RoboPlanner: Towards an Autonomous Robotic Action Planning Framework for Industry 4.0

Ajay Kattepur, Balamuralidhar P

To cite this version:
Ajay Kattepur, Balamuralidhar P. RoboPlanner: Towards an Autonomous Robotic Action Planning Framework for Industry 4.0. 2019. hal-02284497

HAL Id: hal-02284497
https://hal.archives-ouvertes.fr/hal-02284497
Submitted on 11 Sep 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
RoboPlanner: Towards an Autonomous Robotic Action Planning Framework for Industry 4.0

Ajay Kattepur & Balamuralidhar P
Embedded Systems and Robotics, TCS Research & Innovation, India.
ajay.kattepur@tcs.com

Abstract

Autonomous robots are being increasingly integrated into manufacturing, supply chain and retail industries due to the twin advantages of improved throughput and adaptivity. In order to handle complex Industry 4.0 tasks, the autonomous robots require robust action plans, that can self-adapt to runtime changes. A further requirement is efficient implementation of knowledge bases, that may be queried during planning and execution. In this paper, we propose RoboPlanner, a framework to generate action plans in autonomous robots. In RoboPlanner, we model the knowledge of world models, robotic capabilities and task templates using knowledge property graphs and graph databases. Design time queries and robotic perception are used to enable intelligent action planning. At runtime, integrity constraints on world model observations are used to update knowledge bases. We demonstrate these solutions on autonomous picker robots deployed in Industry 4.0 warehouses.

1 Introduction

Advances in robotics, cyber-physical systems and industrial automation has come to the forefront with Industry 4.0 [Lasi et al., 2014], with the following key requirements:

1. Interoperability: Machines, Internet of Things (IoT) [Greengard, 2015] enabled devices and humans connected and coordinating with each other.

2. Information transparency: Physical systems enhanced with sensor data to create added value information systems.

3. Technical Assistance: Use of intelligent devices to aid in informed decision making. Robotic automation may be identified to perform repetitive, unsafe or precise tasks.

4. Decentralized Decisions: The ability of such systems to make autonomous decisions; only critical cases will involve human intervention.

A fundamental characteristic required in Industry 4.0 deployments is the ability of autonomous robotic devices to self-configure in dynamic goal and deployment conditions. Autonomic computing [Huebscher and McCann, 2008] models have been proposed to create self-aware robotic systems that respond to both high level goals as well as external stimuli [Faniyi et al., 2014]. This has led to the development of Cognitive Robotic Architectures [Levesque and Lakemeyer, 2010][Beetz et al., 2010], that are at the intersection of robotics, IoT and Artificial Intelligence [Russell and Norvig, 2015].

Cognitive robots are able to intelligently execute tasks based on high level goals, dependent on world model knowledge and sensory perceptions to generate efficient actions [Levesque and Lakemeyer, 2010]. In order to be deployed in dynamic Industry 4.0 environments, the robots must be autonomous and adaptive to runtime changes. Given a high level task such as “pick ball from warehouse rack”, the autonomous robot must identify appropriate action plans to perform this task. As the robots are intended to be learning world models, knowledge bases are needed to populate information about the world, object, perception and action sequences needed. Any runtime anomalies are dealt with through further queries and eventual exception handling.

Distilling these high level requirements, an autonomous planning module for robots should include: (i) Knowledge Bases that efficiently capture relationships between world models, objects, robot actions and tasks (ii) Action Plans that are efficiently decomposed from a high level goal task; this involves querying the knowledge base as well as triggering perceptions in case of knowledge mismatch (iii) Techniques to Reconfigure actions at runtime, when plans cannot be executed due to constraints (iv) Rules for consistent Updates to the world model, which allows multiple robots to coordinate or analyze exceptions during execution. While individual modules may have been developed in the robotic and embedded software communities, integrating these features into a common framework for industrial deployments remains a challenge.

In this paper, we propose RoboPlanner, a structured technique to generate design time action plans for autonomous robots. In order to enable autonomy in de-
deployments, we integrate knowledge bases, design time action planning and runtime adaptation modules. Knowledge representation and queries are enabled using efficient graph database technologies [Angles and Gutierrez, 2008]. Design time action plans as provided using the formal concurrent programming knowledge Orb [Kitchin et al., 2009], that allows structured composition of action plans. To take care of runtime adaptation, we provide general rules for triggering perception and exception handling. An integrity check is also provided to update the graph database with runtime knowledge. This framework is implemented over a realistic industrial use case involving autonomous picking robots employed in Industry 4.0 warehouses [Wurman et al., 2008].

Principal contributions of this paper:

1. RoboPlanner Knowledge Base module that formally models robotic world models, capabilities, object descriptions and task templates.
2. RoboPlanner Action Planner that uses design-time queries/updates to knowledge graph databases, including exception handling.
3. RoboPlanner Run time simulation, adaptation and performance analysis of action plans using graph queries. This may be used to generate executable task templates for physical robots.
4. RoboPlanner integrity checks for runtime updates to the knowledge base.
5. Demonstration of the framework over an Industry 4.0 warehouse automation task.

The rest of this paper is organized as follows: Section 2 provides an overview of Industry 4.0 warehouse automation and the autonomous robots deployed in them. The RoboPlanner modules are described in Section 3. Details of knowledge base representation using graph databases are covered in Section 4. Section 5 describes the techniques used for action plan generation. Simulation, performance analysis and knowledge updates in autonomous robot deployments are presented in Section 6. The paper ends with related work and conclusions.

2 Warehouse Automation

In this section, we introduce Industry 4.0 warehouse automation tasks that may be fulfilled by autonomous robots. A high level description of autonomous robots is also introduced, which is used to build the RoboPlanner framework in proceeding sections.

2.1 Industry 4.0 Warehouses

Industrial warehouses are employed as buffers in supply-chains to maintain excess product, when there are variations in procurement/customer demand [Bartholdi and Hackman, 2016]. Considerable effort has gone in reducing the stowing and procurement times in such warehouses, with automated picking robots [Zhang and et al., 2016] being throughput of pick & place tasks.

Fig. 1 presents a high level view of operations taking place in automated warehouses. Once a delivery order is received, the products are procured from the warehouse. As shown in Fig. 1, autonomous Picker robots (such as KUKA KMR Quantec1) are being proposed for Industry 4.0 automating pick & place tasks. The robots are intended to be autonomous, with adaptation seen for varying pick-up locations, product dimensions and rates of procurement. When the required products are procured, they are collated and checked for final packing and product shipment.

In order to successfully integrate robotic entities into complex industrial deployments, it is crucial to develop a unified modeling framework for autonomous robots.

2.2 Autonomous Robots

To model the robotic components in warehouses, we make use of the Autonomous Robot abstraction, inspired by intelligent agents [Russell and Norvig, 2015]. Typical activities, for instance with a pick & place robot in a smart warehouse, include:

1. Goals: Understanding goals of each task and subtask, such as, placing correct parts into correct bins within the given time constraints.
2. Perception: Object identification and obstacle detection using camera and odometry sensors that sense the environment. This aids the robot in object detection and identification. Robot location, view and environment may also be perceived.
3. Actions: Identifying granular actionable subtasks, such as, moving to particular location, picking up parts of orders or sorting objects. Constraints may be placed on the robot capabilities, motion plans and accuracy in performing such actions.
4. Knowledge Base: Using domain models of the world for goal completion, such as warehouse environment maps, rack type and product features. The robot capabilities and necessary algorithms should enable completion of goals.

Algorithm 1 presents an overview of an intelligent robot’s perception and action via a Knowledge Base [Russell and Norvig, 2015]. The knowledge base coordinates the appropriate action in relation to an individual robot’s perception. The knowledge base should also include descriptions of domain ontology, task templates, algorithmic implementations and resource descriptions.

1https://www.kuka.com
To integrate the above elements into robotic interactions for Industry 4.0, we propose the *RoboPlanner* autonomous architecture framework.

3 RoboPlanner Modules

In this section, we provide details about the various modules to be integrated within *RoboPlanner*. These modules cover the principal requirements of cognitive robotic architectures [Levesque and Lakemeyer, 2010][Beetz et al., 2010], including knowledge representation, action planning, reconfiguration and knowledge updates. Fig. 2 provides an overview of the modules that are integrated within *RoboPlanner*:

- Design Time Action Planning Module: This module is responsible for generating efficient action plans, when input with a high level goal. The module decomposes the goal into atomic tasks, and applies workflow specification languages (such as Orc [Kitchin et al., 2009]) to complete the goal task. Action planning involves querying the *Knowledge Graph Database Module* to ascertain requirements for goal completion. *Robot perception* may also be triggered to acquire further information for action planning.

- Knowledge Graph Database Module: An integral part of all autonomous/cognitive robotic architectures is the knowledge base. We model this using graph databases [Angles and Gutierrez, 2008], that maintain relationships between data in a graphical form. Entities such as the world model, robotic algorithms and task templates are stored in the database. The knowledge database is queried both at design time for action generation and at runtime for knowledge updates.

- Runtime Execution Module: The action plans are executed by one or multiple autonomous robots to complete the task. Translation of the action plan to a robot specific middleware language such as ROS\(^2\) may be done. The execution module may be aided by robotic perception. Knowledge that is gained during the execution is to be updated to the graph knowledge database, after satisfying some integrity constraints.

- Adaptation Monitoring Module: This module monitors runtime deployments of intelligent robots to estimate plan completion. While robotic perception may be used to aid in unforeseen circumstances, more severe exceptions may require re-planning. Performance degradation (leading to non-completion of plans), may also trigger re-planning. Knowledge of instances that trigger re-planning are learnt and updated.

The following sections dive further into the modeling and implementation of these modules.

4 Robotic Knowledge Base

The robotic knowledge base is modeled using property graphs, with data stored in graph databases. Queries using the Gremlin graph query language are also studied.

4.1 Knowledge Graphs

In order to model knowledge bases inherent in intelligent automation, we make use of property graphs [Angles and Gutierrez, 2008]. Property graphs are attributed, labeled, directed graphs. This is an alternative to semantic ontologies [Grimm et al., 2007] and tuple datastores that are in implementations such as Knowrob [Tenorth and Beetz, 2013] and CRAM [Beetz et al., 2010]. Our knowledge base has the following knowledge graphs included:

- World Models: Describes the environment map and layout, including object locations.
- Object Templates: Describes the target objects of interest, including shape, size, colour and location.
- Robot Capabilities: Provides robot models, capabilities, sensors and actuators that are integrated to perform tasks.
- Robotic Algorithms: Navigation, manipulation and task allocation algorithms that are used within robotic actions.
- Task Templates: High level task requirements and corresponding outputs are provided.

Fig. 3 provides the property graph models for world models, task templates, object templates, robot capabilities and robot algorithms. To describe properties between edges, we limit ourselves to four relations: isOffType, hasProperty, requires and produces. isOffType provides hierarchical sub-class relationships; hasProperty extends property descriptions using key-value pairs; requires provides pre-conditions to extract knowledge from the graph; requires provides post condition effects of executing the node. These relationships may be queried to extract information from the knowledge base.

Fig. 3a provides the capabilities of a *Pick Robot* that *Robot Model*, *Capabilities*, *Perception*: it requires *Target*, *World Model*, *Algorithms* and produces the *Pick*, *Place Actions*. Algorithms necessary for the robotic executions are provided in Fig. 3d, with path planning, image template matching and grasp manipulation algorithms included. Explicit definitions of

---

\(^2\) [http://www.ros.org/](http://www.ros.org/)
each task is provided in Fig. 3b, for instance with the Place task, which requires World Model, Target Object, Picker Robot and produces Placed Object. Fig. 3c provides an example of the Warehouse world model, which hasProperty Map and Object. In order to extract the property of Object Location requires a Map of the area. Fig. 3c provides properties of objects in the world model, including their Location, Shape and Contour Map. Note that the property graph modeling approach provides extensibility and reuse of information across multiple autonomous robotic deployments.
4.2 Graph Database Queries

Semantic ontologies typically store data in tuple data-stores that reduce expressivity provided in graph representations [Angles and Gutierrez, 2008]. Scalability is another hindrance in representation, update and query of large ontologies. Graph databases are emerging as an appropriate tool to model interconnectivity and topology of relationships among large knowledge data sets [Angles and Gutierrez, 2008]. Principal advantages include: (i) Being able to keep all the information about an entity in a single node and show related information by arcs connected to it; (ii) Queries can refer directly to this graph structure, such as finding shortest paths or determining certain subgraphs; (iii) Graph databases provide efficient storage structures for graphs, thus reducing computational complexity in operations. Graph databases are also emerging as high-performance back end stores when making use of complex dialogue and chatbot engines [M. Maro and Origlia, 2017].

To implement the property graphs in Section 4.1, we make use of the multi-modal OrientDB database. Ori- entDB uses a generic vertex persistent class V and a class for edges E. Unlike ontologies that store data using triple stores, graph databases maintain the graphical structure with vertices and edges. In the graph data model, nodes are physically connected to each other via pointers, thus enabling complex queries to be executed faster and more effectively than in a relational database 

An example graph database (of the World Model in Fig. 3e) with vertices, edges and properties in OrientDB is presented below:

```
gremlin> g.V.map
  =>{(Name=Objects, Properties=ObjectProperties, Location=ObjectLocation),
     (Name=Warehouse),
     (Map=Name=Map, Rack=RackConfig, Layout=WarehouseLayout, Aisles=AislesConfig)}
gremlin> g.E
  =>'eV[10:0]'.has('Name', 'Warehouse').out('hasProperty').map
  =>{(v[#10:0], e[#26:0][#10:0-isOfType->#9:0],
      v[#10:0], null),
     (v[#10:0], e[#29:0][#10:0-hasProperty->#11:0],
      v[#11:0], null)}
```

In order to query this graph, we use Gremlin, a domain-specific (DSL) open source programming language focusing on graph traversal and manipulation. The following types of queries may be made:

1. **Filtering**: Filter out vertices or edges according to given property labels. For instance, the query may `g.v().has('Name', 'Warehouse').out('hasProperty').map` matches the vertex with property key–value pair (Name, Warehouse), output edge with property hasProperty and produces an output of the vertices.

2. **Complex Queries**: Queries can combine multiple vertices, edges and properties. Queries can also provide range or equality constraints to numeric property values. For instance, the complex query `g.V.has('Name', 'Warehouse').out('hasProperty').map` matches the vertex with property key–value pair (Name, Warehouse), output edge with property hasProperty and produces an output of the vertices.

3. **Graph Traversal**: Another advantage of storing data using graph databases is the ability to traverse graphs. For instance, the query `g.v().outE.inV.name.path` traverses the output edges (outE) of a vertex, and provides the path traversed.

While we have made use of Gremlin as the language for explicit graph database querying, this can also be a backend for an efficient dialogue/chatbot implementation [M. Maro and Origlia, 2017]. Questions such as “Where is the target?” or “What are the target’s properties?” can be translated into efficient knowledge base queries as defined above. It is of interest to translate this knowledge to efficient action plans for the robot to act upon, which is explored next.

5 Action Plan Generation

In order to study the design time action planning module, we formalize the interaction between the planner and knowledge base. An overview of the concurrent programming language Orc is also provided, that is later used to simulate action planning.

5.1 Orc Language

In order to implement robotic action plans, we make use of the formal specification language Orc. The Orc concurrent programming language is grounded on formal process-calculi to specify complex distributed computing patterns [Kitchin et al., 2009]. The execution of programs in Orc makes use of Expressions, with the atomic abstraction being a site. To create complex expressions based on site invocations, Orc employs the following Combinators:

- **Parallel Combinator (|)**: Given two sites s1 and s2, the expression s1 | s2 invokes both sites in parallel.
- **Sequential Combinator (>>)**: In the expression s1 >> s2 (shorthand s1 s2), site s1 is evaluated

---

1. https://orientdb.com/
initially, with every value published by $s_1$ initiating a separate execution of site $s_2$.

- **Pruning Combinator ($\times < x \times$)**: In the expression $s_1 \times < x \times s_2$ (shorthand $s_1 \times s_2$), both sites $s_1$ and $s_2$ execute in parallel. If $s_2$ publishes a value, that value is bound to $x$ and the execution of $s_2$ is terminated.

- **Otherwise Combinator (:)**: The expression $s_1 : s_2$ first executes site $s_1$. If $s_1$ publishes no value and halts, then $s_2$ is executed instead.

The `val` declaration in Orc binds variables to values. The `def` declaration defines a function. Orc further contains built-in sites incorporating distributed computing paradigms such as channels, semaphores and synchronization primitives (further details available in the Orc website⁵).

### 5.2 Action Planning Module
As specified in Fig. 2, action planning involves interacting with the knowledge base to efficiently plan manipulation, navigation and task planning actions. However, perception and exception handling must also be built in to take care of insufficient knowledge.

To formalize the process of generating action plans required to satisfy goals, we present Algorithm 2. Given an input goal (e.g., pick ball from rack using picker robot), the first step (lines 3, 4 in Algorithm 2) is to verify and subdivide goals from the task descriptions available (pick target, being an atomic subgoal). For each of these subgoals, there are pre-conditions to be satisfied (lines 6–8 in Algorithm 2): subgoals require (actions, targets), actions require (targets, object attributes, capabilities), targets require (object attributes). The object attributes of interest (environment rack, locations) can either be derived from the world model knowledge base or by querying robot perception (robot sensor observation and interpretation, environment point cloud). The target of interest (ball) can either be identified from the object templates knowledge base or by querying robot perception (robot sensor observation and interpretation, perception algorithms). The capability to complete goal (robot model, arm length, battery state) is also extracted from the robot capability knowledge base. Finally, the action (pick ball) needed to satisfy the subgoal is derived, dependent on the specified target, object attributes and capability (line 12 in Algorithm 2). The actions consist of both navigation (path planning) and manipulation (grasping, lifting) procedures. This process is used iteratively for each subgoal to derive the action plan needed to enact the goal (line 13, 14 in Algorithm 2). In case there are Exceptions observed within the subgoal planning, re-planning is triggered.

An example of such an action plan in presented in Fig. 4, wherein the high level input task of: picker | pick | ball | rack is decomposed iteratively to complete the task. Queries to the knowledge base enable generating

<table>
<thead>
<tr>
<th>Algorithm 2: Generating Action Plans for Goals via Knowledge Bases.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong>: Input Goal; Knowledge Base[World Model, Object Templates, Task Descriptions, Robot Capability, Algorithms];</td>
</tr>
<tr>
<td><strong>Output</strong>: Action Plan;</td>
</tr>
<tr>
<td><strong>Goal</strong>: Verify(Input Goal, Knowledge Base[Task Descriptions]);</td>
</tr>
<tr>
<td><strong>Subgoals</strong>: Decompose(Goal, Knowledge Base[Task Descriptions]);</td>
</tr>
<tr>
<td><strong>for each Subgoal do</strong></td>
</tr>
<tr>
<td><strong>(Action?, Target?)</strong> ← Requirements(Subgoal);</td>
</tr>
<tr>
<td><strong>(Target?, Object Attributes?, Capability?)</strong> ← Requirements(Action?);</td>
</tr>
<tr>
<td><strong>Object Attributes?</strong> ← Requirements(Target?);</td>
</tr>
<tr>
<td><strong>if</strong> Object Attributes? is a member of Knowledge Base[World Model] <strong>then</strong></td>
</tr>
<tr>
<td><strong>Object Attributes?</strong> ← Query(Object Attributes?, Knowledge Base[World Model]);</td>
</tr>
<tr>
<td><strong>else</strong></td>
</tr>
<tr>
<td><strong>if</strong> Object Attributes? can be obtained by Perception <strong>then</strong></td>
</tr>
<tr>
<td><strong>Object Attributes?</strong> ← Perception(Object Attributes?, Knowledge Base[World Model, Robot Capability, Perception Algorithms]);</td>
</tr>
<tr>
<td><strong>else</strong></td>
</tr>
<tr>
<td><strong>Exception</strong> ← Object Attributes?</td>
</tr>
<tr>
<td><strong>if</strong> Target? is a member of Knowledge Base[Object Templates] <strong>then</strong></td>
</tr>
<tr>
<td><strong>Target?</strong> ← Query(Target?, Knowledge Base[Object Templates]);</td>
</tr>
<tr>
<td><strong>else</strong></td>
</tr>
<tr>
<td><strong>if</strong> Target? can be obtained by Perception <strong>then</strong></td>
</tr>
<tr>
<td><strong>Target?</strong> ← Perception(Target?, Knowledge Base[World Model, Robot Capability, Perception Algorithms]);</td>
</tr>
<tr>
<td><strong>else</strong></td>
</tr>
<tr>
<td><strong>Exception</strong> ← Target?</td>
</tr>
<tr>
<td><strong>if</strong> Capability satisfies Action <strong>then</strong></td>
</tr>
<tr>
<td><strong>Action</strong> ← Query(Action?, Target, Object Attributes, Capability, Knowledge Base[Navigation/Manipulation Algorithms]);</td>
</tr>
<tr>
<td><strong>else</strong></td>
</tr>
<tr>
<td><strong>Exception</strong> ← Action?</td>
</tr>
<tr>
<td><strong>if</strong> Exception is null <strong>then</strong></td>
</tr>
<tr>
<td><strong>Action Plan</strong> ← Update(Action?);</td>
</tr>
<tr>
<td><strong>else</strong></td>
</tr>
<tr>
<td><strong>Trigger Re-planning of Subgoal</strong></td>
</tr>
<tr>
<td><strong>return</strong> Action Plan satisfying Input Goal;</td>
</tr>
</tbody>
</table>

⁵https://orc.csres.utexas.edu/
6 Simulation and Analysis

In this section, we provide an end-to-end simulation of the design time planning and runtime adaptation process, with further analysis on performance aspects. Further constructs to ensure graph database integrity with knowledge base updates are provided.

6.1 Action Planning Simulation

Given a high level goal task such as “Pick Ball from Rack using Picker Robot”, the first step is to decompose goals into appropriate sub-tasks. The tasks are mapped to appropriate Knowledge Bases (using the member function in Orc), depending on whether they represent actions, targets, robotic components or properties. The following code provides a map of the atomic terms to individual knowledge base elements (Line 12 in Knowledge Resolution). For instance, the term rack is located as a member of the World_model knowledge base (Line 15 in Knowledge Resolution).

<table>
<thead>
<tr>
<th>Artifacts</th>
<th>Instances</th>
</tr>
</thead>
<tbody>
<tr>
<td>world model</td>
<td>warehouse</td>
</tr>
<tr>
<td>object template</td>
<td>ball</td>
</tr>
<tr>
<td>robot capabilities</td>
<td>picking</td>
</tr>
<tr>
<td>perception</td>
<td>depth camera</td>
</tr>
<tr>
<td>task templates</td>
<td>delivery</td>
</tr>
<tr>
<td>robot algorithms</td>
<td>localization &amp; mapping</td>
</tr>
<tr>
<td>target</td>
<td>ball</td>
</tr>
<tr>
<td>action</td>
<td>pick</td>
</tr>
<tr>
<td>goal</td>
<td>action?</td>
</tr>
</tbody>
</table>

Once the appropriate knowledge base elements are recognized, Gremlin queries are used to obtain dependencies from the Graph database. We assume that the knowledge base is pre-populated with property graphs as described in Section 4. Terms used in Fig. 3, such as hasProperty and requires are used in conjunction with Gremlin graph database filtering and complex queries, to populate local robotic knowledge bases (Lines 9-20 in Knowledge Query). While we represent this as explicit queries, alternate implementations may use dialogue engines to extract necessary information from the knowledge base via question-answers [A. Bordes and Weston, 2017][M. Maro and Origlia, 2017]. We make use of the def class declaration that allows us to implement sites within Orc [Kitchin et al., 2009], which provides encapsulation similar to classes in object-oriented programming, we make use of the Ref site in Orc, that creates a rewritable storage location. The following Orc code presents these aspects:

--- Knowledge Query ---

Once the appropriate knowledge base elements are recognized, Gremlin queries are used to obtain dependencies from the Graph database. We assume that the knowledge base is pre-populated with property graphs as described in Section 4. Terms used in Fig. 3, such as hasProperty and requires are used in conjunction with Gremlin graph database filtering and complex queries, to populate local robotic knowledge bases (Lines 9-20 in Knowledge Query). While we represent this as explicit queries, alternate implementations may use dialogue engines to extract necessary information from the knowledge base via question-answers [A. Bordes and Weston, 2017][M. Maro and Origlia, 2017]. We make use of the def class declaration that allows us to implement sites within Orc [Kitchin et al., 2009], which provides encapsulation similar to classes in object-oriented programming, we make use of the Ref site in Orc, that creates a rewritable storage location. The following Orc code presents these aspects:

--- Knowledge Query ---

Once the appropriate knowledge base elements are recognized, Gremlin queries are used to obtain dependencies from the Graph database. We assume that the knowledge base is pre-populated with property graphs as described in Section 4. Terms used in Fig. 3, such as hasProperty and requires are used in conjunction with Gremlin graph database filtering and complex queries, to populate local robotic knowledge bases (Lines 9-20 in Knowledge Query). While we represent this as explicit queries, alternate implementations may use dialogue engines to extract necessary information from the knowledge base via question-answers [A. Bordes and Weston, 2017][M. Maro and Origlia, 2017]. We make use of the def class declaration that allows us to implement sites within Orc [Kitchin et al., 2009], which provides encapsulation similar to classes in object-oriented programming, we make use of the Ref site in Orc, that creates a rewritable storage location. The following Orc code presents these aspects:

--- Knowledge Query ---

Once the appropriate knowledge base elements are recognized, Gremlin queries are used to obtain dependencies from the Graph database. We assume that the knowledge base is pre-populated with property graphs as described in Section 4. Terms used in Fig. 3, such as hasProperty and requires are used in conjunction with Gremlin graph database filtering and complex queries, to populate local robotic knowledge bases (Lines 9-20 in Knowledge Query). While we represent this as explicit queries, alternate implementations may use dialogue engines to extract necessary information from the knowledge base via question-answers [A. Bordes and Weston, 2017][M. Maro and Origlia, 2017]. We make use of the def class declaration that allows us to implement sites within Orc [Kitchin et al., 2009], which provides encapsulation similar to classes in object-oriented programming, we make use of the Ref site in Orc, that creates a rewritable storage location. The following Orc code presents these aspects:

--- Knowledge Query ---

Once the appropriate knowledge base elements are recognized, Gremlin queries are used to obtain dependencies from the Graph database. We assume that the knowledge base is pre-populated with property graphs as described in Section 4. Terms used in Fig. 3, such as hasProperty and requires are used in conjunction with Gremlin graph database filtering and complex queries, to populate local robotic knowledge bases (Lines 9-20 in Knowledge Query). While we represent this as explicit queries, alternate implementations may use dialogue engines to extract necessary information from the knowledge base via question-answers [A. Bordes and Weston, 2017][M. Maro and Origlia, 2017]. We make use of the def class declaration that allows us to implement sites within Orc [Kitchin et al., 2009], which provides encapsulation similar to classes in object-oriented programming, we make use of the Ref site in Orc, that creates a rewritable storage location. The following Orc code presents these aspects:

--- Knowledge Query ---

Once the appropriate knowledge base elements are recognized, Gremlin queries are used to obtain dependencies from the Graph database. We assume that the knowledge base is pre-populated with property graphs as described in Section 4. Terms used in Fig. 3, such as hasProperty and requires are used in conjunction with Gremlin graph database filtering and complex queries, to populate local robotic knowledge bases (Lines 9-20 in Knowledge Query). While we represent this as explicit queries, alternate implementations may use dialogue engines to extract necessary information from the knowledge base via question-answers [A. Bordes and Weston, 2017][M. Maro and Origlia, 2017]. We make use of the def class declaration that allows us to implement sites within Orc [Kitchin et al., 2009], which provides encapsulation similar to classes in object-oriented programming, we make use of the Ref site in Orc, that creates a rewritable storage location. The following Orc code presents these aspects:

--- Knowledge Query ---

Once the appropriate knowledge base elements are recognized, Gremlin queries are used to obtain dependencies from the Graph database. We assume that the knowledge base is pre-populated with property graphs as described in Section 4. Terms used in Fig. 3, such as hasProperty and requires are used in conjunction with Gremlin graph database filtering and complex queries, to populate local robotic knowledge bases (Lines 9-20 in Knowledge Query). While we represent this as explicit queries, alternate implementations may use dialogue engines to extract necessary information from the knowledge base via question-answers [A. Bordes and Weston, 2017][M. Maro and Origlia, 2017]. We make use of the def class declaration that allows us to implement sites within Orc [Kitchin et al., 2009], which provides encapsulation similar to classes in object-oriented programming, we make use of the Ref site in Orc, that creates a rewritable storage location. The following Orc code presents these aspects:

--- Knowledge Query ---

Once the appropriate knowledge base elements are recognized, Gremlin queries are used to obtain dependencies from the Graph database. We assume that the knowledge base is pre-populated with property graphs as described in Section 4. Terms used in Fig. 3, such as hasProperty and requires are used in conjunction with Gremlin graph database filtering and complex queries, to populate local robotic knowledge bases (Lines 9-20 in Knowledge Query). While we represent this as explicit queries, alternate implementations may use dialogue engines to extract necessary information from the knowledge base via question-answers [A. Bordes and Weston, 2017][M. Maro and Origlia, 2017]. We make use of the def class declaration that allows us to implement sites within Orc [Kitchin et al., 2009], which provides encapsulation similar to classes in object-oriented programming, we make use of the Ref site in Orc, that creates a rewritable storage location. The following Orc code presents these aspects:

--- Knowledge Query ---

Once the appropriate knowledge base elements are recognized, Gremlin queries are used to obtain dependencies from the Graph database. We assume that the knowledge base is pre-populated with property graphs as described in Section 4. Terms used in Fig. 3, such as hasProperty and requires are used in conjunction with Gremlin graph database filtering and complex queries, to populate local robotic knowledge bases (Lines 9-20 in Knowledge Query). While we represent this as explicit queries, alternate implementations may use dialogue engines to extract necessary information from the knowledge base via question-answers [A. Bordes and Weston, 2017][M. Maro and Origlia, 2017]. We make use of the def class declaration that allows us to implement sites within Orc [Kitchin et al., 2009], which provides encapsulation similar to classes in object-oriented programming, we make use of the Ref site in Orc, that creates a rewritable storage location. The following Orc code presents these aspects:

--- Knowledge Query ---

Once the appropriate knowledge base elements are recognized, Gremlin queries are used to obtain dependencies from the Graph database. We assume that the knowledge base is pre-populated with property graphs as described in Section 4. Terms used in Fig. 3, such as hasProperty and requires are used in conjunction with Gremlin graph database filtering and complex queries, to populate local robotic knowledge bases (Lines 9-20 in Knowledge Query). While we represent this as explicit queries, alternate implementations may use dialogue engines to extract necessary information from the knowledge base via question-answers [A. Bordes and Weston, 2017][M. Maro and Origlia, 2017]. We make use of the def class declaration that allows us to implement sites within Orc [Kitchin et al., 2009], which provides encapsulation similar to classes in object-oriented programming, we make use of the Ref site in Orc, that creates a rewritable storage location. The following Orc code presents these aspects:

--- Knowledge Query ---

Once the appropriate knowledge base elements are recognized, Gremlin queries are used to obtain dependencies from the Graph database. We assume that the knowledge base is pre-populated with property graphs as described in Section 4. Terms used in Fig. 3, such as hasProperty and requires are used in conjunction with Gremlin graph database filtering and complex queries, to populate local robotic knowledge bases (Lines 9-20 in Knowledge Query). While we represent this as explicit queries, alternate implementations may use dialogue engines to extract necessary information from the knowledge base via question-answers [A. Bordes and Weston, 2017][M. Maro and Origlia, 2017]. We make use of the def class declaration that allows us to implement sites within Orc [Kitchin et al., 2009], which provides encapsulation similar to classes in object-oriented programming, we make use of the Ref site in Orc, that creates a rewritable storage location. The following Orc code presents these aspects:

--- Knowledge Query ---

Once the appropriate knowledge base elements are recognized, Gremlin queries are used to obtain dependencies from the Graph database. We assume that the knowledge base is pre-populated with property graphs as described in Section 4. Terms used in Fig. 3, such as hasProperty and requires are used in conjunction with Gremlin graph database filtering and complex queries, to populate local robotic knowledge bases (Lines 9-20 in Knowledge Query). While we represent this as explicit queries, alternate implementations may use dialogue engines to extract necessary information from the knowledge base via question-answers [A. Bordes and Weston, 2017][M. Maro and Origlia, 2017]. We make use of the def class declaration that allows us to implement sites within Orc [Kitchin et al., 2009], which provides encapsulation similar to classes in object-oriented programming, we make use of the Ref site in Orc, that creates a rewritable storage location. The following Orc code presents these aspects:

--- Knowledge Query ---
Action planning with procedures outlined in Section 5 can now be performed, with the high level goals being enacted through decomposition. Queries are made to the knowledge base to determine if the query terms are located in the world or target models, the absence of which triggers perception (Lines 9–11 in Action Planner). Similarly, queries for the robot and action models are triggered, which can trigger runtime exceptions such as lack of robot capabilities (Lines 14–17 in Action Planner). We also introduce a function to trigger re-planning replan_action, that looks for exceptions and may add capabilities such as a new robot model or action template (Lines 20–21 in Action Planner). The following Orc code presents these aspects:

```orc
+++ Action Planner +++
--Knowledge Base, Perception and Exception Pointers
include "KB.inc"
val perception = Dictionary()
val exception = Dictionary()
val world = Dictionary()

--Queries for world and object templates, with perception
def query1(v,db) = Iff member(v,db) >> (v,db) | Iff member(v,db) >> perception.p := v >> perception.p

--Queries for robot capabilities and actions, with exceptions
def query2(v,world_model) = If v, world_model

--Target query
def target(o) = query1(o, object_template)

--Replanning for runtime exceptions
def add_capabilities(v,db) = merge(db,v)
def replan_action(a) = if exception.ex? = null) >> add_capabilities(v,db)
def replan_action(a) = if exception.ex? = null) >> exception.ex := null >> action(a)

--- Action Planner End ---
```

The output of a typical action plan is now presented, which is input goals that are similar to those planned at design time. Once the query results from various knowledge models are obtained, the action can be performed that includes navigation, manipulation and task completion (Lines 14–15 in Design Time Simulation). Such an execution is straightforward as neither external perception or exceptions are triggered. The following Orc code presents these aspects:

```orc
+++ Design Time Simulation +++
--Input goals
robot("picker") | action("pick") | object("ball") | world("rack")

--Output-----------------------------
Target Query Triggered for ball
```

6.2 Runtime Adaptation Simulation

An important aspect of autonomous robotic deployment is runtime adaptation to changes. The goals are modified with the robot type replaced by mover, action collect and the target object replaced by cylinder. As these requirements are not pre-populated into the graph knowledge base, adaptation and exception handling procedures are triggered in Algorithm 2. We notice that perception is triggered to identify the target cylinder (Lines 12–13 in Runtime Adaptation Simulation). Exceptions are also triggered for the lack of collect actions and mover robot capabilities, that are further added into the knowledge base (Lines 15–21 in Runtime Adaptation Simulation). Post this adaptation, the action execution is completed. The following Orc code presents these aspects:

```orc
+++ Runtime Adaptation Simulation +++
--Input goals
robot("mover") | action("collect") | target("cylinder") | world("rack")

--Output----------------------------------
Action Query Triggered for collect
Target Robot Capability Query Triggered for mover
World Model Query Triggered for mover
Robot Capability Query Triggered for mover
Target Robot Capability Query Triggered for mover
Target Robot Capability Query Triggered for mover
Action Replan Triggered
Exception Triggered for mover
Adding knowledge of mover
Updated KB ["mover", "picker"]
Action Query Triggered for collect
Exception Triggered for collect
Adding knowledge of collect
Updated KB ["collect", "pick", "drop", "assign"]

Action Completed collect
```

6.3 ROS Smach Code Generation

To deploy the action plans to physical/virtual robots, we make use of the open source ROS Smach framework. This is a finite state machine where states and transition of the robot may be described with respect to complex tasks. We auto-generate this from the Orc task list, by referencing robot capabilities, world model and task templates seen in Fig. 3. An example of the

*http://wiki.ros.org/smach*
Figure 5: KUKA Picker Robot API Call Integration via ROS Smach.

Figure 6: Latency Measurements dependent on Knowledge Base Queries.

ROS Smach code generated is presented below, that produces the output of each state transition as succeeded, aborted or preempted. The PERCEPTION task, if successful is followed by ROBOT ARM MOVEMENT; else an abort or preemption is triggered:

ActivityDiagram PickPlace produces outcomes
  (succeeded, aborted, preempted)
  has activities{
    Activity PERCEPTION {
      inputData: {WORLD}
      requireCapability: {robot.camera}
      conditions {
        if (outcome is succeeded) nextActivity : ROBOT_ARM_MOVEMENT,
        if (outcome is preempted) final outcome : PickPlace.preempted,
        if (outcome is aborted) final outcome : PickPlace.aborted
      }
    }
    Activity ROBOT_ARM_MOVEMENT {
      inputData : {WORLD ROBOT TASK TARGET}
      requireCapability: {robot.movement}
      conditions {
        if (outcome is succeeded) nextActivity : ROBOT_GRIPPER_GRASP,
        if (outcome is preempted) final outcome : PickPlace.preempted,
        if (outcome is aborted) final outcome : PickPlace.aborted
      }
    }
    ...
  }

Integrating RoboPlanner with physical robotic simulator for action planning, such as that shown in Fig. 5, is then done using ROS API calls mapped to each ROS Smach task. ROS also provides ROS bridge interfaces to call physical robot sensor-actuator APIs via the task planning framework. This presents an end-to-end system for autonomous robot action planning (refer to Fig. 2), with knowledge integration, design time action planning, runtime execution and adaptation.

6.4 Performance Analysis

Given that we propose the use of knowledge bases and Gremlin graph queries to retrieve the language, performance impact of the queries must be analyzed. This is specially important in the case of Industry 4.0 deployments, where automation is intended to improve throughput. To estimate query and update times in OrientDB graph databases, we run the following stress test on a Linux workstation with 4 core i5-6200U CPU @ 2.30GHz, 4 GB RAM, which simulates the hyper-connected graph traversal over 50 nodes:

Starting workload GSP (concurrencyLevel=4)...
  - Workload in progress 100% [Shortest paths blocks
    (block size=50) executed: 50/50]
  - Total execution time: 2.762 secs
  - Executed 50 shortest paths in 2.762 secs
  - Path depth: maximum 8, average 5.286, not connected 0
  - Throughput: 18.103/sec (Avg 55.240ms/op)

The average graph traversal latency is seen to be around 211 milliseconds, that outperforms conventional perception and object recognition algorithms (2300 milliseconds in [Zhang and et al., 2016]). Using these mean values for exponentially distributed latency outputs, Monte-Carlo runs are performed over 20,000 runs.

Fig. 6 demonstrates outputs for various cases with the Knowledge Base having 100%, 90% and 70% of the action planning information (triggering perception and exception handling in case of missing knowledge). For instance, over the base case of 70% plan information in the Knowledge Base, the 95% percentile latency improves by 56.5% (90% queries answered by knowledge base) and by 73.9% (90% queries answered by knowledge base). This demonstrates that continuous learning and runtime updates have a significant impact on autonomous robotic performance. Thus, it is crucial to maintain an updated knowledge base within the RoboPlanner framework.

6.5 Graph Database Integrity

While the multi-modal OrientDB satisfies ACID (Atomicity, Consistency, Isolation, Durability) properties for databases, integrity checks are to be maintained when updating the databases. Integrity constraints are rules which define the set of consistent database states or changes of state. Typically, three types of checks are performed [Rabuzin et al., 2016]:

1. Schema instance: Entity types and type checking integrity.
2. Referential integrity: This checks that the nodes and edges are uniquely named and that the edges are provided with labels and start/end vertices.
3. Functional dependencies: Value restrictions on particular attributes. Defining minimum and maximum property value.

These checks are incorporated into the below Orc code for knowledge base updates. We notice that type checking (Line 4 in Database Update Integrity), redundancy
Such integrity checks and superior performance aspects can prove useful in other applications such as intelligent chatbots and dialogue engines, where updated knowledge bases and real-time responses are crucial.

In summary, our work demonstrates the following:

1. **RoboPlanner** Knowledge Base module that formally models robotic world models, capabilities, object descriptions and task templates – Fig. 3 and inputs to Knowledge Resolution/Knowledge Query examples in Section 6.

2. **RoboPlanner** Action Planner that uses design-time queries/updates to knowledge graph databases, including exception handling – Algorithm 2, Fig. 4 and Action Planner/Design Time Simulation examples in Section 6.

3. **RoboPlanner** Runtime simulation, adaptation and performance analysis of action plans using graph queries – Runtime Adaptation Simulation example in Section 6 and Fig. 6. Executable task templates as ROS Smach codes as presented in Section 6.3.

4. **RoboPlanner** integrity checks for runtime updates to the knowledge base – Database Update Integrity example in Section 6.

Such modules will prove useful across a host of Industry 4.0 deployments invoking autonomous robots.

7 Related Work

Industry 4.0 deployments [Lasi et al., 2014] propose the use of autonomous robotic entities to complete complex tasks. Commercial deployments have been used in warehouses [Barthokli and Hackman, 2016] [Zhang and et al., 2016] to improve throughput of automated tasks. Amazon\(^7\) has deployed hundreds of autonomous robots to aid in reducing costs of warehouse logistics [Wurman et al., 2008]. Inspiration is drawn from the use of autonomic computing technologies [Huebscher and McCann, 2008],

\(^7\)https://www.amazonrobotics.com/
that allow robotic runtime reconfiguration and adaptation. Architectures with self-aware, self-configuring and self-optimizing capabilities have also been proposed [Faniyi et al., 2014], that may be applied to such automation frameworks.

This has led to recent research on cognitive robotic systems [Levesque and Lakemeyer, 2010], with architectures such as RoboEarth [Tenorth and Beetz, 2013] and CRAM [Beetz et al., 2010] being proposed. While a few of these make use of semantic ontologies to represent knowledge, others make use of biological memory models to cache information. A review of cognitive architectures applied in multiple domains such as vision, learning, memory models and robotics have been provided in [Kotseruba and Tsotsos, 2018]. Table 2 provides a detailed comparison between RoboPlanner and other cognitive/autonomous robotic architectures. We notice that OWL based ontologies [Grimm et al., 2007] and queries using SPARQL/Prolog are heavily used, which suffer from performance deterioration when the knowledge base is large. Automated planners such as ROSPlan [Cashmore and et al., 2015] make use of logical PDDL transitions at task design time, rather than runtime executions. In particular, runtime exception handling and consistent model updates have not been fully considered in these frameworks.

In RoboPlanner, we propose the use of graph databases [Angles and Gutierrez, 2008] for knowledge representation, which maintain graph relationships within the database. Efficient graph queries are useful in dialogue and chatbot engines as presented in [M. Maro and Origlia, 2017]. We also propose using the Orc concurrent programming language, that may be use in conjunction with industrial workflow specifications (redacted for double blind review). Aspects of the Orc framework are similar to Hierarchical Task Networks [Erol et al., 1994], with complex expressions being sub-divided into atomic tasks. Orc further provides granularity in controlling concurrency, temporal actions and runtime behavior, that is more suited for action planning in robotics. A related programming approach is the GOAL agent programming language [Hindriks and Dix, 2014], that makes use of belief-desire-intention approaches to programming intelligent agents. Aspects of knowledge modeling, action templates and goal functions may be mapped to similar axioms provided in our framework. Such an approach may be extended to multiple autonomous robotic deployments.

8 Conclusions

Autonomous robots are being increasingly used in Industry 4.0 deployments to solve problems via intelligent adaptive mechanisms. A central tenet in such deployments is eliciting efficient action plans that may be executed at runtime. In this paper, we generate action plans through graph knowledge base queries via the RoboPlanner framework. Knowledge about robotic world models and capabilities are encoded in efficient graph database models, that may be efficiently queried to extract information for task completion. Using the concurrent programming language Orc, action plans are generated that can handle robotic runtime exceptions and perception information. End-to-end design/runtime simulations and performance analysis demonstrate the advantages of maintaining the robotic knowledge base.

References


