times under identical conditions, the same sensor should result in the same sensor readings.
- The metrics must exhibit no bias for implementations of identical technology,
- The metrics must be useful to users and providers in understanding the performance they experience or provide.

Thus, in the application of agile planning to robotics, the metrics of interest are not for instance, programmer productivity metrics, software management metrics, algorithmic costing metrics, or software design metrics. However, a basic performance metric remains software reliability, which is a probability measurement that the software will perform its intended function for a specified interval under stated conditions [28].

Category	Metric Name	Gwendolen performance
Process Completion Metrics	Qualitative Task Level Success Fulfil Kit Orders Success Total Number of Attempts	Reliability of Gwendolen computation is an important consideration given the relative simplicity of the kitting scenario.
Time Metrics	Task Time Total Time Planning Time Insertion Time	Fuzzy logic (e.g., slow, medium, or fast) applied to speed as well as guarded moves (e.g., soft, medium, hard forces) would be preferred logical mechanism. Insertion time would refer to any mating of assembly parts.
Distance Metrics	Kitting Object Travel Distance Manipulator Distance	Use of embedded optimization while choosing least distance travelled between free kitting slot and supply tray gear. Note, this may not be globally optimal as it may suffer local minima.
Failure Metrics	Hardware Failures Safety Violations Capability violations Sensor blackout	Hardware faults could cause a hand-off from Gwendolen to a paired agent; however, these were not attempted. Safety violations such as humans in danger were not addressed. Finally, capability violations such as reach, payload, or speed violations were out of scope.
Manipulation Metrics	Self-collision Collision avoidance Part awareness	Awareness of Dropped Part within plan. Dropped Part Picked Up if part is reported within reach. Duplicate Part Picked Part default recovery. Obstacle avoidance would be lower level of motion control.
Part Metrics	Faulty part Missing part Bad orientation	Part orientation or part inspection were out of scope. Missing supply parts would be reported as error.
In Process Order Change Metrics	Acknowledge/Ignore Scrap Reuse/remove new kit parts	Interrupting and replanning a Gwendolen kitting order would require delivery of new kits and supply trays which were out of scope.

TABLE 2: Gwendolen Kitting Metrics

The agility handling of abnormal events is the major thrust of this paper but key aspects of agility include the detection, handling, and recovery of application errors. For example, kitting agility includes handling of hardware faults, dropped parts, missing parts, misoriented parts, or faulty parts. Table 2 outlines the kitting performance metrics that include both correctness, efficiency, and agility aspects. The table provides a general performance metric category, specific metric instances, and the Gwendolen performance in the agile tasking category.

In order to evaluate run-time agility, errors are necessary and for the simulation they were injected into the system at random times to monitor system responsiveness. This was done by adding another software process and pseudo-randomly instigating the errors. For the system under test (SUT), ignoring an error state in a robotics environment can be disastrous. Minimally, the SUT should recognize there is a problem, and gracefully halt operation until the problem is fixed. Further, there are degrees of agility in handling faults, such as, acknowledgement and full remediation and recovery from problem, replanning an alternative strategy, or understanding the problem and using a substitute backup plan.

Gwendolen is best suited for adapting to and recovering from kitting failures. Since Gwendolen plans are composable, many guards checking for error conditions were developed, often with alternative plans to recover from foreseeable errors. For Gwendolen evaluation, injecting actuator or encoder faults into the simulation were not evaluated as these should be handled at a lower level of control and reported. Likewise, efficiency of the Gwendolen planner that measured task and planning time, as well as optimizing robot speeds and distance traveled were not a priority but could be handled by fuzzy logic (e.g., slow, medium, or fast) applied to speed as well as the case for guarded moves

(e.g., soft, medium, hard forces). Assembly operations of mating of parts was considered out of scope. Safety violations such as humans in danger or capability violations such as reach, payload, or speed violations were not studied with Gwendolen but could be added as part of a Gwendolen verification of operation where applicable.

For optimization, Gwendolen planning would rely on embedded Model and Control API optimization which could implement a least distance traveled between free kitting slot and supply tray gear optimization criteria. Such lower level optimization may not be globally optimal. Since the scope of the kitting was limited to basic Gwendolen pick-and-place planning with two-finger grippers, acknowledging potential problems was deemed a sufficient level of testing functionality.

CONCLUSION

The ideal smart factory is equipped with machines that are intelligent and adaptable to change and disruption. The occurrence of an abnormal event should be routinely handled by a smart machine with no human intervention. In this paper, robot kitting is used to examine high-level agile task planning using the Gwendolen agent programming language. Gwendolen is designed to routinely handle and recover from abnormal but foreseen events. Gwendolen programming was highlighted to study its response to various kitting agility problems against a list of kitting performance metrics.

The use of Gwendolen programming language as an agile task in a manufacturing kitting case study was studied. Although the case study concentrated on the fundamental elements of kitting, the ability to augment the Gwendolen program with composable plans to handle potential errors was highlighted. Future work would leverage the agent-based nature of Gwendolen, as well as system verification. Potential work includes the integration of multi-agent shared error handling in an adaptive manner so that hardware faults would result in a companion robot to assist in completion of a task. Gwendolen verification of kitting offers many potential benefits not found in many agent-based planning systems, including robot capability to manipulate the kitting objects, payload constraints, as well as monitoring energy use for sustainability analysis.

DISCLAIMER

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.

REFERENCES

[1] Gunasekaran, A., 1999. "Agile manufacturing: a framework for research and development". *International journal of production economics,* ***62***(1-2), pp. 87–105.

[2] Cho, H., Jung, M., and Kim, M., 1996. "Enabling technologies of agile manufacturing and its related activities in Korea". *Computers & Industrial Engineering,* ***30***(3), pp. 323–334.

[3] IEEE. P2940 – IEEE Standard for Measuring Robot Agility. `https://standards.ieee.org/project/2940.html`. Accessed: 2021-4-6.

[4] Downs, A., Kootbally, Z., Harrison, W., Pilliptchak, P., Antonishek, B., Aksu, M., Schlenoff, C., and Gupta, S. K., 2021. "Assessing industrial robot agility through international competitions". *Robotics and Computer-Integrated Manufacturing,* ***70***, pp. 102–113.

[5] Downs, A., Harrison, W., and Schlenoff, C., 2017. "Using standards in a competition to measure and solve industrial robotics agility challenges". In International Manufacturing Science and Engineering Conference, Vol. 50749, American Society of Mechanical Engineers (ASME).

[6] 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***, 08, pp. 495–502.

[7] ROS Fanuc Package. `wiki.ros.org/fanuc`. Accessed: 1-April-2021.

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

[9] 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), pp. 193–212.

[10] Rao, A. S., Georgeff, M. P., et al., 1995. "BDI agents: From theory to practice.". In ICMAS, Vol. 95, pp. 312–319.

[11] Dennis, L. A., and Farwer, B., 2008. "Gwendolen: A BDI language for verifiable agents". In Logic and the Simulation of Interaction and Reasoning, AISB.

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

[13] Caputo, A. C., Pelagagge, P. M., and Salini, P., 2018. "Economic comparison of manual and automation-assisted kitting systems". *IFAC-PapersOnLine,* ***51***(11), pp. 1482–1487. 16th IFAC Symposium on Information Control Problems in Manufacturing INCOM 2018.

[14] Johansson, M. I., 1991. "Kitting systems for small size parts in manual assembly systems". *Production Research Approaching the 21st Century*, pp. 225–230.

[15] Downs, A., Harrison, W., and Schlenoff, C., 2016. "Test methods for robot agility in manufacturing". *Industrial Robot: An International Journal,* ***43***(5).

[16] Optaplanner user guide. `www.optaplanner.org`. Accessed: 1-April-2021.

[17] Kootbally, Z., Schlenoff, C., Lawler, C., Kramer, T., and Gupta, S. K., 2015. "Towards robust assembly with knowledge representation for the planning domain definition language (PDDL)". *Robotics and Computer-Integrated Manufacturing,* ***33***, pp. 42–55.

[18] World Wide Web Consortium, 2009. "OWL 2 Web Ontology Language Primer". In www.w3.org/TR/owl2-primer/.

[19] 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, AAAI Press, pp. 333–341.

[20] 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).

[21] Crick, C., Jay, G., Osentoski, S., Pitzer, B., and Jenkins, O. C., 2017. "ROSbridge: ROS for non-ROS users". In *Robotics Research*. Springer, pp. 493–504.

[22] 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, pp. 1105–1112.

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

[24] 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), pp. 372–389.

[25] 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, pp. 799–807.

[26] 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), AAAI Press, pp. 478–487.

[27] 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, Vol. 59384, American Society of Mechanical Engineers.

[28] Paxson, V., Almes, G., Mahdavi, J., and Mathis, M., 1998. RFC 2330: Framework for IP performance metrics. Tech. rep., The Internet Society.