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

Ajay Kattepur, Balamuralidhar P

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 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 Orc [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* Runtime 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

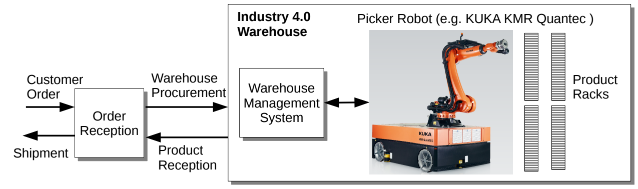


Figure 1: Automation for Warehouse Pick & Place Tasks.

is received, the products are procured from the warehouse. As shown in Fig. 1, autonomous Picker robots (such as KUKA KMR Quantec[1]) 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.

[1] https://www.kuka.com

Algorithm 1: Stateful Intelligent Robotic Agent.

Input: Robot Perception; Knowledge Base; Robot State;
Output: Robot Action;
Robot State ← *Interpret*(Perception);
Knowledge Base ← *Update*(Knowledge Base, Perception);
Action ← *Choose-Best-Action*(Knowledge Base);
Robot State ← *Update*(State, Action);
Knowledge Base ← *Update*(Knowledge Base, Action);

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 modules 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 use 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: `isOfType`, `hasProperty`, `requires` and `produces`. `isOfType` 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/

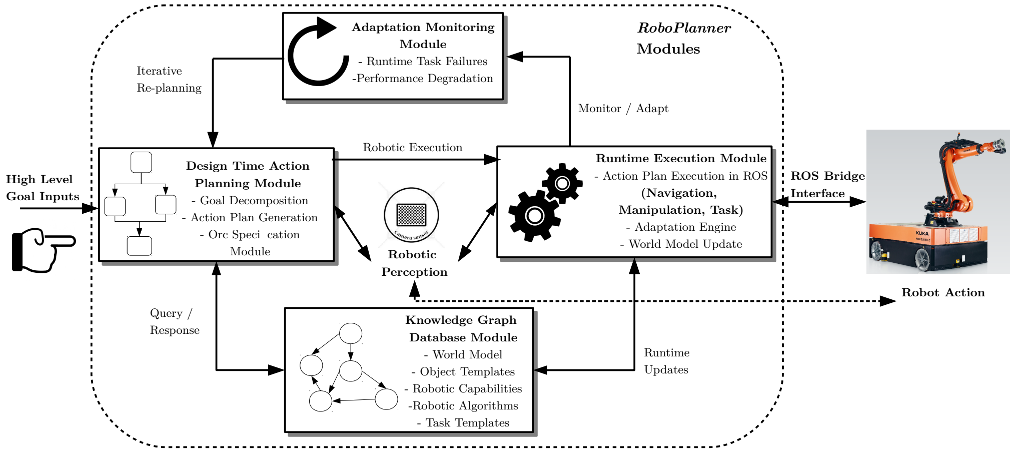
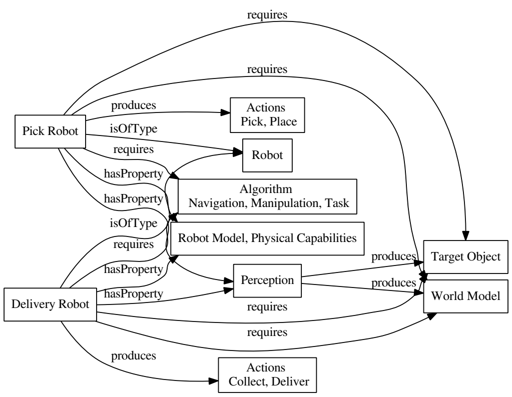
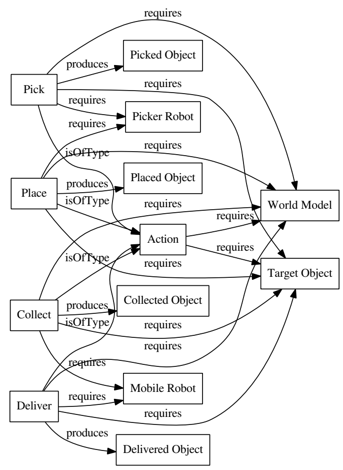


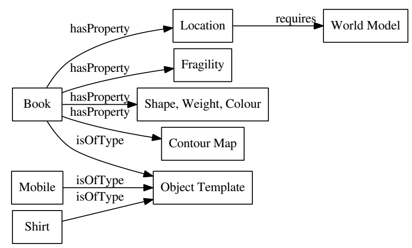
Figure 2: *RoboPlanner* Design/Runtime Execution Modules.



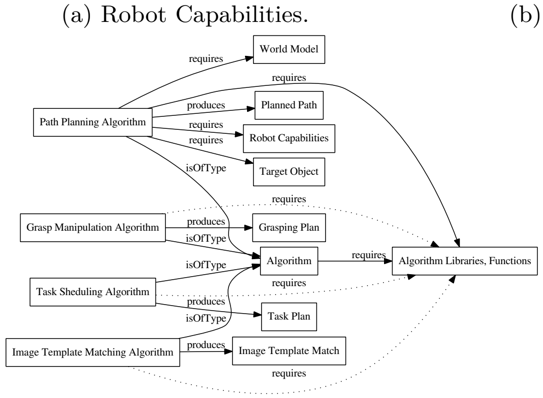
(a) Robot Capabilities.



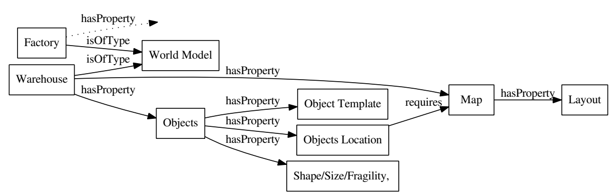
(b) Task Templates.



(c) Object Templates.



(d) Robotic Algorithms.



(e) World Model.

Figure 3: Knowledge Property Graphs for Autonomous Robots.

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. 3e 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 datastores 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[3]. OrientDB 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 data model [Angles and Gutierrez, 2008]. Properties are represented as *Key–Value* pairs that may be queried.

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=WorldModel}
     ==>{Name=Objects, Properties=ObjectProperties,
        Location=ObjectLocation}
     ==>{Name=Warehouse}
     ==>{Name=Map, Rack=RackConfig, Layout=WarehouseLayout,
        Aisles=AislesConfig}
gremlin> g.E
     ==>e[#26:0][#10:0-isOfType->#9:0]
     ==>e[#29:0][#10:0-hasProperty->#11:0]
     ==>e[#30:0][#10:0-hasProperty->#9:1]
     ==>e[#33:0][#9:1-requires->#11:0]
```

In order to query this graph, we use *Gremlin*[4], 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().Name` be used to filter out properties such as `Name` of a vertex.

```
gremlin> g.v('10:0').Name
     ==>Warehouse
gremlin> g.v('10:0').bothE
     ==>e[#26:0][#10:0-isOfType->#9:0]
     ==>e[#29:0][#10:0-hasProperty->#11:0]
     ==>e[#30:0][#10:0-hasProperty->#9:1]
```

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.

```
gremlin> g.V.has('Name', 'Warehouse').
      out('hasProperty').map
     ==>{Rack=RackConfig, Layout=WarehouseLayout,
        Aisles=AislesConfig, Name=Map}
     ==>{Properties=ObjectProperties, Name=Objects,
        Location=ObjectLocation}
```

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.

```
gremlin>  g.v('10:0').outE.inV.name.path
     ==>[v[#10:0], e[#26:0][#10:0-isOfType->#9:0],
        v[#9:0], null]
     ==>[v[#10:0], e[#29:0][#10:0-hasProperty->#11:0],
        v[#11:0], null]
```

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?" or "Can the robot lift this?" 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` s_1 and s_2, the expression $s_1 \mid s_2$ invokes both `sites` in parallel.
- *Sequential Combinator* ($>x>$, $\gg$): In the expression $s_1 >x> s_2$ (shorthand $s_1 \gg s_2$), `site` s_1 is evaluated

[3] https://orientdb.com/

[4] http://tinkerpop.apache.org/

initially, with every value published by s_1 initiating a separate execution of `site` s_2.

- *Pruning Combinator* ($<x<$, $\ll$): In the expression $s_1 <x< s_2$ (shorthand $s_1 \ll s_2$), 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 website[5]).

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 information to identify targets (`target?`) or atomic actions (`action?`). Perception triggering and re-planning in case of exceptions are also provided. Such a process of decomposing high level expressions to actionable tasks has also been employed in the automated planning community with Hierarchical Task Networks [Erol et al., 1994].

[5] https://orc.csres.utexas.edu/

Algorithm 2: Generating Action Plans for Goals via Knowledge Bases.

```
1  Input: Input Goal; Knowledge Base[World Model, Object
          Templates, Task Descriptions, Robot Capability,
          Algorithms];
2  Output: Action Plan;
3  Goal ← Verify(Input Goal, Knowledge Base[Task Descriptions]);
4  Subgoals ← Decompose(Goal, Knowledge Base[Task
     Descriptions]);
5  for each Subgoal do
6      (Action?, Target?) ← Requirements(Subgoal);
7      (Target?, Object Attributes?, Capability?) ←
         Requirements(Action);
8      Object Attributes? ← Requirements(Target);
9      if Object Attributes? is a member of Knowledge
         Base[World Model] then
           Object Attributes ← Query(Object Attributes?,
             Knowledge Base[World Model]);
       else
           if Object Attributes? can be obtained by Perception
             then
               Object Attributes ← Perception(Object
                 Attributes?, Knowledge Base[World Model, Robot
                 Capability, Perception Algorithms]);
           else
               Exception ← Object Attributes?
10     if Target? is a member of Knowledge Base[Object
         Templates] then
           Target ← Query(Target?, Knowledge Base[Object
             Templates]);
       else
           if Target? can be obtained by Perception then
               Target ← Perception(Target?, Knowledge
                 Base[World Model, Robot Capability, Perception
                 Algorithms]);
           else
               Exception ← Target?
11     Capability ← Query(Capability?, Knowledge Base[Robot
         Capability]);
12     if Capability satisfies Action then
           Action ← Query(Action?, Target, Object Attributes,
             Capability, Knowledge Base[Navigation/Manipulation
             Algorithms]);
       else
           Exception ← Action?
13     if Exception is null then
           Action Plan ← Update(Action);
       else
           Trigger Re-planning of Subgoal
14 return Action Plan satisfying Input Goal;
```

Note that though we have represented this via graph queries, another view would be request-responses with a dialogue agent [A. Bordes and Weston, 2017], representing the knowledge base. The dialogue agent could be used to further clarify queries that may be ambiguous. A typical conversation instance could be:

```
user: pick ball
RoboPlanner Dialogue Agent: recognize two target := ball;
                            colour? red colour? blue. Which color?
User: pick red ball
RoboPlanner Dialogue Agent: target := ball; colour := red;
                            action := pick; proceed?
```

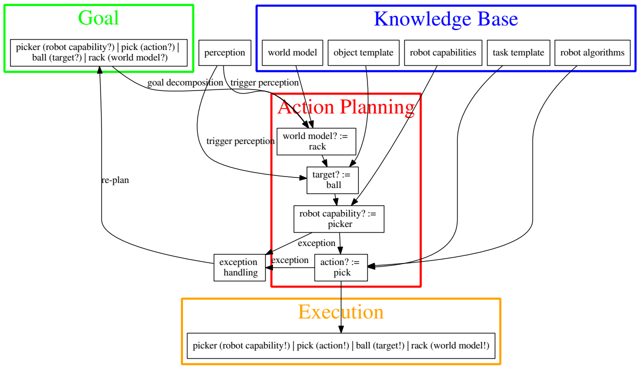


Figure 4: Action Planning for a `pick` task.

Table 1: A (Non-exhaustive) List of Artifacts in the Action Planning Framework.

Artifacts	Instances
world model	`warehouse` | `factory` | `home environment` | `shipyard`
object template	`ball` | `box` | `obstacle` | `component`
robot capabilities	`picking` | `movement` | `obstacle avoidance` | `detection`
perception	`depth camera` | `odometry sensor` | `gyroscope`
task templates	`delivery` | `scheduling` | `monitoring`
robot algorithms	`localization & mapping` | `edge detection` | `path planning`
target	`ball` | `bin` | `book` | `package` | `conveyor belt`
action	`pick` | `grasp` | `move` | `follow` | `drop` | `hold`
goal	`action?` | `target?` | `world model?` | `robot capability?`

```
User: yes
```

Such a system unifies the modeling of both knowledge acquisition from a central repository, robotic updates and queries that may be made to human participants. The action plans that are formulated result in valid goal fulfillment, due to varied knowledge sources incorporated.

To further generalize this action planning framework, we provide instances of multiple artifacts in Table 1. We emphasize that the procedure outlined in Algorithm 2 and Fig. 4 is structured to be generic, allowing action plans to be generated with various world models, robotic capabilities and task templates. Such a structured way of planning actions will prove valuable across multiple deployments.

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`).

```
1  +++ Knowledge Resolution +++
2
3  --Knowledge Base Pointers
4  include "KB.inc"
5  val b=[World_model, Object_template, Robot_cap,
       Task_template, Robot_algo]
6
7  --Mapping Process
8  def search(a,b) = Ift member(a,head(b)) >> Println(
       a+" in "+head(head(b))) | Iff member(a,head(b)
       ) >> search(a,tail(b))
9  def plan(a) = search(a,b)
10
11 --Input Goals
12 map(plan,["pick","ball","rack","picker"])
13
14 --Orc Output---------------------------------------
15 rack in World_model
16 ball in Object_template
17 picker in Robot_cap
18 pick in Task_template
```

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:

```
1  +++ Knowledge Query +++
2
3  --Reference store for retrieved data
4  val World_model = Ref([])
5  def append_model(v) = World_model? >m>
       append([v],m) >q> World_model:= q
6
7
8  --Gremlin Query site
9  def class gremlin()=
10    def find(v,D) = g.V.has(v,D).map >v>
          append_model(v)
11
12    def hasProperty(v,D) =  g.V.has(v,D).outE
13          ('hasProperty').inV.map >v> append_model(v)
14    def requires(v,D) = g.V.has(v,D).outE
15          ('requires').inV.map >v> append_model(v)
16    def isOfType(v,D) = g.V.has(v,D).outE
17          ('isofType').inV.map >v> append_model(v)
18    def produces(v,D) = g.V.has(v,D).outE
19          ('produces').inV.map >v> append_model(v)
20    stop
21
22 --Searching Dependencies for "rack"
23 val gremlin = gremlin()
24 gremlin.find("Name","rack") | gremlin.hasProperty("
       Name","Objects") | gremlin.requires("Name","
```

```
    Map") | gremlin.isOfType("Name","WorldModel")
    >> World_model?

--Output------------------------------------------
["WorldModel", ("Map", "WarehouseLayout"),
("Objects", "ObjectLocation", "ObjectProperties"),
    "rack"]
```

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:

```
1  +++ Action Planner +++
2
3  --Knowledge Base, Perception and Exception Pointers
4  include "KB.inc"
5  val perception = Dictionary()
6  val exception = Dictionary()
7
8  --Queries for world and object templates, with
       perception
9  def query1(v,db) = Ift member(v,db) >> (v,db) |
       Iff member(v,db) >> perception.p := v >>
       perception.p?
10 def world(w) = query1(w, world_model)
11 def target(o) = query1(o, object_template)
12
13 ---Queries for robot capabilities and actions, with
        exceptions
14 def query2(v,db) = Ift member(v,db) >> (v,db) |
15     Iff member(v,db) >> exception.ex := v >>
           add_capabilities(v,db)
16 def robot(r) =  query2(r, robot_cap)
17 def action(a) =  query2(a, task_template) >> (
       query2(("navigation","task","manipulation")
       ,robot_algo)) | replan_action(a))
18
19 --Replanning procedures for runtime exceptions
20 def add_capabilities(v,db) = merge(db,[v])
21 def replan_action(a) = Ift (exception.ex? = null)
       >> stop |  Iff (exception.ex? = null) >>
       exception.ex := null >> action(a)
```

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:

```
1  +++ Design Time Simulation +++
2
3  --Input goals
4  robot("picker") | action("pick") | object("ball") |
       world("rack")
5
6  --Output------------------------------------------
7  Target Query Triggered for ball
8  World Model Query Triggered for rack
9  Robot Capability Query Triggered for picker
10 Action Query Triggered for pick
11 ("ball", ["ball", "cube"])
12 ("picker", ["picker"])
13 ("rack", ["warehouse", "rack"])
14 (("navigation", ["navigation", "manipulation", "
       task"]), ("task", ["navigation", "manipulation
       ", "task"]), ("manipulation", ["navigation", "
       manipulation", "task"]))
15 Action Completed pick
```

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:

```
1  +++ Runtime Adaptation Simulation +++
2
3  --Input goals
4  robot("mover") | action("collect") | target("
       cylinder") | world("rack")
5
6  --Output------------------------------------------
7  Action Query Triggered for collect
8  Target Query Triggered for cylinder
9  Robot Capability Query Triggered for mover
10 World Model Query Triggered for rack
11 ("rack", ["warehouse", "rack"])
12 Perception Trigged for cylinder
13 "cylinder"
14 Action Replan Triggered
15 Exception Trigged for mover
16 Adding knowledge of mover
17 Updated KB ["mover", "picker"]
18 Action Query Triggered for collect
19 Exception Trigged for collect
20 Adding knowledge of collect
21 Updated KB ["collect", "pick", "drop", "assign"]
22 (("navigation", ["navigation", "manipulation", "
       task"]), ("task", ["navigation", "manipulation
       ", "task"]), ("manipulation", ["navigation", "
       manipulation", "task"]))
23 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 [6] 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

[6] http://wiki.ros.org/smach

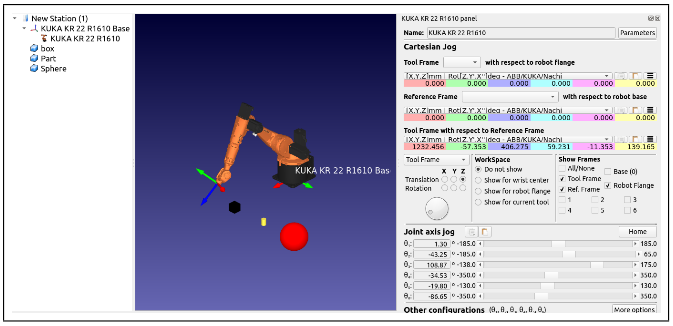


Figure 5: KUKA Picker Robot API Call Integration via ROS Smach.

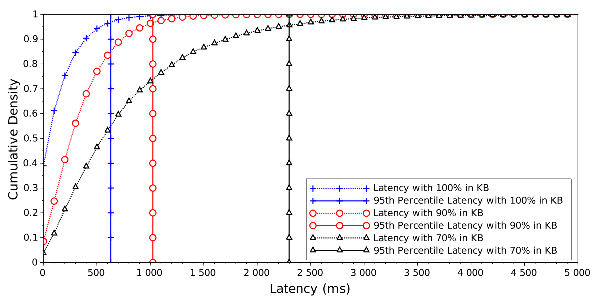


Figure 6: Latency Measurements dependent on Knowledge Base Queries.

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: 2.768 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)
- Latency Avg: 211.996ms/op (58th percentile) - Min: 55.838ms -
               99th Perc: 576.653ms - 99.9th Perc: 576.653ms -
               Max: 576.653ms - Conflicts: 0
```

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

Table 2: Autonomous / Cognitive Robotic Architectures.

Feature Modules	**CRAM** [Beetz et al., 2010] **RoboEarth** [Tenorth and Beetz, 2013]	**ACT-R/E**[Trafton et al., 2013]	**SOAR** [Laird et al., 2012]	**OpenRobots Ontology (ORO)** [Lemaignan et al., 2010]	**RoboPlanner**
Application Domains	Cognitive Service Robots, Knowledge Sharing among Robots	Human–Robot Coordination	Autonomous Mobile Robots	Cognitive Service Robots	**Autonomous Robots**
Knowledge Base	KnowRob [Tenorth and Beetz, 2013] OWL Ontologies	Declarative knowledge (fact-based memories); Procedural knowledge (rule-based memories)	Semantic Memory Models – Symbolic and Episodic	ORO OWL and RDF Triplestore	**Graph Databases with World Model, Robot Capabilities, Algorithms and Task Templates**
Knowledge Queries	Knowrob (Prolog) queries, that can be extended to other ontology queries	High level model interactions	STRIPS [Russell and Norvig, 2015] like decision procedures	SPARQL Queries	**Gremlin Knowledge Graph Queries**
Action Planning	CRAM Plan Language (CPL) allowing concurrent, parallel processes	Intentional (Goal) module	Procedural memory module	CRAM integration with logical rules	**Orc Specifications with Knowledge Base queries; ROS Smach Code**
Runtime Exception Handling	COGNITO reasoning system that processes failure traces	Utility based rewards; Visual and Aural modules	Reenforcement Learning	Human expert intervention	**Adaptation and exception handling modules**
Runtime Knowledge Base Updates	No Explicit Mention	Knowledge chunks updated	Chunks of memory data updated	RDF Triple updates with consistency checks	**Graph database update with integrity checks**
Performance Evaluation	No Explicit Mention	Accuracy of Actions with respect to World Model changes	Cognitive Reactivity measured	ORO Server performance evaluation (updation, queries)	**Graph database performance, exception handling delay**

of input data (Line 0 in `Database Update Integrity`) and valid range of properties (Line 11 in `Database Update Integrity`) are included. When a robot produces a runtime update, the site `update_node(key,value)` checks for integrity before pushing it to the knowledge base (Line 15 in `Database Update Integrity`).

```
1  +++ Database Update Integrity +++
2
3  --Type information
4  type world_model = {. Name :: String, Colour ::
       String, Location :: (Number,Number,Number) .}
5  val new_world_model =  Dictionary()
6
7  --Integrity check site
8  def class integrity()=
9      val range = range
10     def redundancy_check(key,value) = Ift(key =
           value) >> false | Iff(key = value) >>
           true
11     def value_check(key,range) = Ift(member(
           key,range)) >> true | Iff(member(
           key,range)) >> false
12     stop
13
14 def class update()=
15     def update_node(key,value) = Read(key) >aa> (
           integrity.redundancy_check(key,value)
           ,integrity.value_check(key,value)) >>
           new_world_model.aa := value
16     stop
```

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 [Bartholdi 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

[A. Bordes and Weston, 2017] A. Bordes, Y. B. and Weston, J. (2017). Learning end-to-end goal-oriented dialog. In *Intl. Conf. on Learning Representations*.

[Angles and Gutierrez, 2008] Angles, R. and Gutierrez, C. (2008). Survey of graph database models. *ACM Computing Surveys*, 40(1).

[Bartholdi and Hackman, 2016] Bartholdi, J. and Hackman, S. (2016). *Warehouse and Distribution Science*. The Supply Chain and Logistics Institute.

[Beetz et al., 2010] Beetz, J., Mosenlechner, L., and Tenorth, M. (2010). Cram: A cognitive robot abstract machine for everyday manipulation in human environments. In *Intl. Conf. on Intelligent Robots and Sys.*

[Cashmore and et al., 2015] Cashmore, M. and et al. (2015). Rosplan: Planning in the robot operating system. In *Proc. of the Intl. Conf. on on Automated Planning and Scheduling*.

[Erol et al., 1994] Erol, K., Hendler, J., and Nau, D. (1994). Htn planning: Complexity and expressivity. In *AAAI*.

[Faniyi et al., 2014] Faniyi, F., Lewis, P. R., Bahsoon, R., and Yao, X. (2014). Architecting self-aware software systems. In *IEEE/IFIP Conf. on Software Architecture*.

[Greengard, 2015] Greengard, S. (2015). *The Internet of Things*. MIT Press.

[Grimm et al., 2007] Grimm, S., Hitzler, P., and Abecker, A. (2007). Knowledge representation and ontologies. *Springer Semantic Web Services*, pages 51–105.

[Hindriks and Dix, 2014] Hindriks, K. and Dix, J. (2014). Goal: A multi-agent programming language applied to an exploration game. *Springer Agent-Oriented Software Engineering*.

[Huebscher and McCann, 2008] Huebscher, M. and McCann, J. (2008). A survey of autonomic computing – degrees, models, and applications. *ACM Computing Surveys*, 40(3).

[Kitchin et al., 2009] Kitchin, D., Quark, A., Cook, W., and Misra, J. (2009). The orc programming language. In *Proc. of FMOODS/FORTE*.

[Kotseruba and Tsotsos, 2018] Kotseruba, I. and Tsotsos, J. (2018). 40 years of cognitive architectures: core cognitive abilities and practical applications. *Springer Artificial Intelligence Review*.

[Laird et al., 2012] Laird, J., Kinkade, K., Mohan, S., and Xu, J. (2012). Cognitive robotics using the soar cognitive architecture. *AAAI Tech. Report*.

[Lasi et al., 2014] Lasi, H., Fettke, P., Kemper, H., Feld, T., and Hoffmann, M. (2014). Industry 4.0. *Business & Info. Sys. Engineering*, 6:239–242.

[Lemaignan et al., 2010] Lemaignan, S., Ros, R., Mosenlechner, L., Alami, R., and Beetz, M. (2010). Oro, a knowledge management platform for cognitive architectures in robotics. In *Intl. Conf. on Intelligent Robots and Systems*.

[Levesque and Lakemeyer, 2010] Levesque, H. and Lakemeyer, G. (2010). Cognitive robotics. *Dagstuhl Seminar Proc.*

[M. Maro and Origlia, 2017] M. Maro, M. Valentino, A. R. and Origlia, A. (2017). Graph databases for designing high-performance speech recognition grammars. In *Proc. of the 12th Intl. Conf. on Computational Semantics*.

[Rabuzin et al., 2016] Rabuzin, K., Sestak, M., and Konecki, M. (2016). Implementing unique integrity constraint in graph databases. In *Intl. Multi-Conf. on Computing in the Global Information Technology*.

[Russell and Norvig, 2015] Russell, S. and Norvig, P. (2015). *Artificial Intelligence: A Modern Approach*. Pearson.

[Tenorth and Beetz, 2013] Tenorth, M. and Beetz, M. (2013). Knowrob – a knowledge processing infrastructure for cognition-enabled robots. *Intl. J. of Robotics Research*, 32.

[Trafton et al., 2013] Trafton, G., Hiatt, L., Harrison, A., Tamborello, F., Khemlani, S., and Schultz, A. (2013). Act-r/e: An embodied cognitive architecture for human-robot interaction. *J. of Human Robot Interaction*, 2(1).

[Wurman et al., 2008] Wurman, P., D'Andrea, R., and Mountz, M. (2008). Coordinating hundreds of cooperative, autonomous vehicles in warehouses. *AAAI Artificial Intelligence Mag.*, 29:9–19.

[Zhang and et al., 2016] Zhang, H. and et al. (2016). Dorapicker: An autonomous picking system for general objects. In *IEEE Intl. Conf. on Automation Science and Engineering (CASE)*.