NIST Technical Note 2222

Agile Robotic Planning with Gwendolen

John Michaloski
Craig Schlenoff
Rafael C. Cardoso
Michael Fisher

This publication is available free of charge from:
https://doi.org/10.6028/NIST.TN.2222

National Institute of
Standards and Technology
U.S. Department of Commerce

NIST Technical Note 2222

Agile Robotic Planning with Gwendolen

John Michaloski
Craig Schlenoff
Intelligent Systems Division
National Institute of Standards and Technology
Gaithersburg, MD, USA

Rafael Cardoso[1]
Department of Computing Science
University of Aberdeen
Aberdeen, UK

Michael Fisher[2]
Department of Computer Science
The University of Manchester
Manchester, UK

This publication is available free of charge from:
https://doi.org/10.6028/NIST.TN.2222

May 2022

U.S. Department of Commerce
Gina M. Raimondo, Secretary

National Institute of Standards and Technology
Laurie E. Locascio, NIST Director and Undersecretary of Commerce for Standards and Technology

[1]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).
[2]Work supported supported by Royal Academy of Engineering.

National Institute of Standards and Technology Technical Note 2222
Natl. Inst. Stand. Technol. Tech. Note 2222, 48 pages (May 2022)

This publication is available free of charge from: https://doi.org/10.6028/NIST.TN.2222

Abstract

The future of robotics foresees autonomous behavior that can complete tasks intelligently, with a focus on adaptability, flexibility, and versatility. In such systems, it is critical for robots to quickly and safely perform an operation. However, such aptitude is not limited to the speed of solving tasks, but also requires other qualities such as adeptly detecting and recovering from task irregularities, overcoming unforeseen task barriers by replanning to achieve stated goals, and adroitly adapting to dynamic environments such as changing light illumination, noisy sensors, or unexpected conditions. These intelligent characteristics define robot agility (not to be confused with robot agility akin to dexterity), and refer to approaches that allow robotic systems to be flexible and capable of re-tasking in the face of a changing and often unpredictable environment. Because robot task agility requires sophisticated dynamic and continuous planning and replanning, the GWENDOLEN intelligent agent programming language is studied as a high-level robot planner. In this report, we develop a manufacturing kitting case study to research the operation of GWENDOLEN planning. The case study uses the combination of GWENDOLEN, Canonical Robot Command Language (CRCL), Robot Operating System (ROS), and Gazebo software components to simulate and evaluate robot planning. Several Agile Robotics for Industrial Applications Competition (ARIAC) kitting agility challenges are used to evaluate GWENDOLEN planning under various levels of operational duress. Both the benefits and shortcomings will be reviewed.

Key words

robots; planning; agent; agility; simulation; kitting.

Table of Contents

List of Tables

List of Figures

Glossary

AI	Artificial Intelligence
AIL	Agent Infrastructure Layer
AJPF	Agent Java Pathfinder
ANTLR	ANother Tool for Language Recognition
API	Application Programming Interface
APRS	Agility Performance of Robotic Systems
ARIAC	Agile Robotics for Industrial Applications Competition
BDI	Belief–desire–intention
BNF	Backus–Naur form
CRCL	Canonical Robot Control Language
DOM	Document Object Model
IA	Intelligent Agent
IDE	Integrated Development Environment
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
IO	Input/Output
JAXB	Java Architecture for XML Binding
JNDI	Java Naming and Directory Interface
JPF	Java Pathfinder
JVM	Java Virtual Machine
LDAP	Lightweight Directory Access Protocol
MAS	Multi-agent system
MCAPL	Model Checking Agent Programming Language
NASA	National Aeronautics and Space Administration
NIST	National Institute of Standards and Technology
PLC	Programming Logic Controller
PDDL	Planning Domain Definition Language
RCS	Real-time Control System
ROS	Robot Operating System
ROS-I	Robot Operating System Industrial
STL	Standard Tessellation Language
SUT	System Under Test
W3C	World Wide Web Consortium
XJC	XML Java Compiler
XML	eXtensible Markup Language
XSD	XML Schema Definition

1. Background

A robot can be considered an intelligent agent (IA) since it perceives its environment with sensors, takes actions autonomously to achieve goals, and may improve its performance by using knowledge gained through experience. A robot that is assigned a goal is considered more intelligent if it consistently takes actions that successfully maximize its goal.

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) [1]. Tasks can also vary in other ways, including timescale, difficulty, and detail. Given a task goal, an agile task planner previews and then selects from the potential actions, based on the current state of the environment 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 a smarter agile agent.

Intelligence defines a wide range of competence. As such, we make clarifying assumptions as to the behavior of the robot intelligent agent. We assume goal driven behavior makes decisions based on trade-offs between conflicting goals in the face of a dynamic environment. Further we assume that the robot is not merely a simple reflexive agent that makes decisions based solely on the current environment but is a goal-based IA capable of thinking beyond the present to determine the best course of actions in light of potentially changing environment to achieve its goal. Typically, a goal-based IA uses lookahead search in planning in order that the planning maximizes the objective function (the goal) and finds the corresponding actions in order to reach it. Dynamic planning is then defined as reevaluating the lookahead search every iteration in its effort at achieving the goal.

Overall, intelligent agents have been characterized into several models of operation, which we base on Weiss [2] categorization:

- Logic-based agents – in which the planning about what action to perform is made via logical deduction where the syntax is defined by first order predicate logic and the semantics are true or false, with a good example of this model being PDDL [3];
- Reactive agents – in which planning is implemented as a direct mapping from goal to action;
- Belief-desire-intention (BDI) agents – in which planning depends upon the manipulation of data structures representing the beliefs, desires, and intentions of the agent;
- Layered architectures – in which planning is realized via multiple levels of software layers,

such that reason about the environment is at different levels of abstraction, e.g., 4D Real-time Control System (RCS) model of Albus [4];

- Subsumption architecture – is a control architecture conceived by Rodney Brooks that instead of planning behavior by using a top-down representation of the world, couples sensory information to action selection in an bottom-up fashion [5].

In this paper, we investigate GWENDOLEN which handles planning based on the BDI intelligent agent model. BDI is used to reason about and plan goal-driven tasks. 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 IA believes to be possible, desires to bring about, and intends to bring about, respectively. Correspondingly, intelligent agents are viewed as being rational and acting in accordance with their beliefs and goals [6].

One advantage of using rational BDI agents over classical planning, such as with PDDL [3], is that BDI failure handling is achieved at the execution level. Assuming that the failure handling plan exists and is correct, then the agent should be able to adapt to the failure using its current plans without resorting to full replanning. The disadvantage is that some flexibility is lost in executing tasks; since all plans are generated at compile time (usually manually coded by a software developer or imported from an automated planner) we can not call a planner at runtime and ask for an optimal solution to an unforeseen problem. Overall, GWENDOLEN planning can be considered more trustworthy for characteristics such as explainability, traceability, interpretability, and safety, since failure recovery planning must be anticipated, and accountable to avoid unintended random responses.

The GWENDOLEN implementation we study differs from other IA programming languages because it was built from the ground up with formal verification in mind [6]. The GWENDOLEN implementation comes equipped with the Agent Java PathFinder (AJPF) [6] model checker, which is based on the NASA JavaPathfinder. Model checking [7] 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 IA program directly, instead of relying on an abstract model. In agent verification, the IA must be verified for not only what it does, but why it chose that course of action, what it believed that made it choose to act in this way, and what its intentions were that led to this plan. Overall, the explicit intentions in BDI agent languages and the possibility of formal verification in GWENDOLEN both provide benefits for understanding and trust.

Even though GWENDOLEN's effectiveness in responding to failures depends on pre-existing and well-programmed plans, these plans can be formally verified to provide assurances that they will behave as expected. Furthermore, verification of autonomous robots can contribute to the trustworthiness of the system and can be vital in application domains such as safety-critical scenarios [8]. By using the GWENDOLEN language, IA programs can be directly verified using model checking, a formal technique that can be used for the verification of safety properties.

To understand the basis of GWENDOLEN planning in operation, a case study using kitting is developed. Kitting is a method to feed a set of subcomponents to an assembly station, where the set of different subcomponents together are used to assemble one unit of a component. Kitting constitutes collecting the subcomponents into a kit which is then transferred to the assembly station [9]. 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 [10]. For our research purposes, evaluation relied on the open-source frameworks (Robot Operating System and Gazebo) for robot control and simulation. Gazebo uses a physics-based engine to simulate such that the models of the robots, kits, and environment provide a higher-fidelity approximation to the real-world.

This report is organized as follows. First, the GWENDOLEN software architecture is presented to understand the structure of the software and the relationships of the elemental system modules involved in the application of GWENDOLEN to kitting. Second, a kitting case study used to evaluate GWENDOLEN will be presented with basic algorithms to handle kitting, as well as discussion on the enhancement of CRCL to return kitting world model status and inferences. Third, GWENDOLEN programming will be studied with an emphasis on coding kitting GWENDOLEN plans. GWENDOLEN notation and code examples will be discussed. Next the use ofGWENDOLEN to program a plan to handle a continuous and a discrete agile planning challenge will be presented in detail.This will be followed by an assessment of GWENDOLEN for handling other Agile Robotics for Industrial Applications Competition (ARIAC) challenges and performance metrics. A summary discussion will review GWENDOLEN planning for handling robotic agent kitting. A series of appendices follow. The first appendix will contain the complete GWENDOLEN program used in the report. The next appendix will examine the the Java programming environment including Java CRCL library building and communication, as well as a breakdown of the Java files used in the GWENDOLEN programming. This will be followed by an appendix that takes an in-depth look at the CRCL kitting model enhancements that were added to the Extensible Markup Language (XML) schema and how Java was used to build and deploy these extensions, including a summary of the inferences used in understanding the kitting world model. The final appendix will give a brief look at ROS CRCL deployment.

2. Gwendolen Software Architecture

GWENDOLEN is an intelligent agent programming language that defines a planning grammar specialized for BDI planning and multi-agent communication. GWENDOLEN IA programs are compiled into a planning/reasoning Java data structures that define the beliefs, desires, and intentions. In conjunction with GWENDOLEN programs, the ability to call native Java functions through an "action" call is available to connect GWENDOLEN plans to the physical "real" world. In essence GWENDOLEN does goal-directed planning using a belief system in conjunction with the real world beliefs in order to run a plan. It is the ability to compose or aggregate GWENDOLEN plans that provides a powerful planning paradigm. The entire GWENDOLEN beliefs and goals program is run as a compiled Java BDI framework that cyclically plans by choosing intentions, and which uses Java to extend the GWENDOLEN program to enable connection to the real world in order to modify the IA beliefs.

A software architecture describes the interaction and structural software components of a system. For GWENDOLEN the software architecture of the implemented system adapted for the kitting case study is shown in Figure 1. Each module role in the GWENDOLEN system architecture will be discussed in the following, as well as its relationship with other modules.

2.1. Gwendolen

GWENDOLEN programs are presented as a series of plans that are compiled from a GWENDOLEN program into the appropriate Agent Infrastructure Layer (AIL) data structures. GWENDOLEN plans are enabled when an IA has certain beliefs and goals and suggests a sequence of deeds to be performed in order to attain a goal. Plans may also be triggered by events, such as a change in belief or the commitment to a new goal. GWENDOLEN agents also distinguish between two types of goals. "Achievement goals" make statements about beliefs the IA wishes to hold. They remain goals until the IA gains an appropriate belief. "Perform goals" simply state a sequence of deeds to be performed and cease to be a goal as soon as that sequence is complete.

2.2. Agent Infrastructure Layer (AIL)

AIL is a toolkit of Java classes designed to support the implementation of BDI (Belief, Desire, and Intention) programming languages that allows for model checking of programs implemented in these BDI languages. Numerous BDI languages have been ported to the AIL platform, including:

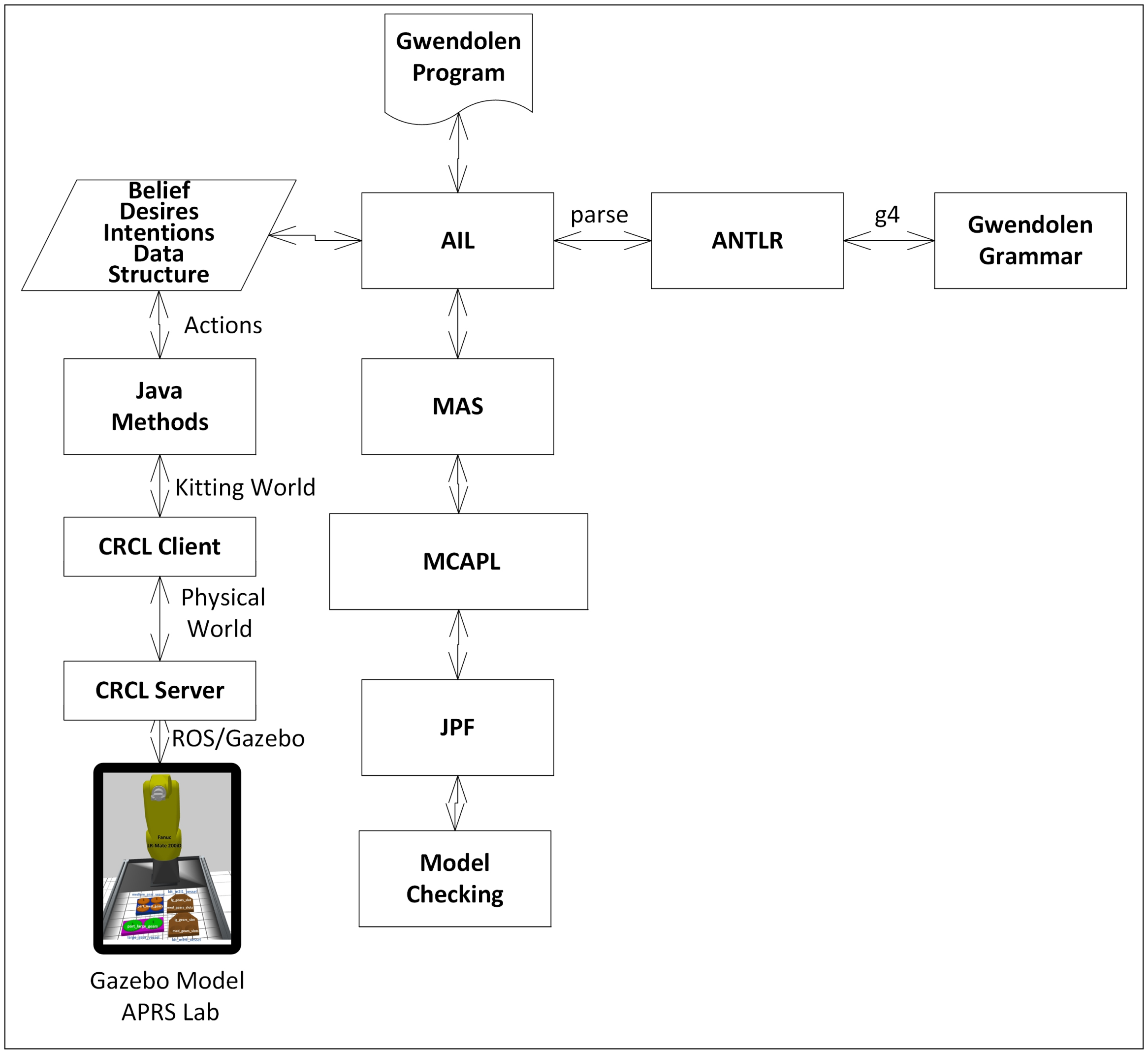


Fig. 1. GWENDOLEN Kitting Software Architecture

Jason [11] implementation of AgentSpeak [12], JAPL [13], SAAPL [14] and GOAL [15]. The AIL contains data structures to store its internal state comprising: a belief base, a plan library, a set of intentions, and other temporary state information which can be adapted to the operational semantics of specific BDI languages.

AIL supports a verification approach that encodes the relevant concepts from the AIL into the model checker just once and then allows multiple languages to benefit from the encoding by utilizing the AIL classes for belief, goal, etc., in their implementation. AIL can potentially prove that it has preserved the semantics of the AIL data structures sufficiently to generate sound results from the model checking processes defined as building blocks.

2.3. Multi-Agent System (MAS)

The MAS aspect of GWENDOLEN agent programming provides synchronization and communication between agents using messages. MAS multi-agent planning was not used in this report.

2.4. Model Checking Agent Programming Languages (MCAPL)

MCAPL interface allows inspection of a GWENDOLEN agent to deduce properties of that agent for use in model checking. MCAPL is a customized Agent JPF extension customized for model verification of AIL agents.

2.5. Agent JPF

Agent Java Pathfinder (AJPF) is an extension of the Java Pathfinder (JPF) model checker for model checking BDI style agent programs.

2.6. Java Pathfinder (JPF)

Initially developed in the early 2000s by NASA, JPF is a verification and testing environment for Java which integrates model checking, program analysis and testing [16]. JPF is implemented in Java and its architecture is modular to support rapid prototyping of new features. JPF implements a custom-made Java Virtual Machine (JVM) that interprets bytecode combined with a search interface to allow the complete behavior of a Java program to be analyzed (including all interleaving's of concurrent programs). JPF supports many model checking features, and by default, it checks for all runtime errors (e.g., uncaught exceptions), assertion violations (supports Java's assert), and deadlocks.

2.7. Model Checking

The model checking used by GWENDOLEN is based on a formal technique to verify software correctness 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). In general, model checking is a technique for verifying state-based concurrent systems with several advantages over traditional approaches that are based on simulation, testing, and deductive reasoning [7]. For embedded real-time systems, model checking uses numerous automated analysis techniques to detect elusive errors in the design of safety-critical systems that often elude conventional simulation and testing techniques.

Unfortunately, model-checking faces a combinatorial explosion of the possible states during analysis that is more commonly known as the state explosion problem. For instance, JPF is an explicit-state model checker software tool, since it enumerates all visited states, and suffers from the state-explosion problem. As such, JPF is ideally suited to analyzing programs less than 10 K lines of code but has been successfully applied to finding errors in concurrent programs up to 100 K lines of code.

2.8. CRCL

To tackle the problem of cooperative robot-independent agility, a communication standard between a host computer and any robot for control and status feedback is necessary. Currently, a myriad of Cartesian and joint level programming schemes are found in commercial off–the–shelf industrial robots that hide the back-end communication scheme from the host computer to the robot. Robot programming that uses a standard-based back-end host-to-robot communication protocol would fundamentally increase the robot agility assuming sufficiently similar capabilities so that the brand of robot would be interchangeable.

The primary standard in process control software programming is International Electrotechnical Commission (IEC) 61131 an international programming standard that is applied to automated industrial robots [17, 18] originally IEC 1131 before the IEC changed its numbering system. In fact, many industrial robots are under the supervisory control of a Programming Logic Controller (PLC) programmed in IEC 61131 invoking proprietary robot programs. To simplify the difficult programming aspect of IEC 61131, PLCOpen has been developed as a graphical XML programming standard that capitalizes on IEC 61131 standard as a back-end device independent representation [19]. IEC-61131 is a programming standard but does not address the host-to-robot communication standard, so that it requires a standard wire interface (such as Foundation Fieldbus [20]) or a proprietary wire interface. The use of a standard robot wire communication standard allows agile collaboration between robot intelligent agents.

A major impediment to PLCOpen and IEC 61131 adoption for all robot programming is the lack of cutting edge planning, motion control, and sensing technology required of advanced robot functionality. For example, robot applications based on ROS offer support for composable system architecture, collision avoidance motion, AI machine learning object grasping, as well as sophisticated sensor fusion, such as navigation based on dynamic sensor mapping. Both industrial and academic robotic application development require advanced software technology that plays a major role in achieving agile robots.

A pure host computer to robot communication standard is the Robot Operating System Industrial

(ROS-I) "Simple Message" protocol [21]. Simple Message is a low-level Ethernet socket interface between ROS host computer and a robot controller that communicates joint messages. A major shortcoming to Simple Message is the omission of Input/Output (IO) status or control (e.g., Emergency Stop). In addition, Simple Message assumes all motion commands are given as joint related command values, so that a Cartesian motion path planned by a real-time robot controller is not possible.

The Canonical Robot Control Language (CRCL) is a higher-level XML abstraction of robot and gripper control and status communication standard. CRCL was developed in the U.S. at the National Institute of Standards and Technology (NIST) [22]. 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 has both joint and Cartesian motion control commands. Although easily extensible, as will be discussed later in this report, CRCL does not currently provide XML abstractions for IO control, sensors, or peripheral robot devices besides grippers. Because CRCL supports Cartesian motion and gripper control, it is well-suited to handle pick and place robot planning operations required for the kitting planning challenges discussed later in the report.

2.8.1. CRCL Client

It is the responsibility of the CRCL Client module to translate between the logical world model required for GWENDOLEN reasoning and a CRCL physical description required of the lower level real world (or in our case simulated real world). For example, GWENDOLEN uses the names for the gears, while it is the responsibility of CRCL to map these gear names into physical locations for commanding a robot. Thus, for each GWENDOLEN action, CRCL maps variables from a logical description into the physical kitting object properties.

To enable planning at a logical reasoning level about the kitting world, the existing CRCL status schema was extended to provide kitting model locations (i.e., gears, trays, kits, etc.) as well as inferences about the kitting models (e.g., a specific gear is located in a supply tray slot). The CRCL status schema extension was intended for reporting on kitting models status in the environment. However, such information could come from a sensor, such as a camera, as well. In fact, the Gazebo simulation model reporting can be considered a "logical" camera in that it reports type and location of an object in the environment. The CRCL status schema extension will be covered further in Appendix 6.

2.8.2. CRCL Server

CRCL Server is an interface to an underlying Robot Operating System (ROS) 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 – ROS and Gazebo – for robot control and simulation. ROS is an open source software framework that provides libraries and tools to help create robot systems. In order to offer robot programming abstraction, the server converts CRCL messages and robot representations into ROS and Gazebo representations. The parsing and serialization of CRCL messages in ROS relied on CodeSynthesis [23] and the Xerces XML Document Object Model (DOM) parser [24]. CodeSynthesis "XSD" is an open-source, cross-platform tool that generates C++ code to handle information modeled in World Wide Web Consortium (W3C) XML Schema. CodeSynthesis uses the open source Apache Xerces XML parser for the CRCL parsing.

2.9. Gazebo Simulation Model of the Kitting World

The NIST Agility Performance of Robotic Systems (APRS) laboratory contains two industrial robots, a Fanuc LR-Mate 200iD and a Motoman SIA20F [25] that have been simulated in Gazebo. 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 objects to combine.

2.10. ANTLR

ANTLR (ANother Tool for Language Recognition) is a public-domain parser generator for reading, processing, executing, or translating structured text. ANTLR combines the flexibility of hand-coded parsing with the convenience of a parser generator [26]. For ANTLR processing, a file name with .g4 extension contains the GWENDOLEN grammar.

2.11. Gwendolen Grammar

GWENDOLEN is a BDI programming language described with a grammar. AIL is designed to accept any representative BDI language that is parseable by ANTLR.

2.12. Run time execution

For GWENDOLEN the system has been shown to be a layered approach to agent planning that is based on a substrate of software model checking. GWENDOLEN is one of many potential BDI languages that can be used by the AIL. The programmer is tasked with developing a combination of a GWENDOLEN BDI program, and the Java action methods that supplement the BDI in planning as well as connecting to the real world.

Upon execution, AIL accepts a GWENDOLEN program file, compiles the GWENDOLEN into a BDI Java representation based on the GWENDOLEN Backus–Naur form (BNF) syntax supplied to ANTLR parser. From a grammar such as GWENDOLEN ANTLR generates a parser that can build and walk parse trees which are used to generate the Java BDI data structures. Once the BDI data structures are built, AIL then deterministically executes the BDI algorithm in solving the GWENDOLEN planning problem.

3. 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 assess agility. 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 understand the GWENDOLEN agile task planner given a list of kitting agility challenges.

As background, the robot kitting operation is tasked with picking the appropriate gears from supply trays, and then placing the gear into a 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 some 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.

For our purposes, GWENDOLEN IA planning for the kitting problem is concerned with loading a kit with two medium gears and one large gear. Of note, low level control and sensing functionality, such as grippers, vision cameras, or other sensors is not relevant at the intelligent agent level of planning for which GWENDOLEN is being evaluated. Clearly the problem can 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. Later, specific GWENDOLEN code to handle agility tests such as dropped gears will be discussed.

3.1. Kitting World and Beliefs

For the GWENDOLEN assessment, the kitting world is defined as: two kit trays, one medium supply tray, one large supply tray, and all the slots each tray contains. Initially, all the supply trays slots contains gears, while all the kitting trays slots are empty. Figure 2 shows the layout of the initial kitting world.

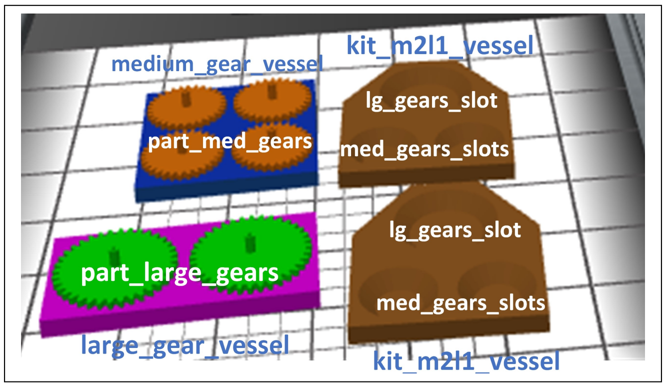


Fig. 2. Initial Kitting World

For GWENDOLEN we use the kitting trays, slots, and gear setup as the initial "belief". These are the beliefs our system starts with when planning. Gear supply trays (or vessels) are defined as the name, type of gear contained, and then a "[]" delimited list of gear names. Kit trays are defined with a kit name, and then slots, where a slot is defined by name (tray name, a period, and slot name), gear type (small, medium or large) and then the state (either empty or the gear name). The use of the kit name concatenated by a period and then the slot name, was a simple mechanism to realize a fully qualified name that can be recognized with just the name string. For grasping logistics, the belief that the gripper is open is asserted also. The following shows the GWENDOLEN initial beliefs:

Listing 1. GWENDOLEN Initial Beliefs World Model

```
:Initial Beliefs:
gripper("open")

gear_tray(medium_gear_vessel16, "medium", [
        "part_medium_gear17",
        "part_medium_gear18",
        "part_medium_gear19",
        "part_medium_gear20"])

gear_tray(large_gear_vessel21, "large", ["part_large_gear22","part_large_gear23"])

kit_tray(kit_m2l1_vessel14,[
    slot("kit_m2l1_vessel14.slot1","medium","empty"),
    slot("kit_m2l1_vessel14.slot2","medium","empty"),
    slot("kit_m2l1_vessel14.slot3","large","empty")])

kit_tray(kit_m2l1_vessel15,[
    slot("kit_m2l1_vessel15.slot1","medium","empty"),
    slot("kit_m2l1_vessel15.slot2","medium","empty"),
    slot("kit_m2l1_vessel15.slot3","large","empty")])
```

3.2. Basic Kitting Algorithm

The basic kitting algorithm, which assumes no error handling, follows. To fulfill a kitting task, first the robot must receive a kitting order (just one at a time for now) and the planner must reason about the kitting problem. The basic planning sequence is shown in Table 1. First, a plan is produced containing kitting actions that are then translated into a CRCL step to be transmitted to the Gazebo simulator. Changes to the world model are reported through CRCL within the status report. These changes include physical properties and or logically derived properties from the physical properties (such as changes to occupancy of tray slots). Any failures are reported by CRCL.

In Table 1, the *acquire_part* plan is run where the goal is to find an open slot in a kit tray and then a matching free gear of the same size in a supply tray. For this planning step, we assume 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, close gripper, 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, open gripper, and retract from the kitting open slot physical pose location.

The CRCL collection of commands execution is not monolithic to be sent all at once as a group, but instead are sent as a "drip feed" to control the robot. In other words, the start of a CRCL "init" alerts the CRCL Server that a CRCL command sequence is about to start. Then, after acknowledgement,

Planner	GWENDOLEN	**CRCL**
acquire_part	Beliefs: gripper is empty Goals: 1) find_slot is a GWENDOLEN Java action to find empty kitting slot, 2) find_gear is a GWENDOLEN Java action to find matching supply gear from supply tray	CRCL status 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	take_part is a GWENDOLEN Java action that does the following: approach gear grasp gear retract	init (speeds, units) moveto pose (approach) moveto pose (grasping point) setgripper 1 (close) moveto pose (retract) end
place_part	place_part is a GWENDOLEN Java action that does the following: 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. Kitting Planning Steps

this is followed by a transmission of each command in order that waits for either a positive status or error status acknowledgement. For a positive acknowledgement, the CRCL Client proceeds to the next CRCL command until an "end" command is reached. Should the CRCL Server respond with a negative or error status, then the CRCL Client will have to interpret and then translate the error result into the GWENDOLEN planning domain.

4. Gwendolen Programming

GWENDOLEN programming is the focus of this section. Since GWENDOLEN has a unique notation, illustrating basic concepts will help in understanding its programming approach. We will use robotic kitting again for examples of code to understand GWENDOLEN intelligent agent programming. Readers are referred to [6, 27] for an in-depth presentation on the syntax, structure, and features of GWENDOLEN. Hereafter, we will cover the basics of GWENDOLEN language.

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:

$$[+!goal \mid \pm belief] : \{guard\} \leftarrow body_1, body_2, \ldots, body_n; \quad (1)$$

A plan may match the top event of an intention, either a goal or the addition/removal of a belief. The guard condition is checked against the agent's state (beliefs) for a matching true belief. For example, $B_{grasped}(Gear)$ is a guard which says proceed if the belief (denoted by B) grasped(Gear) is true. Upon a true conditional guard, the plan body steps are executed. A body contains a series of plan goals shown as $body_1, body_2, \ldots, body_n$ in (1). The body statements can be other goal plans, or Java actions for execution, or belief fulfillment.

4.1. Gwendolen Notation

Briefly, GWENDOLEN uses the following notation to differentiate various BDI concepts, where:

Notation	**Description**
+	denotes the addition of a belief,
-	denotes the removal of a belief,
+!	denotes the addition of a goal,
{ }	encloses the guard (context/precondition) of the plan,
$\sim$	denotes negation in this case of a belief (represented by B),
(_)	denotes a universal variable (which can match with any value).
$\leftarrow$ find_gear(Slots)	$\leftarrow$ is the start of the body of the plan, find_gear(Slots) is an action and Slots is a variable
*gear(Gear)	the * means it waits until it has the belief gear(Gear) to proceed with the plan (in this instance, this belief is added through the execution of the previous action, find_gear(Slots)
+!move(Gear)	adds the goal move(Gear); this will trigger another plan that must be executed before returning to this plan
+!close_gripper	similar to +!move above, but this plan does not have an argument

Table 2. GWENDOLEN Notation

4.2. Gwendolen Example

As an example, the GWENDOLEN code for "kitting" is given in Listing 2. As shown, *kitting* is a goal plan (denoted by $+!$ that is triggered by a matching goal intention with appropriate parameters *Id, Size*, where Id is a kitting open slot, while Size is the open slot size (small, medium, or large). The goal plan conditional guard has two beliefs *{ B grasped(_), B gear_tray(IdGearTray, Size, Slots) }* such that a condition only triggers when the belief that a gripper is not grasping any item and a belief that a gear supply tray belief exists with tray name, size, and all slots in the tray are true. Consequently if the guard is true, the goal plan body performs a series of actions including a *find_gear(Slots)*, which is a Java action to find a free gear from the given the tray slots. **gear(Gear)* waits until the *find_gear* is done and returns a free gear. Note only trays with gears in slots are provided to *find_gear* or it would not be a belief. *+!take_part(Id,Gear) [perform]* is a GWENDOLEN goal action to grasp the Gear. So it is part of the GWENDOLEN goal planning. *+!place_part(Id,Gear) [perform]* is another GWENDOLEN goal plan but this time its goal is to place Gear into slot id. Note, the GWENDOLEN goal is in fact another plan containing subplans.

Regarding the $[perform]$ designation on the goal invocation, GWENDOLEN recognises two types of goal, "achieve" goals and "perform" goals [28]. An important distinction concerns the replanning of GWENDOLEN failed goals. "Achieve" goals mean that if, after execution of the plan, the goal is not achieved then it will be replanned, which is designated by a trailing "[achieve]". "Perform" goals always trigger planning but are not replanned if they fail to achieve some state of the world. In our code we will rely stricly on $[perform]$ as the sole GWENDOLEN replanning mechanism designated by a trailing "[perform]".

Listing 2. GWENDOLEN Kitting Plan

```
// Kitting is a goal with parameters Id, Size
// Id is the kitting open slot, while Size is the open slot size
// [perform] is a replanning designation
// There are two beliefs:
// ~B grasped(_) is a belief that says the gripper must not be grasping anything already
// B gear_tray(IdGearTray, Size, Slots) says a gear supply tray belief exists with tray
    name, size and slots
+!kitting(Id, Size) [perform] : { ~B grasped(_), B gear_tray(IdGearTray, Size, Slots) }
<-
        find_gear(Slots), &\Comment{// Java action to find a free gear given the slots}&
        *gear(Gear), &\Comment{// waits until find\_gear returns a free gear}&
        +!take_part(Gear) [perform], &\Comment{// Goal action to grasp the Gear }&
        +!place_part(Id,Gear) [perform]; &\Comment{// Goal to place Gear into slot id}&
```

5. Gwendolen Kitting Agility

Agility has been defined as the capability of surviving and prospering in a dynamic environment of continuous and unpredictable change by reacting quickly and effectively [29, 30]. The IEEE Robotics and Automation Standards Working Group on Robot Agility [31] provides a list of desirable agility traits of robotic systems which is a complex combination of reconfigurability and autonomy in contrast to the more pervasive pendant-taught robot programming. Aspects of robot agility include hardware reconfigurability, software reconfigurability, communications, task representation, sensing, perception, reasoning, planning, tasking, and execution [32]. Robot agility exhibits many elements of intelligence through adapting to a challenging environment.

In order to assess agile planning, performance metrics are used to judge differing systems and approaches to planning. Performance metrics for kitting have been previously studied in Downs et al. [33]. We will explore a few robot agility scenarios and how GWENDOLEN and its Java Agent environment address the agility issues. In the application of GWENDOLEN planning to robotics, the metrics of interest are agility and correctness. The agility handling of abnormal events is the major thrust of GWENDOLEN evaluation, but key aspects of agility include the detection, handling, and recovery of application agility challenges. For example, kitting agility includes handling of hardware faults, dropped parts, missing parts, misoriented parts, or faulty parts. Table 3 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.

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, were not attempted. Safety violations such as humans in danger are considered. Finally, capability metrics such as payload or speed violations were out of scope, but reach was .used to assess ability to pick up dropped gear.
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 3. Agile IA Kitting Performance Metrics

We will leverage the pre-existing agility challenges in order to explore GWENDOLEN for its planning agility by developing and discussing kitting scenarios that handle several of the challenges.

5.1. Scenario One: Continuous Monitoring of Human Robot Proximity

In scenario one, we apply GWENDOLEN to handling of a continuous task for safely monitoring the proximity of a human to the robot. This is important as most often robot accidents happen when a human may temporarily be within the robot work envelope while power is available to moveable elements of the robot for non-application, such as service, maintenance, and calibration of the system [34]. So not surprisingly, one of the more important safety issues when dealing with robots is restricting the closeness of a human to a robot. In these circumstances, where the human does enter the robot workspace, the robot should immediately stop operation and power until the human exits the robot workspace and is safe.

We will briefly sketch one method in which GWENDOLEN could handle such a "continuous" event - monitoring a human safety violation when entering a robot workspace. Our method to achieve this in GWENDOLEN is to create the *humanProximityViolation* belief in the Java code and then having the GWENDOLEN detect_human_violation plan trigger and then wait until human has exited the robot workspace by waiting until this critical safety condition has cleared by monitoring its condition in the Java code.

The GWENDOLEN code snippet to effect this agent safety behavior is shown in listing 3. GWENDOLEN code for this declares a belief guard monitoring for the creation of a *humanProximityViolation* belief. When the belief *humanProximityViolation* is found to exist, the body of the plan performs the Java code snippet "wait_on_belief", which contains code to wait until the human proximity violation is not longer valid, and then nullifies the *humanProximityViolation* belief by removing it. The "remove_belief" Java action will actually remove the *humanProximityViolation* belief from the GWENDOLEN environment in Java.

Listing 3. GWENDOLEN Human Proximity Belief

```
+humanProximityViolation : { B humanProximityViolation}
<-
   wait_on_belief("humanProximityViolation"),
   remove_belief("humanProximityViolation");
```

In the Java agent environment, a background thread is monitoring human proximity to the robot, and if the distance indicates a proximity violation, we create an environment perception corresponding to a *humanProximityViolation* belief. In Java we do this, by first declaring environment predicate *humanProximityViolationPredicate* which will be used to augment the agent environment when a human proximity violation occurs. For simplicity, we randomly create and then add the *humanProximityViolationPredicate* environment Perception to the agent belief system. At this point, the GWENDOLEN code will trigger the guard associated with this belief described previously. The Java code snippet to effect this agent behavior is shown in listing 4.

Listing 4. GWENDOLEN Java Code to set environment Perception

```
// Declare environment predicate
public Predicate humanProximityViolationPredicate;
...
// Gwendolen Java actions connection to environment
public Unifier executeAction(String agName, Action act) throws AILexception
...
// code to detect human proximity violation omitted

// create and then add environment Perception
humanProximityViolationPredicate = new Predicate("humanProximityViolation");
addPercept(humanProximityViolationPredicate);
```

Within the executeAction Java code we also create a *remove_belief* action to remove the perception from the environment when the human proximity violation has been cleared. Once cleared, we can then remove the *humanProximityViolation* environment perception. Note, we can only remove the *humanProximityViolation* perception that was created in Java (as part of the environment) in the Java agent environment. The Java code snippet to effect this agent behavior is shown in listing 5.

Listing 5. GWENDOLEN Java to Remove an Existing Precept

```
if (actionname.equals("remove_belief")) {
  StringTerm args = (StringTerm) act.getTerm(0);
  if (args.toString().equalsIgnoreCase("humanProximityViolation"))
  {
    // Code to remove human proximity, as its now safe
    removePercept(humanProximityViolationPredicate);
  }
}
```

Timing issues remain a concern as there is no determinism associated with creating and responding to the humanProximityViolation belief. Of note, it is not clear if this is the best mechanism for dealing with real-time, continuously monitored, and spontaneous event in GWENDOLEN. In theory, having this safety belief be continuously monitored before each GWENDOLEN cycle would be preferable, but does not seem to be part of GWENDOLEN programming paradigm.

5.2. Scenario Two: Dropped Gear Challenge

GWENDOLEN is a good fit for the discrete event agile kitting case mainly due to the reactive nature of agent-oriented programming. If an abnormal event happens during the execution but has been identified (failures, decrease in performance, etc.), then this abnormal event will trigger the appropriate plan to react to it. It is important for the decision-making software to be able to handle these abnormal situations in order to remain trustworthy as part of being agile. The following scenario will explain how to use GWENDOLEN to handle a discrete abnormal event, which in this case is a “dropped kitting gear”.

As explained earlier, the basic kitting algorithm identifies an open kit slot, finds a matching gear from the supply tray that matches the empty slot size, grasps the gear; and then places the gear into the open, matching gear size, kitting slot. Readily adapting to dropped gears could be considered a litmus test for a kitting planner. We will explore handling a few combinations of discrete problems associated with dropping the gear when "taking" the gear. In the best case, we would like to recover gracefully from the dropped gear, and resume the kitting procedure if possible.

Assuming the gear has been dropped, we handle a few potential scenarios. For now, we will ignore how the robot dropped the gear, be it either the momentary loss of pneumatic pressure to the gripper or mishandling of the gear when grasping or losing hold of gear while moving to a goal destination due to insufficient friction while grasping the gear. Further we will assume the dropped gear lands in a correct, graspable orientation. Problems such as gripper air pressure or robot servo errors would mean aborting all kitting and calling maintenance. Handing off kitting operation to a nearby robot is out of scope in this scenario.

Given this scenario, the agility problem can range from:

- attempting to pick up the gear again which assumes the gear is within reach,
- the gear is out of reach, and thus requires aborting this kitting operation and moving on to the next kitting operation, possibly alerting personnel
- completely shutting off the robot operation due to some system fault.

For a potential best-case recovery operation, the dropped gear could remain on the table and be within the reach of the robot so that it can try picking up the gear again.

We will explore the GWENDOLEN programming technique to wait on the completion of the "take_part" operation and error handling. GWENDOLEN has a wait on belief operation $*abelief(code)$ that signals that a plan waits until an *abelief* has been added to the environment, and then proceeds to call a plan based on the *code* which can be true for successful operation or some text describing the failure reason. In listing 6 we see the $+!take\_part(Location)$ plan contains a line of code $*action\_result(Result)$, that waits for the Java *take_part* operation to create an *action_result* belief with a list item *Result*.

Listing 6. GWENDOLEN Code to take a part from a tray

```
+!take_part(Location) [perform] : { ~B grasped(_), B gripper("open")}
<-
       take_part(Location),
        *action_result(Result),
        +!check_action_result_take_part(Location, Result) [perform];
```

As mentioned, the $*action\_result(Result)$ waits until a status has been returned from *take_part*.

Using this *Result* we can then branch to the *check_action_result_take_part* plan appropriate for the success of the *take_part* operation. The *Result* of the action is then used to call one of several $+!check\_action\_result\_take\_part(Location, Result)$.

When the *Result* is "true", the grasping of the gear consisting of the sequence of lower level operations approach, pick, and retract has been successfully completed. In this case the GWENDOLEN *take_part* plan is shown in listing 7. The *take_part* plan essentially updates the belief system regarding the gripper and the grasped object.

Listing 7. GWENDOLEN Update to Belief upon Successful Take Part

```
+!check_action_result_take_part(Location, "true") [perform] : { True }
<-
        -gripper("open"),
        +gripper("close"),
        +grasped(Gear);
```

Listing 8 shows three agile GWENDOLEN error handling goal plans all named $+!check\_action\_result\_take\_part$. For $+!check\_action\_result\_take\_part$ $(Location,$ "$droppedGear''$) we check to see if the dropped gear is reachable.

This involves calling the $+!check\_action\_result\_take\_part(Location,$"$reachableGear''$) GWENDOLEN goal plan which returns either an "abort" or a "reachableGear" to resume the kitting operation. Of note, the "reachableGear" $+!check\_action\_result\_take\_part$ plan attempts the *take_part* goal plan again, which can have unintended consequences as it is not a numerically bounded test and could result in an infinite planning loop: grasp, drop, grasp, drop, etc. This is known as the "butter-fingers" robot infinite drop loop [3]. Although seemingly benign and improbable, such a set of circumstances is indeed possible, and a completely verifiable and correct IA should anticipate and intelligently handle such egregious circumstances.

Listing 8. GWENDOLEN Plans for recovering from errors

```
+!check_action_result_take_part(Location, "droppedGear") [perform] : { True}
<-
        reachable_gear(Location),
        *action_result(Result),
        -action_result(Result),
        +!check_action_result_take_part(Location, Result) [perform];

+!check_action_result_take_part(Location, "reachableGear") [perform] : { True}
<-
        take_part(Location),
        *action_result(Result),
        -action_result(Result),
        +!check_action_result_take_part(Location, Result) [perform];

+!check_action_result_take_part(Location, "abort") [perform] : { True }
<-
        trace("check_action_result_take_part abort taking Gear");
```

[3]Yes I made it up!

Listing 9 shows the trace of operations using the GWENDOLEN kitting recovery plans as outlined above. We use a simulated failure, in that the *take_part* Java code has randomly dropped the gear to simulate some abnormal event, and when triggered, returns the "droppedGear" result. Subsequently the goal plan *check_action_result_take_part* for "droppedGear" determines whether the gear is within reach. If the gear is reachable, the goal plan *check_action_result_take_part* for "reachableGear" is invoked which attempts the *take_part* goal plan again. Since the Java code only simulates one "dropped gear", *take_part* goal plan succeeds and then the goal to *place_part* is performed.

Listing 9. GWENDOLEN Plans for recovering from errors

```
find_slot
find_gear
take_part
    take_part(part_large_gear22)
    check_action_result_take_part droppedGear
            reachable_gear(part_large_gear22)
            check_action_result_take_part reachableGear
                take_part(part_large_gear22)
                check_action_result_take_part T
place_part
    place_part(kit_m2l1_vessel14.slot3)
    check_action_result_place_part T
```

5.3. ARIAC Scenario Discussion

The ARIAC competition has a list of agility challenges [35] and performance metrics that contribute to a final competition score, such as system cost, and some other Key Performance Indicators (e.g., total cycle time, throughput) [36]. Previous winners of ARIAC competitions have documented both agility challenges and the technical approach they adopted in winning the ARIAC competition as well as handling the kitting challenges [37–39].

So far, two types of kitting challenges have been programmed and examined in GWENDOLEN: discrete and continuous. The discrete agile event handling can be generalized to handle many of the ARIAC competition challenges, such as faulty gripper or robot servo fault, sensor fault, and random fault constraints. Although GWENDOLEN has been shown to handle a continuous agile event, it is a more circumspect solution.

Some of the ARIAC challenges do not appear to be solvable for a novice GWENDOLEN planning programmer. The ability to exhibit adaptive GWENDOLEN planning for handling priority-based interruptions, such as a new and higher priority kitting order, is not readily apparent. Another ARIAC challenge is exhibiting flexibility in robot sharing work in the case where one robot has suffered a catastrophic fault (or "blackout"), and cannot continue kitting, and needs to delegate to another IA robot. GWENDOLEN planning does support MAS but was not tested. The lack of sharing of BDI or the world model between robots could make hand-off of partially completed task daunting. One of the biggest headaches found in the robot industry is resuming a line of robots after a fault brings one of the robots down. Clearly, discarding unfinished auto bodies is not feasible.

In order to maximize the ARIAC performance metric of kitting cycle time or in other words complete the kitting as quickly as possible, optimizing kit filling by minimizing the robot distance traveled is an obvious goal in planning. For optimization, GWENDOLEN planning could rely on embedded Java actions 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 nor intrinsically obvious. Since the scope of the kitting was limited to basic GWENDOLEN kitting planning evaluation, acknowledging potential issues was deemed a sufficient level of evaluation.

In addition to the ARIAC challenges, GWENDOLEN has lower-level planning issues that could present problems, including:

- managing system correctness, such as do the beliefs match the environment. A simple example of a flawed belief system, is the logical paradox when both $B \& \neg B$ are both true beliefs, such as the case when the belief grasping a gear and gripper open both exist (and implicitly true in GWENDOLEN).
- detecting endless repetitive loops in overcoming dropped gear using a timing or loop count mechanism.
- determining if kitting sufficiency requirements are satisfied such as bad gear orientation such as upside down, insufficient free gears, or defective gears.
- faulty sensor error detection such as reporting incorrect gear locations and orientations and understanding for instance, handling the CRCL error indicating a failed grasping operation.
- part quality so that defective parts are not used.
- robot sufficiency requirements, enough payload, reach, dexterity to pick, possibly reorient, and place a gear. If unable to reorient a gear, an extra reorienting device may be necessary.

To handle many continuous challenges, a periodic Java action can be run every cycle to reassess challenges, for example, every cycle determining if the kit requirements are satisfied by the free gear supply. If the continuous planning requirements are not met, an environment perception belief can be issued that triggers a GWENDOLEN error handling plan.

6. Discussion

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 report, we address the issue of intelligent behavior in the face of abnormal and challenging events. Due to several, yet seemingly inadequate, solutions to IA planning, the GWENDOLEN programming language applied to a manufacturing kitting case study was studied. GWENDOLEN is designed to handle and recover from abnormal but foreseen events. This report studies the GWENDOLEN programming and its response to various kitting agility problems against a proven set of kitting performance metrics from the ARIAC competition. A GWENDOLEN program offers the planning option of composable plans to handle potential errors, which was explored as a solution to the requirement of correctness, efficacy, as well as safety.

For the robotic IA, ignoring an error state can be unthinkable. Minimally, according to Asimov's Three Laws of Robotics, the robotic IA should when recognizing there is a problem, cause no harm, gracefully operate until the problem is fixed, and attempt to fail-safely as a last resort. In all, there are degrees of agility in handling faults, such as, acknowledgement and full remediation and recovery from the problem, replanning an alternative strategy, or understanding the problem and using a substitute backup plan. Although the case study concentrated on the fundamental elements of kitting, planning for challenges either discrete or continuous were explored. Clearly, challenging tests are necessary to evaluate run-time agility performance. In order to undertake these tests, simulation was used to inject challenges at random times to evaluate system responsiveness. This was done by replacing the actual communication to a robot with a "loopback" robot communication option and pseudo-randomly introducing errors.

GWENDOLEN is suited for adapting to and recovering from kitting failures. Since GWENDOLEN plans are composable, illustrative conditions checking for errors 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 its native Java programming interface. Safety violations such as humans in danger or capability violations were studied with GWENDOLEN as this issue is paramount to industrial root planning.

We have found GWENDOLEN to exhibit many positive qualities. Most intriguing include the ability to verify internal correctness operation via JPF checks. Some of the GWENDOLEN verification techniques may not be appropriate, for many of the considered recovery techniques could lead to infinite loops, timing issues, or other failings. Trustworthiness of any IA computer system is an overwhelming consideration regarding system confidence and acceptance. Without these attributes, adoption of the technology is unclear.

Future work would leverage the agent-based nature of GWENDOLEN to explore more of the ARIAC established challenges to verify the robustness of the GWENDOLEN planning. Potential work includes work based on ARIAC agility challenges including the integration of multi-agent shared error handling in an adaptive manner. Other illustrative agile challenges desirable for GWENDOLEN IA planning include recovery from robot payload violations, robot reach improbabilities, and the intrinsic understanding robot capabilities exemplified by these challenges.

Demonstration code for this report can be found in the github autonomy-and-verification user under the gwendolen-crcl-kitting repository located at github.com/autonomy-and-verification/gwendolen-crcl-kitting.

References

[1] Gerkey BP, Matarić MJ (2004) A formal analysis and taxonomy of task allocation in multi-robot systems. *The International Journal of Robotics Research* 23(9):939–954.

[2] Weiss G (2013) *Multiagent systems* (MIT Press), 2nd Ed.

[3] McDermott D (2003) The formal semantics of processes in PDDL. *Proceedings of ICAPS Workshop on PDDL* (Citeseer), pp 101–155.

[4] Albus J, Huang HM, Messina E, Murphy K, Juberts M, Lacaze A, Balakirsky S, Shneier M, Hong T, Scott H, Proctor F, Shackleford W, Michaloski J, Wavering A, Kramer T, Dagalakis N, Rippey W, Stouffer K, Legowik S (2002) 4d/rcs version 2.0: A reference model architecture for unmanned vehicle systems (National Institute of Standards and Technology, Gaithersburg, MD), NISTIR 6910. https://doi.org/https://doi.org/10.6028/NIST.IR.6910

[5] Brooks R (1986) A robust layered control system for a mobile robot. *IEEE Journal on Robotics and Automation* 2(1):14–23.

[6] Dennis LA, Fisher M, Webster MP, Bordini RH (2012) Model checking agent programming languages. *Automated Software Engineering* 19(1):5–63.

[7] Clarke EM, Grumberg O, Kroening D, Peled D, Veith H (2018) *Model checking* (MIT press).

[8] Farrell M, Luckcuck M, Fisher M (2018) Robotics and integrated formal methods: Necessity meets opportunity. *Integrated Formal Methods* (Springer), *LNCS*, Vol. 11023, pp 161–171.

[9] Caputo AC, Pelagagge PM, Salini P (2018) Economic comparison of manual and automation-assisted kitting systems. *IFAC-PapersOnLine* 51(11):1482–1487. https://doi.org/10.1016/j.ifacol.2018.08.293

[10] Johansson MI (1991) Kitting systems for small size parts in manual assembly systems. *Production Research - Approaching the 21st Century*, eds Pridham M, O'Brien C (London: Taylor Francis), pp 225–230.

[11] Bordini RH, Hübner JF, Wooldridge M (2007) *Programming multi-agent systems in AgentSpeak using Jason*. Vol. 8 (John Wiley & Sons).

[12] Rao AS (1996) AgentSpeak (L): BDI agents speak out in a logical computable language. *European Workshop on Modelling Autonomous Agents in a Multi-agent World* (Springer), pp 42–55.

[13] Bahaj M, Soklabi A (2013) JAPL: The JADE agent programming language. *Journal of Emerging Technologies in Web Intelligence* 5(3):272–278.

[14] Winikoff M (2007) Implementing commitment-based interactions. *Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems*, pp 1–8.

[15] de Boer FS, Hindriks KV, van der Hoek W, Meyer JJC (2007) A verification framework for agent programming with declarative goals. *Journal of Applied Logic* 5(2):277–302.

[16] NASA (2012) A Model Checker for Java Programs, ti.arc.nasa.gov/tech/rse/vandv/jpf. Accessed: 1-Nov-2021.

[17] Bristol Babcock IEC-1131 - the first universal process control language, www.automation.com/en-us/articles/2016-1/iec-1131-the-first-universal-process-control-langu. Accessed: 2022-2-2.

[18] IEC Programmable controllers – Part 1 General information, webstore.iec.ch/preview/info_iec61131-1%7Bed2.0%7Den.pdf. Accessed: 2022-2-2.

[19] van der Wal E (2009) PLCopen. *IEEE Industrial Electronics Magazine* 3(4).

[20] FieldComm Group Foundation fieldbus technical specifications, www.fieldcommgroup.org/foundation-fieldbus-technical-specifications#230548828-3323929298. Accessed: 2022-2-2.

[21] ROSorg simple_message, wiki.ros.org/simple_message. Accessed: 2022-2-2.

[22] Proctor F, Balakirsky S, Kootbally Z, Kramer T, Schlenoff C, Shackleford W (2016) The canonical robot command language (CRCL). *Industrial Robot: An International Journal* 43:495–502. https://doi.org/10.1108/IR-01-2016-0037

[23] CodeSynthesis XSD XML data binding for C++, www.codesynthesis.com/products/xsd. Accessed: 2018-12-11.

[24] The Apache Software Foundation Xerces C++ parser, xerces.apache.org/xerces-c. Accessed: 2018-12-11.

[25] Piliptchak P, Aksu M, Proctor FM, Michaloski JL (2019) Physics-based simulation of agile robotic systems. *ASME International Mechanical Engineering Congress and Exposition* (American Society of Mechanical Engineers), Vol. 59384.

[26] Parr TJ, Quong RW (1995) ANTLR: A predicated-LL (K) parser generator. *Software: Practice and Experience* 25(7):789–810.

[27] Dennis LA, Farwer B (2008) Gwendolen: A bdi language for verifiable agents. *Proceedings of the AISB 2008 Symposium on Logic and the Simulation of Interaction and Reasoning, Society for the Study of Artificial Intelligence and Simulation of Behaviour* (Citeseer), pp 16–23.

[28] Dennis L Gwendolen semantics: 2017, personalpages.manchester.ac.uk/staff/louise.dennis/pubs/ulcs-17-001.pdf. Accessed: 2021-12-11.

[29] Gunasekaran A (1999) Agile manufacturing A framework for research and development. *International Journal of Production Economics* 62(1-2):87–105.

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

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

[32] Downs A, Kootbally Z, Harrison W, Pilliptchak P, Antonishek B, Aksu M, Schlenoff C, Gupta

SK (2021) Assessing industrial robot agility through international competitions. *Robotics and Computer-Integrated Manufacturing* 70:102–113. https://doi.org/10.1016/j.rcim.2020.102113

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

[34] OSHA (1987) Guidelines for robotics safety (U.S. Department of Labor), STD 01-12-002.

[35] National Institute of Standards and Technology ARIAC agility challenges, github.com/usnistgov/ARIAC/blob/ariac2021/wiki/documentation/agility_challenges.md. Accessed: 2021-12-11.

[36] National Institute of Standards and Technology ARIAC scoring, github.com/usnistgov/ARIAC/blob/ariac2021/wiki/documentation/scoring.md. Accessed: 2021-12-11.

[37] Feng SW, Guo T, Bekris KE, Yu J (2021) Team rubot's experiences and lessons from the ARIAC. *Robotics and Computer-Integrated Manufacturing* 70:102–126.

[38] Vidács A, Szabó G (2021) Winning ARIAC 2020 by KISSing the BEAR: Keeping things simple in Best Effort Agile Robotics. *Robotics and Computer-Integrated Manufacturing* 71:102166.

[39] Wan G, Dong X, Dong Q, He Y, Zeng P Design and implementation of agent-based robotic system for agile manufacturing: A case study of ARIAC 2021. *Robotics and Computer-Integrated Manufacturing* Submitted.

[40] Apache Netbeans, netbeans.apache.org. Accessed: 2022-4-22.

[41] Apache Maven, maven.apache.org. Accessed: 2022-4-22.

[42] National Institute of Standards and Technology CRCL to ROS github repository, github.com/usnistgov/crcl2ros. Accessed: 2021-12-11.

Appendix A: GWENDOLEN Kitting Program

A GWENDOLEN program consists of a file with five elements: name, initial beliefs, reasoning rules, initial goals, and plans. The ":name:" of the IA robot is "lrmate". The "initial beliefs" describe the beliefs of the starting IA and has previously been discussed when describing the starting kitting world model. Of note, at the higher level of GWENDOLEN IA planning, we have assumed all the belief definitions are logical, and no physical location of trays, slots or gears are defined or required. The belief for a tray slot require either the slot contains a gear or is "empty". The "initial goal" is to perform the "new_tray" action. The "plans" for "place_part" are not as fully fleshed out as the "take_part" plans, which is used to illustrate GWENDOLEN agility programming features. We do not have any "reasoning rules".

Listing 10. GWENDOLEN Kitting Listing

```
// Name of the intelligent agent Fanuc LRMate
:name: lrmate

// For more information See :Initial Beliefs:
// tray and kit setup explained earlier
:Initial Beliefs:
gripper("open")

gear_tray(medium_gear_vessel16, "medium", [
       "part_medium_gear17",
       "part_medium_gear18",
       "part_medium_gear19",
       "part_medium_gear20"])

gear_tray(large_gear_vessel21, "large", ["part_large_gear22","part_large_gear23"])

kit_tray(kit_m2l1_vessel14,[
    slot("kit_m2l1_vessel14.slot1","medium","empty"),
    slot("kit_m2l1_vessel14.slot2","medium","empty"),
    slot("kit_m2l1_vessel14.slot3","large","empty")])

kit_tray(kit_m2l1_vessel15,[
    slot("kit_m2l1_vessel15.slot1","medium","empty"),
    slot("kit_m2l1_vessel15.slot2","medium","empty"),
    slot("kit_m2l1_vessel15.slot3","large","empty")])

:Initial Goals:
new_tray [perform]

:Plans:
+!new_tray [perform] : { B kit_tray(IdKitTray,Slots) }
<-
    find_slot(Slots), &\Comment{ // call Java find\_slot action}&
    -kit_tray(IdKitTray,Slots), &\Comment{ // remove belief}&
    +!loop [perform]; &\Comment{ // initiate goal}&

// loop through all the kit open slot with size
+!loop [perform] : { B slot_active(Id, SizeSlot, Slot) }
<-
    -slot_active(Id, SizeSlot, Slot), &\Comment{// remove belief}&
    +!kitting(Id, SizeSlot) [perform], &\Comment{// perform kitting goal}&
    +!loop [perform]; &\Comment{// loop while slots\_active belief is true}&

// Kitting goal plan takes kit slot and gear size
```

```
+!kitting(Id, Size) [perform] : { ~B grasped(_), B gear_tray(IdGearTray, Size, Slots) }
<-
    find_gear(Slots),&\Comment{// invoke Java action}&
    *gear(Gear),&\Comment{// wait for Java gear result}&
    +!take_part(Gear) [perform],&\Comment{// perform gwen take\_part goal with gear}&
    +!place_part(Id,Gear) [perform]; &\Comment{// perform place\_part twen goal with id
        slot and gear}&

// Make slot-active belief given lot belief belief match
+slot(Id, SizeSlot, Slot) : { True }
<-
    +slot_active(Id, SizeSlot, Slot);

// Keep looping goal if kitting tray belief exists
// notes, this comes before loop below with simple
// true belief
+!loop [perform] : { B kit_tray(IdKitTray, Slots) }
<-
    +!new_tray [perform];

// All done, loop until another goal
+!loop [perform] : { True }
<-
    wait(2000),
    +!loop [perform];

// This belief is called when a human gets too close to the robot
+humanProximityViolation : { B humanProximityViolation}
<-
    remove_belief("humanProximityViolation");&\Comment{// for now remove perception
        belief using Java action}&

// Basic kitting action to pickup up gear from supply tray
+!take_part(Location) [perform] : { ~B grasped(_), B gripper("open")}
<-
    take_part(Location),
    *action_result(Result),,&\Comment{// wait for Java take/gear result}&
    -gripper("open"),&\Comment{// valid belief unless aborted}&
    +gripper("close"),
    +grasped(Gear),
    // check if fail or success
    +!check_action_result_take_part(Location, Result) [perform];

// When take_part result is true - no error
+!check_action_result_take_part(Location, "true") [perform] : { True } ;

// Dropped gear, if reachable try again, else abort
+!check_action_result_take_part(Location, "droppedGear") [perform] : { True}
<-
    reachable_gear(Location),
    *action_result(Result),
    +!check_action_result_take_part(Location, Result) [perform];

// If dropped gear is reachable, attempt pickup
+!check_action_result_take_part(Location, "reachableGear") [perform] : { True}
<-
    take_part(Location),&\Comment{// call Java action}&
    *action_result(Result), &\Comment{// wait for take\_part result}&
    // use check action corresponding to take_part result
    +!check_action_result_take_part(Location, Result) [perform];

// abort plan, undo take_part beliefs
+!check_action_result_take_part(Location, "abortGear") [perform] : { True }
<-
    -grasped(Gear),
    -gripper("close"),
    +gripper("open");


// place_part has no agile fault recovery
// the gwen goal plan puts a Gear into a slot location and returns T/F
+!place_part(Location, Gear) [perform] : { B grasped(Gear), B gripper("close")}
<-
        place_part(Location),
        *action_result(Result),
        -grasped(Gear),
        -gripper("close"),
        +gripper("open"),
        +!check_action_result_place_part(Location, Gear, Result) [perform];

// Successful place of gear into kit
+!check_action_result_place_part(Location, Gear, "true") [perform] : { True } ;

// Unsuccessful place of gear into kit, retry again
// No agile fault handling for place_part
+!check_action_result_place_part(Location, Gear, "false") [perform] : { True }
<-
        place_part(Location, Gear),
        *action_result(Result),
        +!check_action_result_place_part(Location, Gear, Result) [perform];
```

Appendix B: JAVA ENVIRONMENT

The GWENDOLEN environment planning actions are coded in Java. Java is an integral part of the GWENDOLEN programming. Upon invocation, the AIL parses the GWENDOLEN program as given in Appendix A for syntactic correctness, and then compiles plans from GWENDOLEN into Java code. Active goal and belief plans are evaluated in order every cycle until a true guard has been detected, and its associated plan is executed. Note, within each plan a goal can be desired, which invokes the associated goal/belief subplans which attempt to solve the goal. The use of a "*" belief within a plan is akin to a co-routine, in that the plan waits until the matching belief has been asserted.

B.1 CRCL Java Jar Support

The GWENDOLEN environment planning actions relies on CRCL Java language support. In addition several Java jar libraries were used to support use of CRCL, including:

- crcl4java-base – XML Java Compiler (XJC) autogenerated Java Architecture for XML Binding (JAXB) annotated classes corresponding to the CRCL schemas. Code found at https://github.com/ros-industrial/crcl.
- crcl4java-utils – class for sending and receiving CRCL classes over a TCP Socket, Pose math conversions, etc. Code found at https://github.com/ros-industrial/crcl.
- rcsjava – Java jar library that implements the NIST rcslib "posemath" classes in Java for

representing robot positions, rotations, and translations in a variety of coordinate systems. The original rcslib Java code repository can be found at https://github.com/usnistgov/rcslib

The source for these CRCL Java jar libraries has been bundled and included in the repository under the java folder found at https://github.com/autonomy-and-verification/gwendolen-crcl-kitting/tree/master/java. The CRCL java code was developed using Apache Netbeans [40] and Maven [41], so if you need to modify the CRCL XML schemas to recompile or regenerate the jar library files, you will need to install Netbeans which includes Maven. You can either use the Netbeans Integrated Development Environment (IDE) to generate the jar libraries or use the Windows batch script *win32build.bat* which uses Maven to generate the jar library. In either case, you will find the jar library under the target folder.

B.2 Gwendolen-CRCL Repository

The GWENDOLEN repository is based on the MCAPL code base found at https://sourceforge.net/projects/mcapl/. This code base has been adapted by removing much of the non-GWENDOLEN programming as possible. The use of Eclipse to run Gwendolen code remains. After cloning the gwendolen-crcl-kitting repository you will find the GWENDOLEN code in the ajpf2018/src/examples/gwendolen/crcl folder.

B.3 Gwendolen CRCL Java Communication

The primary connection to the real-world is the CRCL Client written in java.

The intelligent agent plan reasoning used a Java kitting model that can run either the CRCL Client locally or have the CRCL Client connect to a CRCL Server which controls a real robot. For evaluating GWENDOLEN planning code, the “loopback” CRCL Client was used as it handles all world modeling locally, which makes debugging easier. To enable this “loopback” functionality, the CRCL Client is provided a description of initial kitting world model and then programmatically determines object properties (object name, pose, location, type) by monitoring CRCL moves of gears between tray locations. CRCL Client “loopback” status reporting included inferring gear and kitting tray slot properties using this monitoring that are not reported by an actual CRCL Server. Within the local CRCL Client, the ability to derive first order reasoning about the kitting objects was coded and used for understanding open slots and free gears locations. 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.

The CRCL Client reads the *Globals.bLoopback*, which is a flag that when true does a "loopback" CRCL Client connection by accepting CRCL commands and simulating the actions that the robot would take. Real or simulated, the CRCL Client must have established the scene and gear/tray properties that is size, location, type, and for trays all the slots within the tray have been defined. Inferences can be performed locally or remotely to understand the underlying slot states of the trays, either open or contains a gear. Assuming the "loopback" capability, self-contained inferences determine the state of all the slots and gear locations, and is used to determine a sequence of operations to grasp a free gear that matches the kit open slot sizes. To achieve updates of gear locations, the gripper close and open operations are monitored. When a gripper is closed, the last moveto pose is used to find the closet gear, and that is the object that is being moved. When the gripper is opened then the gear is being placed into the kit open slot and the gear's location is now set to the open slot location.

When CRCL Server communication is established, then CRCL commands are sent, and the status contains the model and model inferences. Clearly, the inferences could be done locally in GWENDOLEN planner or be done at a lower level of control.

B.4 Gwendolen Java Files

The source for the GWENDOLEN evaluation Java code can be found in the `\ajpf\src\examples\crcl\kitting` folder that contains the following files shown in Table 4.

File	**Description**
CRCLClient.java	Client interface to CRCL which uses the JAXB class definitions to communicate XML messages to CRCL Server. If loopback flag set, CRCL connection channel ignored.
CShape.java	CShape describes the basic attributes of the world model objects defined with a name, type and location. Slots can also be shapes. Trays also have a contains "slot" items. In addition, each shape has inferences about its state - for example if a slot is “empty”, or contains a gear. Depending on the parent (kit or supply tray) this can be a free gear or a filled slot.
CShapes.java	CShapes is a container for the 1) shape instances in the kitting scene, and 2) feature definitions of the parts in the scene (gears, kits, supply vessel, and as well as the slots in a tray) such as centroid, size, containing slots, etc.
Globals.java	Globals is a wrapper for global flags and other general purpose utilities. All variables are public static.
KittingEnv.java	Inherited implementation of AILEnv. Implements AILEnv which is an interface to be satisfied by any environment that is to interact with the AIL classes. The class KittingEnv provides the implementation of an “action” GWENDOLEN call which connects GWENDOLEN plans to the physical “real” world. In essence GWENDOLEN does goal-directed planning using a belief system in conjunction with the real world beliefs in order to run a plan.
KittingInterface.java	Provides methods to perform or fake performing the kitting demonstration for the Fanuc LRMate robot. If live CRCL model status, then the robot will use the model inferences to determine the free gear and matching kit open slot.
TestCases.java	A smattering of methods to test the CRCL communication as well as lower level simulation validity.
rcs_robot.java	Basic definitions of the robot that will be manipulated. Contains definitions for robot base offset from the world origin (0,0,0), grasping offsets, bend pose for the robot to grasp the gears, and a approach/retract amount for grasp/release pre/post operation.
rcs_world.java	Model information about the objects in the scene. This is simplified for testing purpose.
robot.ail	AIL configuration data. See following discussion.
robot.gwen	GWENDOLEN program. See Appendix A.
run.bat	A Windows batch file to run the GWENDOLEN evaluation application. Note, most of the batch environment variables are derived from relative folder locations. However, the Eclipse java executable may have to be modified: `C:\Users\myusername\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.1.v20211116-1657\jre\bin\java.exe` where “myusername” is your home directory. The easiest way to determine the Eclipse Java compiler is to Navigate to *Run to Run Configurations* and then Click [Show Command Button] button to see actual command.

Table 4. GWENDOLEN CRCL File Descriptions

Listing 11 shows the *robot.ail* configuration file used by the AIL layer in the GWENDOLEN software architecture. The AIL configuration file requires the *mas.file* parameter to give the BDI program, which is the GWENDOLEN program. In Listing 11, the parameter *mas.file* is given as `mas.file=C:\\Users\\michalos\\Documents\\agilityperformancemetrics\\Gwendol`

```
en\\gwendolen-crcl-kitting-master\\ajpf2018\\src\\examples\\gwendolen\\crc
l\backslash\backslashkitting\\robot.gwen
```

where michalos is the home directory and gives the full path to the GWENDOLEN program file. Note, since we are testing on a Windows system, it is required to have double backslashes (i.e., \\) as the separator character between folders.

Listing 11. GWENDOLEN MAS Configuration File

```
mas.file = C:\\Users\\michalos\\Documents\\agilityperformancemetrics\\Gwendolen\\
    gwendolen-crcl-kitting-master\\ajpf2018\\src\\examples\\gwendolen\\crcl\\kitting\\
    robot.gwen
mas.builder = gwendolen.GwendolenMASBuilder
env = gwendolen.crcl.kitting.KittingEnv
log.info = ail.mas.DefaultEnvironment
log.fine = ail.semantics.operationalrules
log.warning = ajpf.MCAPLAgent
log.format = brief
```

The *env* parameter gives the GWENDOLEN Java environment file as an identifiers which is the folder path of the file separated by a period (i.e., "."). The other parameters are boilerplate and did not require special configuration.

B.5 Gwendolen Java Environment

The Gwendolen Java environment file *KittingEnv.java* contains the Java actions associated with the GWENDOLEN program plans. The Java method *executeAction* is the primary mechanism for GWENDOLEN plans to communicate with the environment and/or lower level control functionality. The Java method *executeAction* accepts a string and multiple arguments and then uses a branching mechanism to determine which GWENDOLEN action to execute. In theory the Java action execution could be an infinite loop, which would be a problem although GWENDOLEN is a multi-threaded execution environment. When the Java method *executeAction* has finished executing it returns an environment belief, generally a true or false belief string is returned to indicate success or failure. However, as was shown an action can return a more descriptive belief string to indicate various forms of status.

Listing 12. Java executeAction method

```
        public Unifier executeAction(String agName, Action act) throws AILexception {
               String actionname = act.getFunctor();
```

The $find\_gear(Slots)$ action was invoked from within the $+!kitting$ GWENDOLEN goal plan. The $find\_gear$ action gets a list of slots containing gears defined as $gear\_tray(IdGearTray, Size, Slots)$ beliefs that are known to be in a supply tray of the correct size. The $find\_gear$ action uses the CRCL API method $gearInSupplyTray$ with the slot as a gear name to see if gear is in supply tray.

If so we create a new AIL environment Predicate (i.e., gearPredicate = *new Predicate*(“*gear*″);) called “gear” and add the term slot (i.e., *gearPredicate.addTerm*(*slot*);) which is the name of gear (e.g., part_medium_gear17) as part of the predicate, which is then published to the AIL system as a belief (i.e., *addPercept*(*gearPredicate*);) . GWENDOLEN code $*gear(Gear)$ has been waiting on this belief to be satisfied before preceding with the next step in the goal plan.

Listing 13. Java findGear match inside executeAction method

```
if (actionname.equals("find_gear")) {
       ListTerm slots = (ListTerm) act.getTerm(0);
       for (Term slot : slots) {
              // No gear has empty assigned as a property
              // instead will check to see if in tray
              if(crcl.gearInSupplyTray(slot.toString()) > 0) {
                 gearPredicate = new Predicate("gear");
                 gearPredicate.addTerm(slot);
                 addPercept(gearPredicate);
                 break;
              }
   }
}
```

Finally, Javadoc for the CRCL that does not contain the status extension is available online and can be found here: http://ros-industrial.github.io/crcl/crcl/index.html.

Appendix C: CRCL KITTING MODEL ENHANCEMENTS

CRCL is designed to handle pick-and-place industrial robot applications. The CRCL goal is to abstract the many different communication protocols that control commercial, industrial robots. This section covers in detail the extensions to CRCL to communicate kitting world model status and inferences.

For our kitting plan evaluation, we extended CRCL status functionality to include reporting the kitting world model as well as derived kitting inferences. Such extension allowed a point-to-point communication scheme between world model sensors and the command/control/planning IA. This was done by extending CRCL XSD to incorporate model property reporting as well as all tray property inferences.

C.1 CRCL XSD Extension

Fortunately, in spite of XML complexity this extension was a mostly straightforward affair. We modified the XSD schemas found in the crcl4java-base project resource folder and used XJC in Netbeans to autogenerate JAXB annotated classes corresponding to the extended CRCL schemas. These JAXB classes were bundled into a Java JAR library that is included in the autonomy-and-verification github gwendolen-crcl-kitting repository library distributions. Code to rebuild or modify the CRCL XSD schemas is included in the gwendolen-crcl-kitting repository under the java/crcl4java-base folder with instructions to build the JAR file using Netbeans.

We found that reporting kitting model status as well as the kitting slot inferences could be done either at the CRCL Server (requiring the CRCL XSD status extensions) or as a loopback testing mechanism in the CRCL Client. In either case, CRCL must have some domain knowledge about the kitting world model in order to make inferences about trays and slots. However, this "loopback" ability to discern the kitting world and make inferences simplified GWENDOLEN evaluation. Thus, GWENDOLEN IA evaluation could be done in situ without controlling an actual robot and still be as effective.

Listing 14 illustrates the XSD definition of the CRCLStatusType which is used to report all status information be it from the robot, gripper, force torque sensor, or in our case an environment model. CRCLStatusType need not be monolithic, and status reporting can be configured to report more or less status information. For model status information, each CRCLStatusType can contain zero or more ModelStatus elements to report on logical models, which we examine further.

Listing 14. CRCL Extension to handle list of model parameters

```
<xs:complexType name="CRCLStatusType">
...
   <xs:sequence minOccurs="0" >
       <xs:element name="ModelStatus" maxOccurs="unbounded" type="ModelsStatusType"/>
   </xs:sequence>
```

Listing 15 gives an overview of the XSD for the ModelsStatusType, which reports on the logical models in the environment. The basic logical model is name and pose. Included in the properties are the inferred properties about a model that are included as part of the model status. An instance of ModelsStatusType has the following elements for Name; Pose which is defined as a CRCL pose type (xyz rotation: x rotation, z rotation); an optional Twist; an optional Wrench; and finally a sequence of Properties, defined as a MapType XSD type.

Listing 15. ModelsStatusType Extension to CRCL Status

```
<xs:complexType name="ModelsStatusType">
    <xs:sequence>
        <xs:element name="Name" type="xs:string"/>
        <xs:element name="Pose" type="PoseType"/>
        <xs:element name="Twist" type="TwistType" minOccurs="0"/>
        <xs:element name="Wrench" type="WrenchType" minOccurs="0"/>
        <xs:sequence minOccurs="0" >
            <xs:element name="Properties" type="MapType"/>
        </xs:sequence>
    </xs:sequence>
</xs:complexType>
```

Listing 16 gives an overview of the XSD for the MapType. MapType Properties describe either a slot or a gear parent. For a slot, name, type, location, and state are given. Location is the reoriented position based on the orientation of the parent tray. For a gear, the property gives the parent tray name hosting the gear (if one) and the slot in the parent tray. Although all gears are supposed to be within a slot in a tray, in the case of a dropped gear, the gear could fall and land in a reachable position within the robot workspace.

Listing 16. XSD Map Type to define Kitting Properties

```
        <xs:complexType>
            <xs:sequence>
                <xs:element name="item" type="MapItemType" minOccurs="0" maxOccurs="
                    unbounded"/>
            </xs:sequence>
        </xs:complexType>
        <xs:unique name="item">
            <xs:selector xpath="item"/>
            <xs:field xpath="key"/>
        </xs:unique>
    </xs:element>

    <xs:complexType name="MapType">
       <xs:sequence>
            <xs:element ref="map"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="MapItemType">
        <xs:sequence>
            <xs:element name="key" type="xs:string"/>
            <xs:element name="value" type="xs:string"/>
        </xs:sequence>
     <xs:attribute name="name" use="required"/>
    </xs:complexType>
```

C.2 CRCL Inference Explanation

The CRCL Client 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. The CRCL Client was responsible for reporting model properties that lead to generating inferences about the kitting models (e.g., a gear located in a supply tray slot).

For each kit, supply vessel, and gear, knowledge inferences are produced by studying the relationship of the gears and trays. The kitting world model is known à priori. As an example, we will show a medium kitting tray and how its slots are defined as offsets. Figure 3 shows the physical relationships for the one large gear and two medium gears kitting tray.

In order to make inferences, CRCL must understand the relationship between each tray type and its slots. Thus, the CRCL Client contains physical descriptions of all trays (both supply and kit) types as well as the relationship to contained slots. Since the kitting model is maintained in Gazebo, we have used Standard Tessellation Language (STL) to define kitting model object and for each object defined the centroid is defined at the minimum Z and with a centered (x,y) on this plane. Note, not all the kitting world model objects were so positioned, so we reoriented the STL object file to adhere to this scene positioning. We adjusted all kitting world model objects to use this representation so that we understand how to place an object on a flat surface, as well as understanding where the object is when Gazebo reports a kitting model location. Further, without a centroid at the bottom of the part, a gear in a Gazebo physics based simulation will bounce if incorrectly placed into a tray slot.

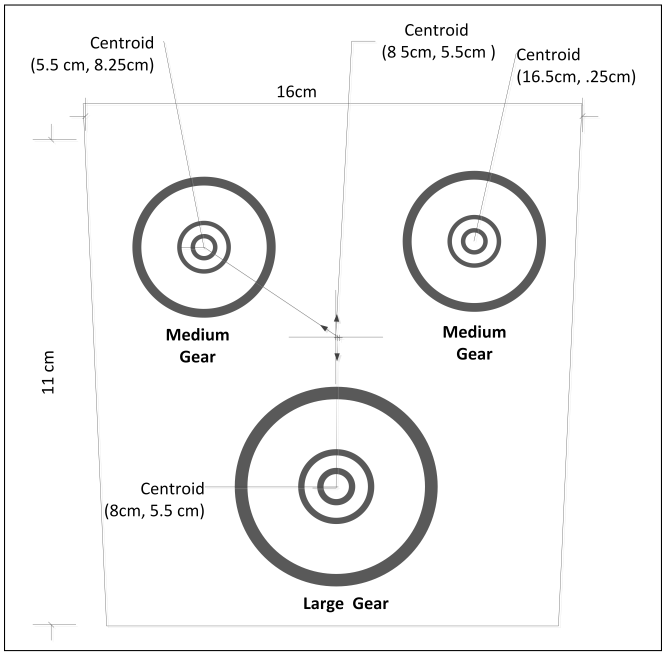


Fig. 3. Kitting Tray with Slot Offsets

Below is a diagnostic dump of the inferred knowledge. The gear inferences are produced by studying the location of a gear and comparing to all the locations of each tray's slots. If close enough (we assume a small error), the gear is inferred to be resident in the tray slot. Thus for each gear in our kitting demo after we have moved gear part_medium_gear17, we have the following inferences about all the gears. This unto itself is generally not useful, as we are either trying to find an EMPTY slot in a kit or a matching gear size in a supply tray to the empty kitting slot. However, if we are trying to recover from an adversity, such as dropping the gear, the location of the gear can be important as if the gear is reachable by the robot, it can be picked up even if not in a gear tray. Listing 17 gives a diagnostic dump of gears 17–20 and 22–23 inferred knowledge. In the listing, the gear name is followed by its location given as pose in world coordinate space (x,y,z and xyzw quaternion), followed by what tray the gear is located. Note, gears have inferences (but are more informative and not as important for solving the kitting problem) that give the tray (parent) and the tray slot where the gear is located. It is easier to scan trays for an appropriate gear, than search each gear for an appropriate tray match.

Listing 17. CRCL gear inferences after moving part_medium_gear17

```
part_medium_gear17 at 0.4564, -1.0120, 0.9200 0.0000, 0.0000, 0.0000, 1.0000,
     In: medium_gear_vessel16(slot1)

part_medium_gear18 at 0.1500, -1.2000, 0.9200 -0.0010, -0.0000, 0.0170, 0.9999,
     In: medium_gear_vessel16(slot2)

part_medium_gear19 at 0.1500, -1.2800, 0.9200 0.0010, -0.0030, 0.0180, 0.9998,
     In: medium_gear_vessel16(slot4)

part_medium_gear20 at 0.2300, -1.2800, 0.9200 -0.0020, 0.0010, 0.0120, 0.9999,
     In: medium_gear_vessel16(slot3)

part_large_gear22 at 0.3900, -1.2100, 0.9200 0.0020, -0.0020, 0.0160, 0.9999,
     In: large_gear_vessel21(slot2)

part_large_gear23 at 0.3900, -1.3200, 0.9200 -0.0020, -0.0010, 0.1880, 0.9822,
     In: large_gear_vessel21(slot1)
```

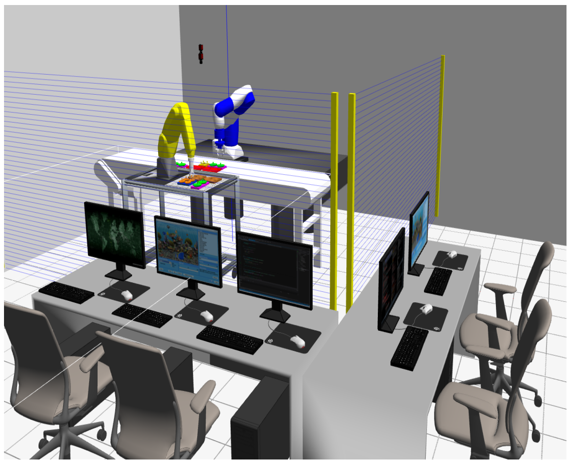
For tray inferences, such as for kits or supply vessels, the inferences provide the status of each of the trays slots – either containing a name slot or “empty”. For a kit or supply tray (e.g., kit= kit_m2l1_vessel14 while gear supply vessel= medium_gear_vessel16) the Gazebo derived model knowledge gives the location (as well as orientation) of the kitting object. Inferences follow and are indented to display the inferred knowledge about the slots contained in the tray and their state. The state shows whether the tray is empty or occupied. If occupied, the name of the gear in the slot (e.g., part_medium_gear17) is given. If unoccupied, "empty" is designated as the name. This information is inferred by knowing the reoriented location of the slot (based on the tray orientation) and whether there is a gear close to this location. If a gear is nearby, it is inferred to be in the slot. If no gear is near, then the slot is “empty”. Listing 18 gives a diagnostic dump of CRCL tray slot inferences after a gear move. Of note, each slot inference also contains the reoriented location of the slot in world coordinates so this is helpful knowledge. All shapes are represented in world coordinates but must be transformed into the robot coordinate system to be useful to a CRCL Server robot.

Listing 18. CRCL tray slot inferences after gear move

```
kit_m2l1_vessel14 at 0.4000, -1.0500, 0.9200 0.0000, 0.0000, -0.7200, 0.6940,
     slot1 part_medium_gear part_medium_gear17 ( 0.4564,-1.0120, 0.9200)
     slot2 part_medium_gear empty ( 0.4535,-1.0920, 0.9200)
     slot3 part_large_gear empty ( 0.3600,-1.0485, 0.9200)

kit_m2l1_vessel15 at 0.1800, -1.0500, 0.9200 0.0000, 0.0000, -0.7200, 0.6940,
     slot1 part_medium_gear empty ( 0.2364,-1.0120, 0.9200)
     slot2 part_medium_gear empty ( 0.2335,-1.0920, 0.9200)
     slot3 part_large_gear empty ( 0.1400,-1.0485, 0.9200)

medium_gear_vessel16 at 0.1900, -1.2400, 0.9200 0.0000, 0.0000, 0.0170, 0.9999,
     slot1 part_medium_gear empty ( 0.2282,-1.1991, 0.9200)
     slot2 part_medium_gear part_medium_gear18 ( 0.1491,-1.2018, 0.9200)
     slot3 part_medium_gear part_medium_gear20 ( 0.2309,-1.2782, 0.9200)
     slot4 part_medium_gear part_medium_gear19 ( 0.1518,-1.2809, 0.9200)

large_gear_vessel21 at 0.3900, -1.2600, 0.9200 0.0000, 0.0000, 0.7210, 0.6930,
     slot1 part_large_gear part_large_gear23 ( 0.3922,-1.3150, 0.9200)
     slot2 part_large_gear part_large_gear22 ( 0.3878,-1.2050, 0.9200)
```

Appendix D: ROS

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.


(a) Gazebo Simulation of APRS Kitting

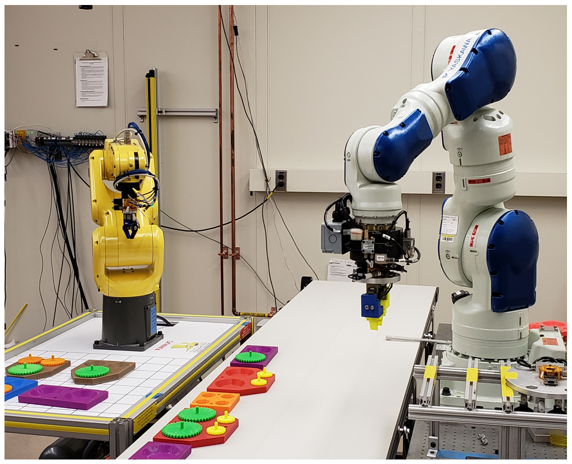

(b) Physical Real World APRS Kitting Layout

Fig. 4. NIST Agility Performance of Robotic Systems

There are several CRCL Server GitHub repositories that implement a CRCL Server with a ROS and Gazebo back-end. However, there is only one GitHub repository that supports CRCL and the kitting world model enhancements [42]. This repository has Gazebo, ROS, and CRCL code to model the NIST APRS laboratory setup for real world kitting shown in Figure 4b. Figure 4a shows the Gazebo simulation of the APRS laboratory, which only supports a Fanuc LRMate robot at this time. You can add the Motoman robot by having the gazebo_ros_api plugin launch the Motoman robot. This repository has only been tested for a Ubuntu 16.04 and 18.04 Linux distribution. For compatibility issues with existing ROS modules and GWENDOLEN implementations (i.e., deprecated ROSBridge Version 1), the implementation used the ROS Kinetic version and Gazebo version 7. Note, it is not difficult porting the Windows GWENDOLEN Java code to Ubuntu Eclipse and was originally done by the authors.