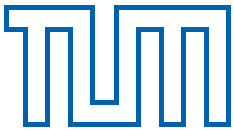
Institut für Informatik
Technische Universität München

The Cognitive Robot Abstract Machine

A Framework for Cognitive Robotics

Dissertation

Lorenz Mösenlechner



TECHNISCHE UNIVERSITÄT MÜNCHEN

Lehrstuhl für Informatik XXIII

Sensor Based Robotic Systems and Intelligent Assistance Systems

The Cognitive Robot Abstract Machine:

A Framework for Cognitive Robotics

Lorenz Mösenlechner

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:		.....
Prüfer der Dissertation:	1.	.....
	2.	.....
	3.	.....

Die Dissertation wurde am ..... bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am ..... angenommen.

Abstract

With the development of mobile robotic platforms that can operate in domestic domains such as a human household, for instance Willow Garage's PR2, solving tasks such as cleaning up or cooking food has become possible. The software that controls these robots became more and more complex and recent research provides solutions for navigation, manipulation and perception. With rising complexity of the tasks to be performed, the requirements for a high level executive that orchestrates arm motions, navigation and perception changed towards a tight integration of planning, reasoning and action execution. This thesis describes CRAM, a complete framework for implementing cognitive high-level robot control programs to enable robotic agents to execute their tasks more robustly, reliably and flexibly.

CRAM combines a powerful language for programming robot actions with sophisticated reasoning mechanisms. This allows the programmer to state plans more abstractly by letting the robot make decisions based on its current belief about the environment and the predicted future course of actions. More specifically, besides the plan language, CRAM provides a Prolog-like reasoning engine and many components to solve problems such as *where to stand* to perform actions, *is an object visible* to the robot from a certain location, *from where can an object be seen or grasped* but also components that allow the robot to reason about what it did, when and why. In addition to providing the means for implementing high-level control programs, the system also provides components for predicting the outcome of a plan. This, in combination with the other reasoning modules, allows the robot to base its decisions not only on knowledge about the current state of the world but to integrate the future course of actions.

Kurzfassung

Mit der Entwicklung mobiler Roboterplattformen wie Willow Garages PR2, die in häuslichen Umgebungen, wie zum Beispiel einem Haushalt, operieren können, wurde die Ausführung anspruchsvoller Aufgaben wie putzen oder kochen mit diesen Robotern möglich. Die Software, die diese Roboter steuert, ist immer komplexer geworden und aktuelle Forschungsergebnisse bieten Lösungen für Navigation, Manipulation und Perzeption. Mit zunehmender Komplexität der zu erfüllenden Aufgaben haben sich auch die Anforderungen an eine übergeordnete Kontrollschicht hin zu einer Integration von Planungsalgorithmen, logischem Schließen und der Ausführung von Aktionen verändert. In der hier vorliegenden Arbeit wird CRAM beschrieben, ein komplettes Framework um kognitive Kontrollprogramme für Roboter zu entwickeln, die es ihnen erlauben, ihre Aufgaben robuster, zuverlässiger und flexibler zu erfüllen.

CRAM kombiniert eine Programmiersprache zur Robotersteuerung mit Mechanismen zum automatischen Ziehen logischer Schlussfolgerungen. Damit kann der Programmierer Pläne abstrakter schreiben und dem Roboter bestimmte Entscheidungen überlassen, die dieser basierend auf seinem Umgebungsmodell und einer Vorhersage zukünftiger Aktionen treffen kann. Neben einer Programmiersprache zum Schreiben von Plänen enthält CRAM eine prologartige Sprache zur Lösung logischer Probleme sowie eine Vielzahl anderer Komponenten, die es ermöglichen, Probleme, wie zum Beispiel wo der Roboter stehen soll um bestimmte Aktionen auszuführen, ob ein Objekt an einer bestimmten Stelle sichtbar ist oder wo der Roboter stehen soll, um ein bestimmtes Objekt zu sehen oder zu greifen, zu lösen. Daneben entält CRAM Komponenten, die dem Roboter Schlüsse über seine Aktionen erlauben, insbesondere was er getan hat, wann und aus welchem Grund. Zusätzlich enthält CRAM Komponenten um die Effekte und Ergebnisse von Plänen vorherzusagen, was es dem Roboter in Kombination mit den anderen Inferenzalgorithmen ermöglicht, Entscheidungen nicht

nur basierend auf Wissen über den aktuellen Zustand der Umgebung, sondern auch unter Einbeziehung zukünftiger möglicher Weltzustände zu treffen.

Acknowledgements

The work presented in this thesis would not have been possible without the support of my advisor Michael Beetz. I want to thank him for his great vision, the many motivating and fruitful discussions and the support I got from him.

Furthermore, I want to thank all my colleagues who provided a great and unique working atmosphere, for the many interesting and enriching discussions I had with them. I especially want to thank Moritz Tenorth, Ulrich Klank, Dominik Jain, Nico Blodow, Thibault Kruse for their friendship and for the many things I could learned from them. They helped me to become a better scientist and a better software engineer. I also owe thanks to my former students Nikolaus Demmel, Tobias Rittweiler and Gayane Kazhoyan for helping me with the implementation of CRAM and for giving me valuable feedback on its design.

I also want to thank Elena, my wife, for enriching my live and for the support and the motivation she gave me during my academic career.

And last but not least I want to thank my family and my friends for their encouragement and support.

Contents

Abstract III

Kurzfassung V

Acknowledgements VII

Contents IX

List of Resources XIII

1 Introduction 1
- 1.1 Problem Description . . . 4
- 1.2 System Architecture . . . 10
- 1.3 Contributions . . . 16
- 1.4 Reader's Guide . . . 18

2 The CRAM Software Architecture 21
- 2.1 The CRAM Plan Language . . . 23
  - 2.1.1 Language Syntax . . . 24
  - 2.1.2 Definition of Terms . . . 32
  - 2.1.3 Fluents . . . 35
  - 2.1.4 CRAM Plan Language Expressions . . . 46
- 2.2 Reasoning and the CRAM Prolog Interpreter . . . 66
  - 2.2.1 Implementation of the Prolog Interpreter . . . 69
  - 2.2.2 Implementation of Predicates . . . 70
- 2.3 Symbolic Plan Parametrization . . . 74
  - 2.3.1 Designator Concepts . . . 74
  - 2.3.2 Designator Resolution . . . 81

2.4 Process Modules 85
2.4.1 The CRAM Process Module Interface 88
2.4.2 Synchronous and Asynchronous Action Execution 91
2.5 The CRAM Plan Library 95
2.5.1 Transparent Plans 95
2.5.2 Goals in the Current CRAM System 98
2.6 Related Work 102
2.7 Discussion 103

3 CRAM Reasoning Extensions 105
3.1 Physics-Based Reasoning 106
3.1.1 Physics Engine Integration and the World Database 110
3.1.2 Visibility Computation 126
3.1.3 Reachability Reasoning 131
3.1.4 Belief State Representation using CRAM's World Database . . . 139
3.2 Reasoning about Plan Execution 143
3.2.1 Representation of Execution Traces 145
3.2.2 Querying Execution Traces 148
3.2.3 Examples for Reasoning on Execution Traces 157
3.3 Related Work 160
3.4 Discussion 161

4 Parameterizing Plans 163
4.1 Density Maps for Sampling Solution Candidates 165
4.1.1 Implementation of Density Maps 169
4.2 Generation of Height and Orientation Values 189
4.3 The Density Map Generator API and Implementation Details 191
4.3.1 The Internal Representation of Density Maps and Generators . 192
4.3.2 Prolog API for Integration in Designator Resolution 194
4.3.3 Integration in Designator Resolution 197
4.4 Related Work 198
4.5 Discussion 198

5 Temporal Projection of Plans **201**
5.1 Projection of Plans and Generation of Timelines 203
5.1.1 The Projection Environment 203
5.1.2 Process Modules for Projection 205
5.2 Handling of Time in Projection 209
5.3 Definition and Matching of Behavior Flaws 211
5.4 Location Designator Optimization Using Projection 217
5.5 Related Work 217
5.6 Discussion 218

6 Conclusion **221**

Bibliography **231**

Index **239**

List of Resources

Figures

1.1 The PR2 (left) and TUM Rosie (right). 2
1.2 In our example, silverware, mugs and plates are stored in two different drawers at the counter. 5
1.3 Table set for breakfast for two persons. This is the final configuration for the example in this Section. 9
1.4 When manipulating a plate after the knife has been placed, the knife is blocking the put-down action because the robot's gripper collides with it. 10
1.5 Overview of the different CRAM components presented in this thesis. Components are grouped into two main areas, reasoning and the executive. The executive updates information stored in the reasoning system and uses the reasoning components for planning and decision making. 11

2.1 The architecture of our CRAM based executive. 22
2.2 The task tree if the simple navigation plan shown in Listing 2.2. 34
2.3 Simple example fluent network for a fluent that is true when the robot is closer than 20cm to its goal. 43
2.4 Allowed state transitions during the life time of a task object. 61
2.5 Code tree of the CRAM program shown in Listing 2.3. The same function is executed twice. Each execution branch has a unique path. . . . 64

2.6 Process module encapsulating a navigation control process. The input is an action designator specifying that the action is a navigation action and the containing the goal location as represented by a location designator, e.g. *(a location (to see) (obj cup))*. 86

3.1 Unstable scenes in the initial configuration (left) and after simulating for 0.5 seconds. In the top row, a mug is standing on the edge of a plate, in the bottom row, the plate has been placed on a mug. 124
3.2 Example scene with a table, the robot and tree objects: a bowl (white), a mug (blue) and pot (black) 127
3.3 Generated images to infer the visibility (and occluding objects) of the pot. Top left: rendered scene from robot perspective; top right: the pot centered; bottom left: only the pot; bottom right: the complete scene, every object has a unique color. The pot is (partly) occluding the bowl. 128
3.4 The default tool frame relative to the PR2's gripper frame. 133
3.5 Side and top grasp and front grasp used for reasoning about reachability.134
3.6 Computation of the blocking predicate for the cup. The top row shows the IK solutions for the three possible grasps for the right arm and the bottom row shows the corresponding solutions for the left arm. The robot is in collision with the pot when using the right arm, so the pot is considered a blocking object. 138
3.7 Scenes with objects being triangulated from raw point cloud data. The top row shows the camera image and the bottom row a visualization of the corresponding world database. As can be seen, only half of the pancake maker is visible for the robot and has been triangulated. The picture shows the initial state with the box standing left behind the bottle and a later detection where the box has been moved externally. The system retracted the original instance of the box and asserted a new instance at a different location. 144
3.8 Plan execution and recording of execution traces. The task tree and the internal state of the CRAM plan execution environment (including the belief state) are logged and stored in an execution trace. 145

4.1 Density Map for locations *on a counter-top*. The red areas indicate values in the density map that are greater than zero meaning that these points are potential solutions for locations on counter tops. 166
4.2 The different generator functions to generate a (sampled) pose in three dimensional space, including its orientation. 166
4.3 The three different density maps and the resulting density map that are generated by the designator description(location ((to see) (to reach) (obj Cup1))). As can be seen, the example scene consists of a cup and a pot that is standing close to the cup with the pot occluding the cup. . . 168
4.4 Occupancy grid used by navigation and the corresponding density map for representing locations the robot can navigate to. 170
4.5 Different density maps generated from Knowrob's semantic map. . . . 172
4.6 OpenGL depth maps generated by rendering the scene from the location of one of the cups in four directions (0 degrees, 90 degrees, 180 degrees and 270 degrees). 174
4.7 Density Map generated from the depth maps shown in Figure 4.6. . . . 176
4.8 Reachability maps for the PR2's two arms. 181
4.9 Different visualizations of inverse reachability maps for the PR2's left arm. The colored spheres indicate locations for placing the root of the manipulator's kinematic chain from which the origin can be reached. A red color indicates that more orientations exist from which the origin can be reached, blue indicates less orientations. 182
4.10 Generated reachability density map for reaching the cup on the counter top. 184
4.11 Different coordinate systems inferred based on the closed edge on the supporting object. 187
4.12 Density Maps for resolving the spatial relations *"right-of"* and *"behind"*. 188
4.13 Density Maps for resolving the *"near"* and the *"far"* relations. 189

5.1 System overview of plan projection for designator resolution. 202
5.2 Projection of two parallel actions in two different process modules and the generated projection timeline. 211

Tables

2.1 Functions for creating fluent networks. 44
2.2 Occasions that can be achieved in the current implementation of the CRAM plan library. 98

3.1 Predicates used to perform physics-based inferences 109
3.2 Currently supported object types and their parameters that can be asserted in the CRAM reasoing world database. 117
3.3 Event statements. 151

Algorithms

1 The code generated by the pursue macro. It executes all forms in parallel and terminates as soon as one of the sub-forms terminate. 58

2 The algorithm for generating a visibility density map based on previously rendered depth maps in all 4 directions around the object of interest. 178
3 The algorithm for computing a reachability map. 180
4 The algorithm for computing an inverse reachability map from a reachability map as generated by Algorithm 3. 183
5 The algorithm for generating a reachability density map as shown in Figure 4.10. The algorithm uses an inverse reachability map generated with Algorithm 4. 184
6 The algorithm for generating density maps for the directional spatial relations *“left-of”*, *“right-of”*, *“behind”* and *“in-front-of”*. 188

Chapter 1

Introduction

In the last couple of years, huge advancements in mobile service robotics have been made. In contrast to classical industrial robots, new robotic platforms such as Willow Garage's *PR2* [Wyrobek et al., 2008], Fraunhofer's *Care-O-bot* [Reiser et al., 2009] or TUM-Rosie [Beetz et al., 2010] have the potential to execute complex and sophisticated actions such as cleaning, cooking and doing laundry autonomously and in dexterous environments. Impressive demonstration videos, for instance the PR2 making popcorn and TUM-Rosie making sandwiches[1], the PR2 folding towels [Maitin-Shepard et al., 2010] or TUM Rosie making pancakes [Beetz et al., 2011], show the potential of these platforms and the state of current research. Most of the shown applications are a variation of pick-and-place tasks with the addition of actions such as stirring, pouring and the manipulation of articulated objects. While showing impressive work, these videos demonstrate specific sub-components but rarely show systems that integrate reasoning, decision making and high-level control. Instead, the orchestration of the different actions is ad hoc and decision making is mostly based on heuristics.

Recently, research in the field of mobile robotics started to develop sophisticated solutions for a lot of problems involved in mobile manipulation as can be seen in the previously mentioned demonstrations. This includes navigation with ready-to-use modules for 2D SLAM based on laser scans [Grisetti et al., 2007], navigation path planning [Meeussen et al., 2010] and collision avoidance based on laser scans. Huge advancements in the area of motion planning have been made. Systems such as

[1] http://youtu.be/BVAlt0FYmiI

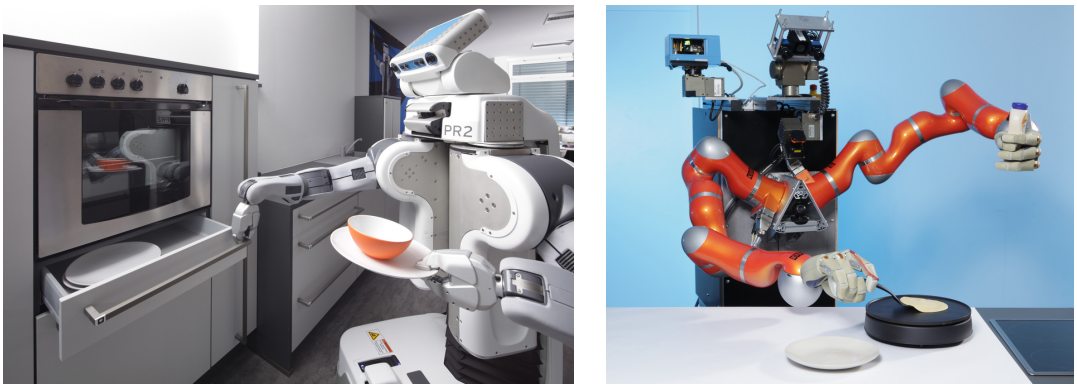


Figure 1.1: *The PR2 (left) and TUM Rosie (right).*

move_arm[2] and its successor *moveit*[3] provide flexible and extensible motion planning solutions that can be applied on many robots. Systems such as *OpenRAVE*[4] [Diankov and Kuffner, 2008] provide, in addition to motion planning, modules for grasp planning and for fast solutions for inverse kinematics. Perception is one of the most critical parts when executing actions in unknown, dynamic and changing environments since objects can only be manipulated if they can be detected reliably. Unfortunately, perception is also one of the hardest problems and, to date, satisfying solutions only exist for very restricted problem domains and sets of objects. Examples of such systems include *COP* [Klank, 2012] and a tabletop object detector used in Willow Garage's grasping applications.

Although most of the sub-modules for flexible and reliable execution of dexterous manipulation tasks are available, their integration into a complete system is far from being solved. All components make certain assumptions about the state of the robot, its location in the environment or how the environment is set up and just plugging them together certainly would not work. Although current navigation components work relatively reliable, they might assume a fixed robot footprint and do not take into account the three-dimensional shape of the robot and the position of its arms. As a result, although the navigation algorithm works well, the robot might still collide or a valid path might not be found. Additionally, navigation accuracy is limited by the accuracy of odometry and the laser sensors used for self localization. In real world applications it is often not better than 2-5cm. Perception routines often suffer from

[2]http://ros.org/wiki/move_arm
[3]http://moveit.ros.org/
[4]http://openrave.org/

false positives and false negatives or objects that are detected only partially which makes maintaining an accurate belief state that the robot requires for action execution hard. Components such as motion and grasp planning are computationally expensive and suffer from perception inaccuracies as well.

Let us consider the example of fetching bottles from a refrigerator [Bohren et al., 2011]. The paper explains a system where different components such as perception, navigation and manipulation are orchestrated by a state machine written in a Python library called SMACH. Although the demonstrated behavior is quite impressive, it would not work well in a different environment, for a different refrigerator, if the bottles cannot be reached without removing other objects in front of them or if other unexpected conditions occur unless a lot of manual adjustments are made. To overcome these limitations, it is important for a high-level executive to not only call the respective sub-components at specific points in time as done in simple scripts, but more sophisticated reasoning and decision making systems have to be built. This includes the generation of locations for the robot to stand or for placing objects while basing this location generation on the integration of additional constraints, for instance keeping objects that are needed in future actions visible for the robot, i.e. not putting down objects at locations where they will occlude other objects that are needed later. More specifically, creating a high-level executive not only means calling the right components in the right order but to also find parameters that maximize the performance of the resulting actions and prevent errors. This also includes prediction mechanisms to integrate the future course of actions.

Reasoning systems developed in particular for robotic applications such as Knowrob [Tenorth and Beetz, 2013] and RoboEarth [Waibel et al., 2011] provide huge knowledge bases that allow robots to base their decisions not only on heuristics but on strong knowledge processing systems. However, the integration of the knowledge provided by these systems into robot control programs and feeding back information is non-trivial either. For instance, (sub-symbolic) detection results generated by perception need to be related to (symbolic) object instances as used in knowledge bases, as for instance described in [Blodow et al., 2010].

In this thesis, the author presents CRAM, a complete framework for implementing cognitive high-level robot control programs to enable robotic agents to execute their tasks more robustly, reliably and flexibly. The framework includes a plan language for

implementing concurrent reactive control programs. It is a modernized reincarnation of Drew McDermott's RPL [McDermott, 1993] with similar reasoning extensions as described by Michael Beetz [Beetz, 2000], adding support for multithreading to utilize the full capacity of current multicore processors and support for modern middle ware architectures such as ROS [Quigley et al., 2009]. Besides the plan language, CRAM provides a Prolog-like reasoning engine and many components to solve problems such as *where to stand* to perform actions, *is an object visible* for the robot from a certain location, *from where can an object be seen or grasped* but also components that allow the robot to reason about what it did, when and why. In addition to providing the means for implementing high-level control programs, the system also provides components for predicting the outcome of a plan. This, in combination with the other reasoning modules, allows the robot to base its decisions not only on knowledge about the current state of the world but to integrate the future course of actions.

1.1 Problem Description

In the domain of a human household, many problems can be reduced to a variation of pick-and-place tasks with the extension of actions such as pouring and opening and closing drawers and doors. To illustrate the system presented in this thesis, let us consider table setting as an example. To keep the example simple, the robot's task is to set a breakfast table for two persons with one plate, one mug and one knife for each seating location. The table is empty and the objects to be set on the table are stored in drawers at the counter as shown in Figure 1.2. A simplified plan for the robot to set the table for one person can informally be defined as follows:

1. Place the mug on the table.
2. Place the knife on the table.
3. Place the plate on the table.

To set the table for more persons, these steps need to be executed repeatedly.

The transformation of these plan steps from natural language to an actual CRAM plan that can be executed by the CRAM executive is not in the context of this thesis. We assume it to be provided by external components such as the KNOWROB Web im-

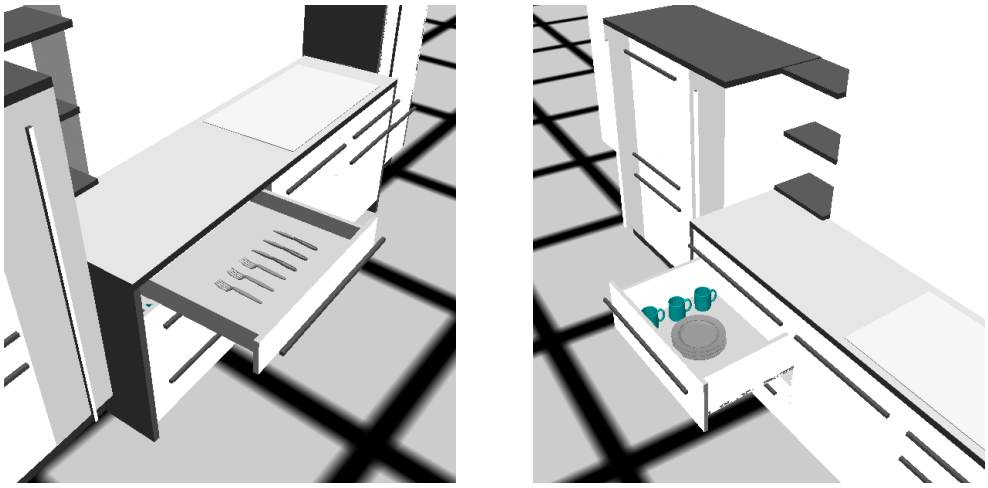


Figure 1.2: *In our example, silverware, mugs and plates are stored in two different drawers at the counter.*

port [Tenorth et al., 2010b]. A resulting CRAM plan corresponding to the above three actions is shown in Listing 1.1.

Listing 1.1: *Example plan for setting a table for breakfast. The robot's task is to put three objects on the table, a mug, a plate and a knife.*

```
1  (with-designators
2      ((mug (object '((type mug))))
3       (plate (object '((type plate))))
4       (knife (object '((type knife))))
5       (mug-location (location `((on table) (for ,mug)
6                                 (context breakfast)))
7       (plate-location (location `((on table) (for ,plate)
8                                   (context breakfast))))
9       (knife-location (location `((on table) (for ,knife)
10                                  (context breakfast)))))
11     (achieve `(location ,mug ,mug-location))
12     (achieve `(location ,knife ,knife-location))
13     (achieve `(location ,plate ,plate-location)))
```

As can be seen, this high-level plan is relatively abstract and therefore simple. Executing it requires to interpret each action instruction based on the current environment and the context of the plan. To execute it reliably, the system needs to rely on a tight

integration of reasoning, an accurate representation of the environment and action execution. The plan shown in Listing 1.1 executes three high-level sub-plans sequentially. First the mug is placed on the table, then the knife and finally the plate.

As can be seen, plan parameters (in our example the variables mug, plate, knife, mug–location, plate–location and knife–location that are bound by the with–designators macro) are specified as symbolic constraints expressed by key-value pairs. Solutions for these parameters must be entities that fulfill all specified constraints. For instance, a location for the mug on the table implies that we are searching for a location where the object is standing stable. In the context of a put-down action, the additional constraint that the location must be reachable by the robot must be taken into account. Finally, the context of the action, in our example table setting for breakfast, constrains possible locations even more, for instance by only allowing locations behind and right of the plate.

On the most abstract level, each pick-and-place action in the high-level plan requires the following steps:

1. Execute actions to make detecting and grasping the object possible (e.g. opening doors and drawers or moving other objects to temporary locations).
2. Detect the object.
3. Grasp the object.
4. Undo the actions of step 1.
5. Execute actions to make the put-down action possible, e.g. move objects out of the way.
6. Put down the object.
7. Undo the actions of step 5.

As can be seen the robot not only needs to infer parameters such as put-down locations but it also needs to infer additional actions to make the execution of the goal actions possible. Classically, these additional steps are added by a symbolic planner. However, symbolic planners require a complete world model beforehand and, in the worst case, need to re-plan after the detection of new objects or changes in the environment.

On the other hand, the plan steps above show that pick-and-place actions are highly structured, i.e. the steps to be performed only change slightly for different objects and for different pick-up and put-down locations. This allows to reduce the planning problems involved in such actions to very simple problems that can be solved and executed at run-time by relying on carefully hand-crafted low-level plans and different reasoning mechanisms.

In order to execute each of the actions, the robot first needs to detect the object of interest. First, this requires to find a location for searching for the object. For instance, cups are often stored in one specific cupboard and humans are assumed to clean up the environment which keeps these storage locations valid. This concept of environment stabilization is shown in [Hammond et al., 1995]. In order to find a location for an object, CRAM makes queries to KNOWROB for finding the typical storage locations of objects. In case of the first object in our example plan, the system infers a cupboard as the most probable location of a clean cup [Schuster et al., 2012]. To be able to pick it up, the system first needs to detect the object. Since the storage location is inside a cupboard and the cupboard door is closed in the initial situation, the robot infers that it needs to execute an opening action in order to be able to detect the object. Listing 1.2 shows a simplified version of the corresponding sub-plan to open a cupboard.

Listing 1.2: *Simplified version of the sub-plan to open drawers and cupboards.*

```
1 (def-goal (achieve (object-opened ?object))
2   (with-designators
3       ((open-action (action `((to open) (object ,?object))))
4        (location
5            (location `((to execute) (action ,open-action)))))
6     (at-location (location)
7       (perform open-action))))
```

As can be seen, two (symbolic) plan parameters are defined. One parameter (open-action) is describing an action to open an object and the other parameter (location) is the location for the robot to stand when performing the action. The location is inferred using a physics-based reasoning system that is able to generate solutions for such symbolic constraints. For instance, to find locations to execute a specific action, it first

infers a minimal set of trajectory points that need to be reached. In case of opening an object, the current location of the door or drawer handle and the final location of it are used. Then the reasoning system uses an inverse reachability map to find locations from which the robot can reach both locations. Additionally, the robot must not be in collision with the environment or other objects when reaching for these trajectory points.

After the door has been opened, the robot needs to move to a location from which it can detect cups in the cupboard. For that, it needs to be able to generate locations from which specific poses in space can be seen, taking into occlusions account. For instance, the robot should not stand behind the cupboard door because it would occlude all objects inside the cupboard. To solve this problem, CRAM uses OpenGL and depth maps to find locations from which a specific pose can be seen.

Next, in order to transport an object from the cupboard to the table, the robot needs to find a location from which it can pick up the object. It uses the same inverse reachability map that was used for finding a location to stand in order to open the cupboard door before. After navigating to that location, the robot uses motion and grasp planning modules provided by externally provided components to pick up the object and move it to a carry pose.

For putting down the object, the robot first generates a put-down location, i.e. the destination location for the object. The corresponding location designator for the mug is:

```
(location `((on table) (for ,mug) (context breakfast)))
```

The constraints encoded in this designator are relatively abstract. The system resolves the context to a seating location at the table and the location for the mug to a location on the back-left corner of that seating location as shown in Figure 1.3. After generating a destination location for the mug, the system needs to generate a location for the robot to stand in order to put down the object. This again uses the inverse reachability map. Finally, a put-down action that moves the arm with which the object has been grasped is performed.

The same sequence of actions is repeated for the other two objects which leads to a table set for breakfast with the specified objects for one person.

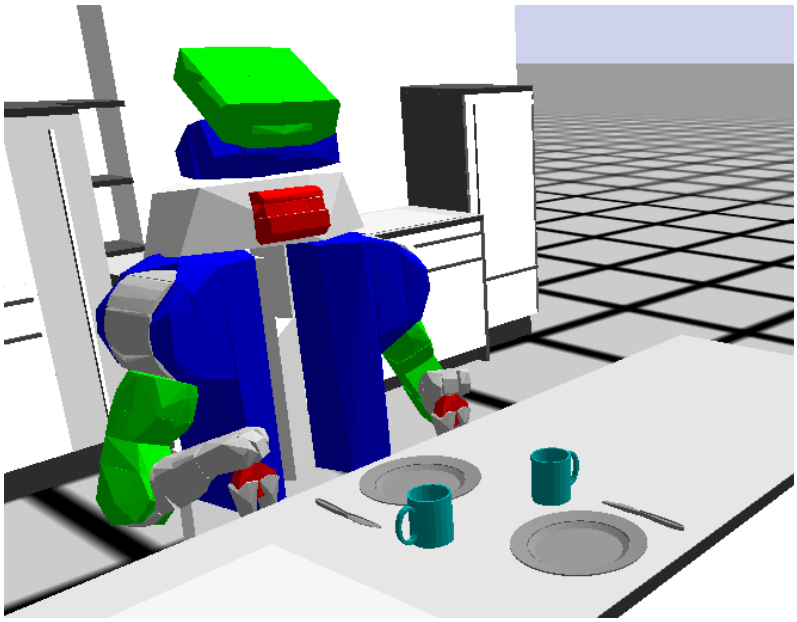


Figure 1.3: *Table set for breakfast for two persons. This is the final configuration for the example in this Section.*

Two flaws in the above sequence of actions are evident. First, to be more efficient, the robot could grasp the mug and the knife at the same time with two different arms if they are reachable from the same location. Second, putting down the plate after the knife can lead to problems because the plate is grasped with two arms, from both sides, and the robot might be in collision with the knife as can be seen in Figure 1.4. The former flaw can be resolved by executing top-level plan steps for picking and placing the objects in parallel and relying on powerful resource management mechanisms provided by CRAM. The latter flaw can be resolved by using fast temporal projection, a prediction mechanism to generate execution traces of possible plan execution episodes in a lightweight and fast simulation environment. Based on these execution traces, the system can infer flaws such as problems caused by the order in which actions are executed (e.g. putting down the knife before putting down the plate) or unwanted occlusions, e.g. by placing objects in front of other objects that are needed later. The projection mechanisms presented in this work can be used to generate locations taking into account the future course of actions and to infer required partial orderings to minimize potential problems.

To summarize, reliably executing a task such as setting a table with a robot is surprisingly complex and involves the integration of reasoning on different levels of ab-

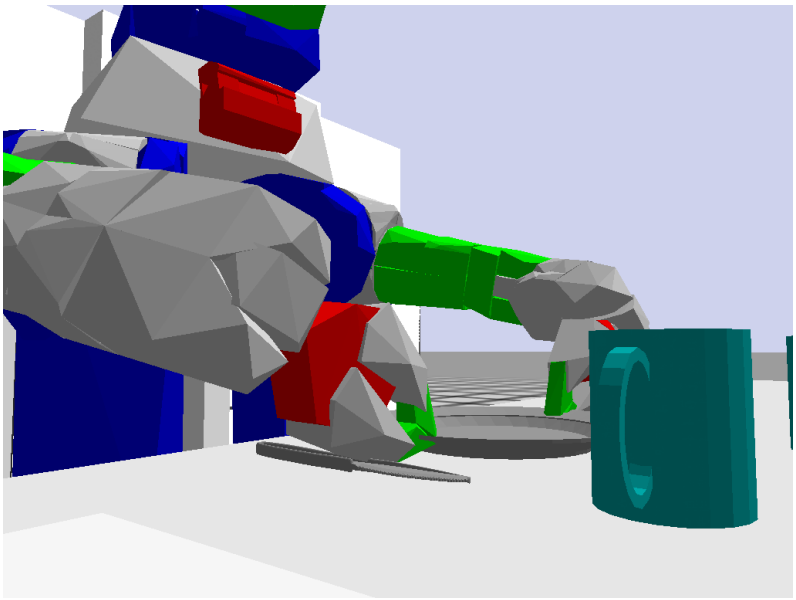


Figure 1.4: *When manipulating a plate after the knife has been placed, the knife is blocking the put-down action because the robot's gripper collides with it.*

straction, including temporal reasoning, quantitative and qualitative spatial reasoning and reasoning about the robot's own actions. A system architecture that allows for sophisticated error handling and recovery behaviors and the seamless integration of reasoning mechanisms is required. Such a system architecture should abstract away from robot-specific properties such as the exact hardware to allow for the implementation of high-level actions that can be executed on different robot platforms [Tenorth and Beetz, 2012].

1.2 System Architecture

In this thesis, the author describes the Cognitive Robot Abstract Machine (CRAM), a framework containing components for action execution and reasoning. The current implementation is designed to be applied on mobile service robots such as the PR2 or TUM-Rosie but is not restricted to these platforms. For instance, some components have successfully been tested in the context of the Saphari EU project [5] on a stationary robotic platform based on two DLR arms.

[5] http://www.saphari.eu/

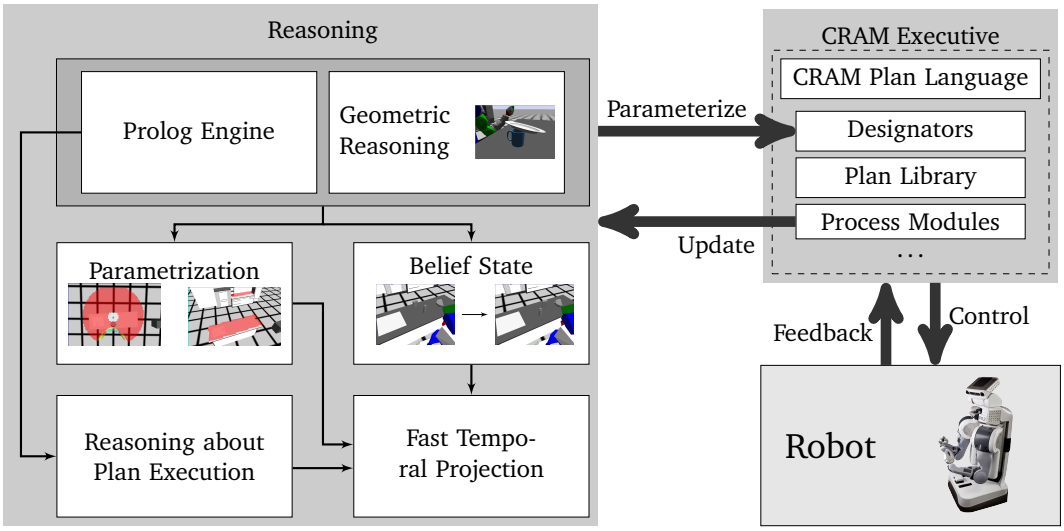


Figure 1.5: *Overview of the different* CRAM *components presented in this thesis. Components are grouped into two main areas, reasoning and the executive. The executive updates information stored in the reasoning system and uses the reasoning components for planning and decision making.*

Figure 1.5 gives a high-level overview of the CRAM system and the modules that were developed in the context of this thesis. CRAM is not specific to a certain robot platform but communicates with all hardware-specific modules through well-defined interfaces which makes it highly flexible and general. In particular the reasoning components were shown to be robot independent. The central component of a robot controlled by CRAM is the CRAM executive. It is responsible for executing (generated or hand-coded) plans. A set of modules and libraries including *designators*, the CRAM *plan library* for creating pick-and-place behavior and *process modules* for communicating with the robot's hardware by sending control commands and receiving feedback messages were implemented. Since CRAM plans are high-level (symbolic) descriptions of the actions to be executed, plan parameters such as locations for the robot's base to stand have to be generated and a consistent geometrically accurate belief state has to be maintained. The corresponding functionality is provided by CRAM's reasoning components. The base is a Prolog engine with extensions for geometric and physics-based reasoning. Extensions provide the functionality for generating plan parameters such as locations and for simple entity resolution and maintaining a consistent belief state. Higher-level libraries provide the means for reasoning about plans and for projection plans. In the following, a short overview of the components that were implemented in the context of this theses is given.

The CRAM Plan Language. The CRAM plan language is a *domain specific language* for implementing highly concurrent control programs. It includes special language support for executing multiple instructions in parallel, for monitoring them and provides sophisticated and powerful error handling mechanisms that are not restricted to single operating system threads. It provides support for propagating error objects across thread boundaries and for terminating, evaporating, suspending and resuming control programs or parts of it. Additionally it provides support for semantic annotations of plans which is required to ground them in CRAM's reasoning system to allow for reasoning about plan execution and for projecting plans.

The CRAM plan library. The plan library contains plans for all high-level actions such as picking up and putting down objects, opening and closing drawers, navigation and perception. These plans integrate external reasoning components provided by, for instance, the KNOWROB system and make extensive use of the internal reasoning mechanisms provided by the CRAM reasoning modules. Plans are written to be robot independent, declarative and universal. This means that a plan does not reference any robot specific information without querying it from a knowledge base. Plans are semantically annotated, i.e. their purpose is defined as a Prolog expression that is grounded in the underlying reasoning system. Informally speaking, universal plans [Schoppers, 1987] are plans that can be executed in any (or at least in many) different environment representations. In contrast to classical plans that require certain preconditions to hold before they can be executed, universal plans execute actions for making their preconditions true directly.

Designators. Designators are used to interface robot-specific components of the executive from high-level plans. While representing parameters for lower-level components of the robot on a purely symbolic level, they are resolved by the CRAM reasoning system to generate low-level parameterizations such as commands for ROS drivers. For instance, the command sent to the navigation process module is an action Designator as follows:

(an action (to navigate) (goal at-table))

Parameters defined by Designators can be incomplete and the missing information is filled in using the different reasoning mechanisms included in CRAM. Besides the

parameterization of actions and locations, Designators are used to describe objects involved in plans. For instance, the Designator corresponding to any cup in the cupboard can be written as follows:

(an object (type cup) (in cupboard))

Please note that cupboard is a location Designator describing all possible locations in a cupboard. One unique property of object Designators is that they are used to track object identities over time in the robot's belief. Although the robot's perception system in CRAM creates a new object Designator for each object detection, Designators that reference the same object are linked together.

Process modules. Robot specific code, or more generally, code that directly interacts with the environment, is grouped in so-called process modules. High-level plans interact with process modules by sending them symbolic descriptions of the actions to be performed in the form of action designators. Interaction in this context not only means physically moving objects or the robot but also perceiving the environment or, for instance, communicating with humans. Most robots that are used in the context of mobile manipulation in domestic environments implement at least four basic functionality blocks:

- Navigation: navigation is probably one of the oldest and best researched areas in mobile robotics research. That's why the CRAM executives assumes it to work relatively reliable, i.e. the high-level executive can abstract away from path planning and navigation.

- Manipulation: to be useful in domestic environments, robots need to interact with their environment. The code of the manipulation process module is more complex than other process modules since many more decisions have to be made. For instance, it needs to select an appropriate grasp planner, provide enough information for obstacle avoidance, decide on the grasp to use etc. To achieve the best performance, most of these decisions need to include the current context of the action and additional knowledge about the environment. For instance, when grasping a mug that is filled with coffee, it is important to keep it upright. When putting down an object temporarily, e.g. to be able to reach another object (i.e. to unblock another object), the orientation does not matter

while putting down objects when setting the table requires to use the correct orientation.

- Perception: being able to detect and re-detect objects that are manipulated by the robot is vital for performing pick-and-place actions in a domestic, changing and uncertain environment. The perception process module converts object designators, e.g. *(an object (type mug))*, to calls to ROS components for detecting the corresponding object. If one or more matching objects could be detected, the process module converts the result back to symbolic object designators to be used by subsequent actions.

- Re-positioning of sensors: most robots provide ways to change the position of sensors on the robot, for instance with a pan-tilt-unit. Additionally, robots such as the PR2 have cameras in their forearm. The functionality to move a sensor that is later used by the perception process module to a configuration that allows to see a specific object is encapsulated in this process module.

Prolog reasoning engine. CRAM contains a full-featured reasoning engine based on a Prolog interpreter. In contrast to classical Prolog reasoning engines, CRAM's Prolog implementation is optimized for easy and seamless programmatic use of Prolog queries by encapsulating the computational context of a Prolog query in a lazy list object representing all solutions of the query. In contrast to, for instance SWI-Prolog[6] [Wielemaker et al., 2012], multiple queries can be stored and subsequent solutions for all stored queries can be requested without explicitly starting multiple Prolog engines. Due to the close integration of CRAM's Prolog engine, it can easily be extended by other CRAM components, for instance the geometric and physics based reasoning engine.

Geometric and physics based reasoning. In highly dynamic environments such as a human household, the ability to create a geometrically accurate representation of the robot's environment and to reason about spatial relations, stability, visibility and reachability is essential for reliably executing actions. CRAM provides a library that uses the Bullet physics engine, OpenGL rendering and inverse kinematics computation to define predicates in the Prolog engine in order to reason about the physical and

[6] http://www.swi-prolog.org

geometric aspects of the environment. This allows, for instance, to infer if the robot will be able to see a specific object from a specific location and the objects that are possibly occluding it.

Belief state. Decision making during plan execution and the resolution of plan parameters often requires a detailed and accurate model of the current environment. However, adding new information and updating it from sensory input to an internal representation of the environment is considered a hard problem since a system has to deal with ambiguities and has to track object identities. CRAM provides a relatively simple solution for this problem by using the geometric and physics based reasoning system to represent the environment. To decide if a new object instance for sensory input has to be generated, if an existing instance has to be updated or if instances have to be removed, the system uses visibility reasoning. Objects that should have been detected but were not are removed and new detections that overlap with existing objects and are of the same kind cause instance updates. The resulting representation of the environment is accurate and consistent in most cases and is used as the default database for all geometric reasoning tasks.

Parameter resolution. In CRAM, plan parameters such as the locations for putting down objects and the locations for the robot to stand while performing specific actions, e.g. picking up objects, opening and closing drawers or to see a specific object, are specified symbolically using designators. In most cases, a great number of solutions is valid for a given symbolic description. For instance, an infinite number of solutions exist for a designator that just specifies any location on the table. To find a specific parameter, i.e. a position in three dimensional space and the corresponding orientation, CRAM uses sampling from a distribution that is generated taking into account the world state stored in the geometric reasoning system, the constraints specified in the designator, an inverse reachability map and visibility reasoning.

Reasoning about plan execution. To perform actions reliably, robots need to have an understanding of the actions they are executing. They need to be able to decide if a sub-plan terminated successfully or if it caused the top-level plan to fail. Even if no error occurred, a sub-plan can have failed or it could have revoked a goal that was achieved by a previously executed sub-plan and hence cause the top-level plan to not achieve all its goals. Additionally, the ability to understand its plans enables

robots to analyze previous executions of a plan or simulated plan executions to infer flaws and optimize and fix plans and to generate input data for planning algorithms. CRAM provides the means for recording extensive execution traces and to make inferences about plan execution based on these traces. This allows for finding flaws, for implementing "deeper" error handling not just based on the propagation of exception objects but on making symbolic queries on the execution trace and for extracting data to be used by learning algorithms.

Temporal projection. Resolving plan parameters just based on the current world state is not sufficient in many cases. For instance, objects put down earlier in a plan might block put-down locations for objects that are put down later because of poorly chosen locations. Temporal projection of plans allows for predicting the effects of a plan by "simulating" it in the geometric and physics-based reasoning engine. By then analyzing these projected episodes using the reasoning mechanisms for making inferences about plan execution, the corresponding CRAM libraries allow for integrating all actions of a plan in the generation of plan parameters such as locations.

1.3 Contributions

In the context of this thesis, a number of libraries and algorithms has been developed that allow for the creation of a full featured cognitive execution architecture for mobile robots performing actions in domestic environments such as a human household. Most of the work described in this thesis has been released as CRAM components.

The main contributions of the work presented in this thesis are:

1. The CRAM *Plan Language*, an extension and reimplementation of RPL [McDermott, 1993]. The CRAM Plan Language extends RPL with explicit support for programming real autonomous robots with all their complications. In contrast to RPL programs which are evaluated by an interpreter, the CRAM Plan Language is implemented using plain Common Lips. This implies that it is not interpreted but compiled to native machine code. The specific components of the CRAM Plan Language's execution environment that were implemented are:

- The CRAM *Plan Language* itself, a domain specific programming language for implementing high-level control programs on robots, featuring concurrent execution, synchronization, reactivity and sophisticated error handling.

- A library for abstracting away from the actual robot's hardware to allow for the implementation of a generic library of high-level plans. This library defines and implements the so-called *Process Modules*.

- A library for *Designators*, symbolic descriptions of plan parameters. The library defines interfaces for plugging in different resolution mechanisms and reasoning components.

2. Built-in reasoning mechanisms that allow programmers to state plans more vaguely by letting the robot make the respective decisions based on the execution context. These reasoning mechanisms enable the robot to infer the most appropriate course of actions by taking all available execution information into account. The specific reasoning components implemented in the context of this thesis are:

   - A Prolog-like reasoning engine that can easily be extended and embedded in programs.

   - Reasoning extensions for quantitative, physics based reasoning using a geometrically accurate three dimensional representation of the robot's environment, a physics engine and OpenGL rendering.

   - Reasoning extensions to reason about plan execution, including a mechanism for recording execution traces from plan execution.

   - A framework for resolving Designators based on sampling from probability distributions. These distributions are generating by compiling designator properties to separate distributions and combining them.

   - The implementation of different probability distributions for resolving designator properties related to visibility and reachability.

- A system for fast plan projection based on the physics-based reasoning engine.
- The integration of projection in location designator resolution which enables it to generate locations not only from the current world state but also from all actions that are executed in a plan.

3. A plan library for pick and place plans and an executive for executing these plans on TUM-James and TUM-Rosie. More specifically, the following components were implemented:
   - A library of high-level plans for picking up, putting down objects and opening and closing articulated objects.
   - The implementation of comprehensive plan execution systems for the two autonomous mobile manipulation platforms TUM-James and TUM-Rosie.

1.4 Reader's Guide

This thesis describes all components of CRAM that allow for implementing highly dynamic, robust cognitive robot behavior on mobile service robots such as the PR2. It is organized as follows:

Chapter 2. gives a detailed explanation of the core CRAM components, namely the CRAM Plan Language and the CRAM reasoning system which is based on a Prolog-like language. Higher level libraries such as Designators which are used for symbolically describing parameters, Process Modules which are the interface to the robot's hardware and the CRAM plan library are explained.

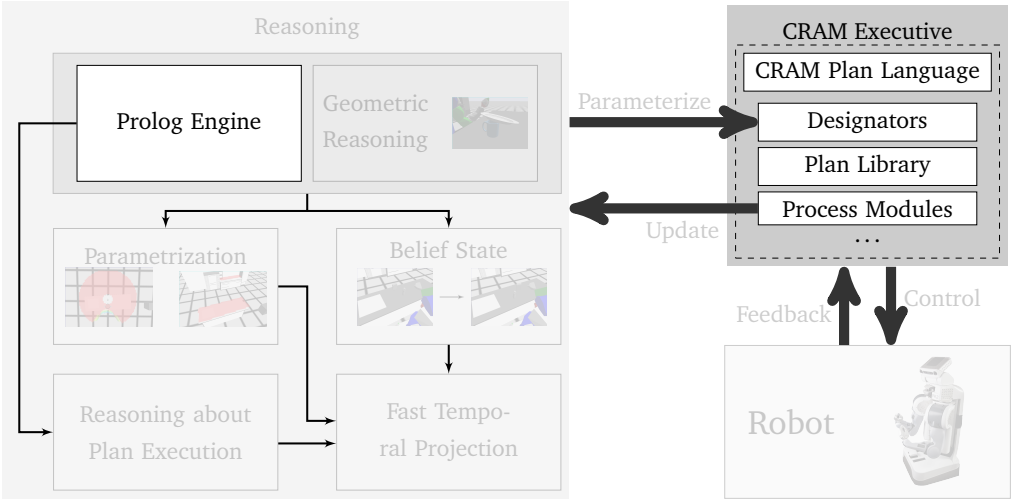
Chapter 3. presents the implementation of a geometrically realistic physics based reasoning engine. Details on the implementation, which is based on the Bullet physics engine, OpenGL and inverse kinematics computation are given and the belief state representation used in CRAM is explained. The second part of the chapter explains the recording of execution traces and how reasoning about plan execution is implemented.

Chapter 4. presents details on the sampling based algorithm for generating locations based on symbolic constraints. Details are given on how distribution functions are generated from symbolic constraints and how they are grounded. This chapter relies on the functionality introduced in Chapter 3.

Chapter 5. finally combines most of the components introduced in the previous chapters and shows how a fast, still accurate, temporal projection mechanism is implemented and how it can be used to find plan parameters that are not only generated based on the currently know world state but also on future actions of a plan.

Chapter 2

The CRAM Software Architecture



Enabling robots to perform complex actions such as cooking or cleaning up in dynamic, changing and unpredictable environments such as a human household bears completely different challenges than, for instance, industrial robotics. While industrial robots repeatedly perform the same sequence of actions in a more or less static environment, robots in domestic environments need to make sophisticated decisions based on the current environment configuration while performing a rich set of activities and it is important that they are capable of recovering from a huge variety of different errors that might occur during action execution. Many cognitive architectures have been proposed [Vernon et al., 2007], such as the 3T architecture [Bonasso et al., 1997] or Icarus [Langley and Choi, 2006]. However, none could prove so far to be powerful enough to handle the complexity that comes with performing tasks in a human household.

CRAM is a software toolbox that provides a set of libraries to implement complex actions such as cooking or cleaning up that require a tight integration of action execution, reasoning, decision making, execution monitoring and failure handling. Its core libraries include of a domain specific programming language, the CRAM Plan Language, a full-featured Prolog-like reasoning engine and support for so called Designators, symbolic descriptions of plan parameterizations such as objects, locations and actions. Figure 2.1 gives an overview of the architecture of an executive implemented using CRAM. Plans are defined using the CRAM Plan Language and stored in a plan library. Plan parameters such as locations, actions and objects are represented on a symbolic level using Designators. This allows for abstracting away from the robot's actual hardware. To actually communicate with the robots hardware, Process Modules receive commands as Action Designators, resolve them to generate hardware specific commands and send them to the robot's lower level components, e.g. ROS actions such as navigation or arm control. To update the robot's belief state, process modules can emit events. Although designator resolution can be implemented on arbitrary reasoning components, the current implementation in CRAM uses CRAM's powerful reasoning mechanisms in combination with external knowledge processing systems such as KNOWROB.

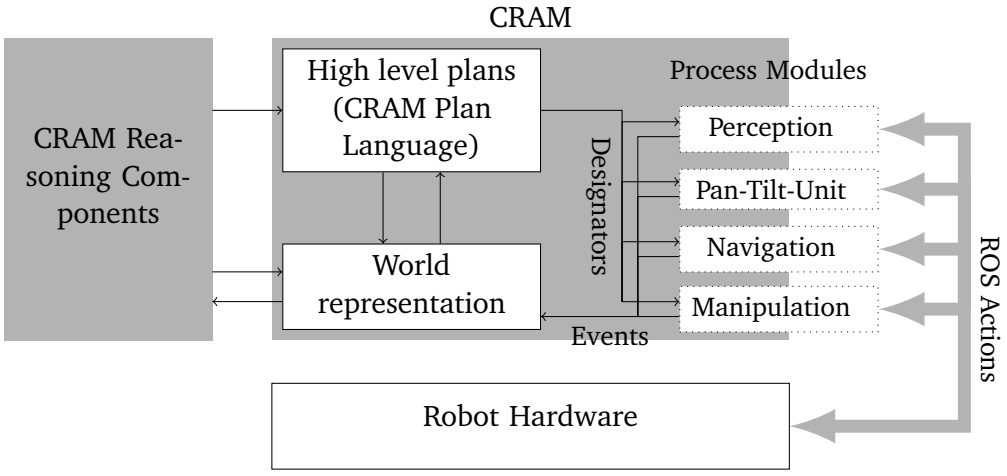


Figure 2.1: *The architecture of our* CRAM *based executive.*

The programming language of choice for implementing such a system is Common Lisp because it provides enormous flexibility through its macro system and its strong integration of the functional programming paradigm.

In this chapter, we describe the implementation of the core components of CRAM, namely the CRAM Plan Language, the CRAM Reasoning Engine based on a Prolog interpreter, symbolic plan parameterization based on Designators and Process Modules. Finally, an overview of the currently implemented plans in the CRAM plan library is given.

2.1 The CRAM Plan Language

The history of reactive plan execution goes back to the 1980s with the work of Firby [Firby, 1987] and [McDermott, 1993]. In particular in complex, changing and unpredictable domains, the idea of reactive planning is still attractive. Applying classical planning in is either really difficult or really easy. It is really difficult if the environment's geometry is taken into account as required for instance in complex unstructured environments such as a human household. Classical symbolic planning is really easy if applied to high-level purely symbolic environment and action specifications.

The CRAM Plan Language is based on Drew McDermott's *Reactive Plan Language (RPL)* [McDermott, 1993]. RPL was an interpreted Lisp-like language for the implementation of reactive control programs for robotic agents. However, it was only simulating concurrency by shuffling the order in which expressions are interpreted randomly. Although RPL could be extended to allow for the evaluation of arbitrary Lisp code, it did not allow for evaluating arbitrary RPL code in Lisp. While trying to imitate the semantics of RPL, the CRAM Plan Language overcomes these limitations and allows for use all processors on modern multi-core CPUs and to integrate with existing Common Lisp libraries.

The CRAM Plan Language is designed to support the programmer with the implementation of CRAM plans, control programs that cannot only be executed but also reasoned about, and that are reactive, concurrent and tightly integrate with reasoning components. The CRAM plan language is a full featured general purpose programming language, implemented based on Common Lisp and designed for the special requirements in writing control programs for mobile robots performing complex plans. In particular, this includes first class support for parallel execution and monitoring of concurrent processes, the integration of sensory input and synchronization. Special exception handling mechanisms based on Common Lisps powerful condition system

but with support for propagating error objects across thread boundaries is integrated in the language, too. Another special feature of the CRAM Plan Language is its tight integration with reasoning for instance by supporting plan annotations using logical expressions grounded in a reasoning engine in order to reason about plan execution and higher level language forms and libraries to integrate reasoning in the decision making process used in CRAM plans.

2.1.1 Language Syntax

The CRAM Plan Language is implemented as a domain specific language in Common Lisp. The syntax of programs implemented in the CRAM Plan Language hence is inherited from Common Lisp and the complete standard library and language features of Common Lisp are available in CRAM programs.

Common Lisp is a general purpose multi-paradigm programming language strongly emphasizing the functional programming paradigm. However, it also supports object oriented programming through the Common Lisp Object System (CLOS) and imperative programming. It supports lexical and dynamic scoping and provides an optional type system. A number of powerful commercial and open source compilers are available. While CRAM and the CRAM Plan Language is mostly written in compiler independent ANSI Common Lisp, in particular for multithreading, it relies on libraries specific for Steal Bank Common Lisp (SBCL) [1]. SBCL is an industry strength open source implementation of a Common Lisp compiler that outputs high performance native machine code and that has a strong type inference system.

The most important reason for choosing Common Lisp to implement CRAM's plan language is its powerful macro system which allows for the relatively easy implementation of domain specific (compiled) languages. In contrast to other macro systems that most often rely on simple string replacement, for instance C/C++'s macro system, Common Lisp macros are normal functions that get a Lisp expression as input and return a transformed Lisp expression. In other words, Lisp macros operate on code instead of data. The CRAM Plan Language is implemented as a runtime system

[1] www.sbcl.org

written and a set of macros that transform CRAM plans to Common Lisp code using this runtime.

To make it easier for the reader to understand the code presented in this paper and the implementation of CRAM language internals, a brief overview of the most important concepts and features and Common Lisp and its syntax is given in this section.

S-expressions. S-expressions are the basic building blocks in all Lisp dialects. Every lisp program is defined of an ordered sequence of S-expressions. [McCarthy, 1960] defines S-expressions as follows:

1. Atomic symbols are S-expressions.
2. If e_1 and e_2 are S-expressions, then $(e_1 \; . \; e_2)$ is also an S-expression (note: the "." separates the first element of a list from the rest).

Essentially, all lisp programs consist of expressions that are built from atoms, atoms in parenthesis and nested forms. In other words, S-expressions form nested lists. When the Lisp system evaluates these lists, it interprets the first element of such a list which must be an atom as the name of the special form, macro or function to execute and the rest of that list as parameters. If such parameters are not atoms, they are first evaluated recursively according to the evaluation rules of S-expressions. The following example shows how a nested arithmetic expression is evaluated:

$$(+\ 5\ (*\ 2\ 3)) \rightarrow (+\ 5\ 6) \rightarrow 11$$

Basic Language Features. Common Lisp is a general purpose programming language. Functions are defined using the special form `defun`:

```
(defun <name> (<parameter>*) <body-form>*)
```

In addition to fixed size parameter lists, Common Lisp also supports optional and keyword parameters.

Besides numbers, strings and lists (build of cons cells, i.e. pairs of pointers) and arrays, Common Lisp defines symbols as an additional basic data type. A symbol is an identifier and can have two values bound to it, a value and a function. If the symbol appears in an S-expression as a function name, i.e. as the first element of the S-expression, its function value is used to perform a function call. If the symbol appears anywhere else in the S-expression, it evaluates to the value bound to it which can be any legal Common Lisp value, for instance a number, a string, a list or a function object.

Lexically scoped variables are defined using `let` blocks. For instance, the following code shows how to define and bind to variables `a` and `b` and bind them to two numbers:

```
1 (let ((a 1)
2       (b 2))
3   ...)
```

Besides lexical scoping which also allows for closures as defined in [Sussman and Jr., 1975], Common Lisp also supports dynamically scoped (global) variables, defined by the forms `defvar` and `defparameter`. The difference between the two forms is that the former does not rebind the value of the corresponding symbol if it exists already while the latter always rebinds the value to the default value specified in the respective definition form.

Although it has strong support for functional programming, it is not a purely functional language. It allows for side effects, i.e. the value of variables can be changed. For updating variable values, the most important because most generic special form is `setf`. It can be used to update member variables of classes and structs, to set array elements and to set the value of normal values. For instance, the following expression sets the value of the variable `a`:

```
1 (setf a 2)
```

Common Lisp is a strongly typed language with dynamic typing of variables. That means that each value as a specific, well defined type while variables can reference values of arbitrary type. Basically, that means that variables do not have a type but the type is bound to the actual value. However, Common Lisp optionally supports the declaration of types for variables which causes the compiler to produce type errors at compile time if possible and at run time otherwise in case a value with a wrong type is assigned to a variable.

Conditions and error handling. Common Lisp provides an extremely powerful error-handling mechanism that differs from most commonly used programming languages. The system is based on signaling *conditions*. Conditions do not just represent errors but are rather signals that can be emitted by code. In contrast to the exception handling mechanisms in Java, C++ or Python where the stack is unwound until a corresponding handler is found, Common Lisp signal handlers are executed in the stack frame that signaled a condition. Several classes of signals are defined per default, including warnings and errors. In other words, errors are just special conditions that cause the Lisp environment to enter the debugger if they are not handled. Error conditions are signaled by executing the function `error`, similar to `throw` in Java and C++ or `raise` in Python. Executing signal handlers in the caller's stack frame allows for so-called restarts, special exit points that provide the functionality to recover from errors. For instance, if in a network library sending fails because a socket has been closed, the library can provide a restart for reconnecting the socket. An error handler can then decide if it makes sense to try reconnecting and execute the corresponding restart. Condition handlers are established with the forms `handler–bind` which allows for executing a function in the stack frame of the error call and `handler–case` which is similar to try-catch blocks in Java or C++ where the error handler is executed in the stack frame of the `handler–case` form after unwinding the stack.

CLOS. CRAM makes extensive use of the Common Lisp Object System. Common Lisp's approach to object orientation is slightly different from languages such as Java and C++. The main difference is that classes do not contain methods but only data slots. CLOS supports multiple inheritance, automatically generated constructors for initializing data slots and is extremely extensible because of a powerful metaobject protocol [Kiczales and Rivieres, 1991]. Methods are implemented by analyzing the type of their parameters when they are called and calling the most specific method

matching the type. The following example shows the definition of the class object:

```
1 (defclass cup (object)
2   ((handle-pose :initarg :pose :reader handle-pose))
```

The class contains one member variable, handle-pose, and a reader method (i.e. a getter in Java terms) is defined. It inherits from one parent class, the class object, which is the base class for all objects.

To improve readability, classes in this thesis are formatted in a more readable format. The object class of the example above is written as follows: Each class defines its own

Class 1 An example definition of a class object.

class CUP (**superclasses:** object)
slot HANDLE-POSE: The pose of the handle of the cup that can be used for grasping it.

type and is a sub-type of all its parent classes. Methods are defined in the context of generic functions. A generic function is a function that analyzes the types of specially declared parameters and selects methods based on these types and other rules such as the method combination to use. Then, the matching methods are executed and the result is computed based on the method combination. To define a generic function, the macro defgeneric is provided by Common Lisp. The following example shows the definition of the generic function grasp that is supposed to grasp an object.

```
(defgeneric draw (object side))
```

Please note that the above declaration of the generic function does not attach any code. To implement a method for this generic function, the macro defmethod is used:

```
(defmethod draw ((object cup) side) ...)
```

The expression (object cup) used to define the first parameter of the method declares

that this method should be executed for all instances of class `cup` (and its subclasses if no more specific methods are defined) and to bind the corresponding instance in the lexical context of the method's body to the variable `object`.

Besides the definition of classes and simple methods, the Common Lisp Object System supports methods that are selected based on the type of more than one parameter, user-defined method combinations, e.g. to execute all methods matching a type and combining the corresponding result values and meta-classes to change the internal representation of classes.

Macros. Macros are one unique feature of all Lisp dialects. Common Lisp macros are similar to normal functions with the difference that they are evaluated at compile time. Essentially, macros are functions that are applied on Lisp code, transform it and return the new code. The most important difference of macros to functions is that they allow the programmer to control the evaluation of code which is in particular useful for the implementation of domain specific languages.

For instance in Common Lisp, the logical or operator `or` is implemented as a macro. It evaluates all forms in its body sequentially until one returns a non-NIL value and returns that value. No further forms are evaluated. Let us consider the following example:

```
1 (or nil (setf a 1) (setf a 2))
```

The first expression, `nil`, evaluates to itself and thus is `nil`. The second form sets the value of the variable `a` to 1 and returns that value. According to the definition of the `or` macro, evaluation must stop now, i.e. `a` will not be set to 2. Internally, the `or` macro returns code equivalent to the following listing:

```
1 (let ((tmp-1 nil))
2   (if tmp-1
3       tmp-1
4       (let ((tmp-2 (setf a 1)))
5         (if tmp-2
6             tmp-2
7             (let z ((setf a 2))
8               (if z tmp-2 nil))))))
```

As can be seen, the macro must be recursive. It must bind the result of each expression to a temporary value and expand to nested `if` expressions. The following listing shows one possible implementation of the `or` macro:

```
1 (defmacro or (&body forms)
2   (let ((var (gensym)))
3     (when (first forms)
4       `(let ((,var ,(first forms)))
5          (if ,var ,var (or ,@(rest forms)))))))
```

As can be seen in the example, macros are defined with the macro `defmacro`. Parameters are specified similar to `defun` with the difference that nested lambda lists can be specified that are matched against the lisp code in the body of the macro when it is used. The identifier `&body` is just an alias for `&rest` and indicates that an arbitrary number of parameters can be passed. In the above example, the list of parameters, i.e. the body forms of `or` is bound to the variable `forms`. Line 2 generates an unused symbol to be used as the name of the required temporary variable to avoid any conflicts between already existing variables. Line 3 is responsible for only generating code if there are body forms left in the recursive expansion of the macro. Finally, line 4 and 5 assemble the code the `or` macro expression should be transformed into. Basically, the backquote reader macro suppresses evaluation of the following expression. Only expressions prefixed with a komma are evaluated and the result is inserted at the corresponding positions in the list. Expressions prefixed with `,@` are evaluated and

the result is spliced, i.e. the result of such an expression must be a list where each element is inserted. For instance, `'(a ,@(list 1 2 3) b)` evaluates to `(a 1 2 3 b)`. Macros are a widely used tool in CRAM for instance to implement all special forms of the CRAM Plan Language or for lexically binding variables to inferred variable bindings with the `with-vars-bound` macro.

Delayed Evaluation and Lazy Lists. In contrast to languages such as Haskell, Scala, or Clojure, Common Lisp does not provide built-in support or libraries for delayed evaluation. However, an extension to implement delayed evaluation is rather simple since only a proxy object needs to be created that contains a callable object for the generation of the value. Delayed evaluation essentially allows the programmer to assign a generator function that computes a value *when it is needed* instead of binding the value itself to a variable. Essentially, delayed evaluation "freezes" a computational context in the proxy object, i.e. the code that is using a value can be completely unrelated to the generating code.

Lazy lists are lists where elements are generated by a generator function only when they are accessed. This allows to define lists of infinite length and perform computations on them. One application for lazy lists in CRAM are bindings for logical variables in CRAM's reasoning component. The system might generate an infinite number of solutions to a logical query. By returning a lazy list instead of an ordinary list, the system does not require to limit the set of solutions and only the solutions that are really processed by the system are generated. To generate a lazy list in CRAM, the macro `lazy-list` is provided. The following example shows how to create the infinite list of all even natural numbers:

Listing 2.1: *Lazy list for generating the infinite sequence of natural numbers.*

```
1 (lazy-list ((n 0))
2   (cont n (+ n 2)))
```

The macro expects a list of variable bindings that should be enclosed in the lazy list, i.e. a list of variables that are required to generate an element of the lazy list and that need to be updated in each generation step. The format of the variable bindings is similar to Common Lisp's `let` form, i.e. it is a list with elements of the form `(name value)`.

In Listing 2.1, only one variable is needed for generating the list, the current number. In the lexical context of the lazy–list macro, local functions for yielding an element to the list, for re-evaluating the generator function (i.e. the body of the macro) with different variable assignments and without generating an element and to end the lazy list are bound:

- (cont element &rest variable–assignments): Adds the element element to the list. The generator function is only executed again when the user request a new element of the lazy list. The additional parameters of the function are the updated variable bindings in the same order as they were specified in the lazy–list macro.
- (next &rest variable–assignments): Does not generate a value but immediately re-executes the generator function with new variable bindings.
- (finish element): Finishes the list with the last element element.

When one of the above functions is called, a non-local exit of the body of the lazy–list macro is performed. If the body of the macro terminates, i.e. does not call one of the above local functions, the list is finished.

The user can access the first element of a lazy list using the function lazy–car which is similar to car in Common Lisp. To expand one more element, the system provides lazy–cdr which generates a new cons cell containing a new element and the generator function. In addition, to generate all required elements and return one with a specific index, the system provides lazy–elt. To expand a complete lazy list the function force–ll is provided. It essentially converts a lazy list into an ordinary Lisp list. Please note that if the lazy list is infinite, force–ll will never terminate.

Besides the basic implementation of lazy lists, CRAM also provides equivalents to Common Lisp's mapper functions as well as reduce, namely lazy–mapcar, lazy–mapcan, lazy–fold (the equivalent to reduce), lazy–append and a few other helper functions.

2.1.2 Definition of Terms

Let us consider the following very simple example plan for navigating the robot to a goal location.

Listing 2.2: *Example navigation plan*

```
1  (def-goal (perform (navigate-to ?location))
2    (pursue
3      (wait-for *navigation-goal-reached*)
4      (whenever (*navigation-error-occurred*)
5        (with-task-suspended navigation
6          (fix-error)))
7      (:tag navigation
8        (loop repeat do
9          (update-navigation-command
10           (calculate-navigation-command ?location)))))
```

The specific properties of the language forms used in the above plan will be explained in the remainder of this section. We will, however, explain the different terms we use in this thesis based on the example above. The pursue statement executes its three child forms in parallel and terminates as soon as one terminates. The wait-for form terminates as soon as the corresponding variable changes its value to true. Both, whenever and loop in the navigation sub-task never terminate. The whenever form is responsible for handling navigation errors, for instance if localization is lost, it can temporary halt the navigation action and execute a specific action for re-localizing the robot. The loop repeatedly recomputes and sends the command for robot's controllers based on the current position and the goal location.

The complete code snippet above shows a *plan*. Plans are similar to functions or methods Common Lisp with the difference that they provide enough information for a reasoning engine. In particular, the property that characterizes the plan above is that it contains the *goal* (perform (navigate-to ?location)) at line 1.

The CRAM Plan Language provides mechanisms for reactivity, i.e. for executing code whenever some sensor value or a value derived from sensors changes. One example is the wait-for statement in line 3 that blocks execution until the *fluent* *navigation-goal-reached* turns true. A fluent is similar to fluents in the fluent calculus [Thielscher, 1998] with the difference that it is not a logical statement or condition but a variable, or, more

specifically, a variable proxy object. Fluents can be combined by arithmetic operators such as +, −, ∗ or /. The result of such arithmetic expressions on fluents are so-called *fluent networks*.

The basic execution unit of a CRAM plan is a *task*. A task is a piece of code that can be executed. Plans combine tasks using CRAM Plan Language forms and constrain their order of execution. Each task contains a sequence of Common Lisp forms that are executed when the task is executed. The CRAM Plan Language provides forms for executing tasks in parallel as well as sequentially with the support for parallel ordering constraints. In the example above, the (perform ...) statement creates three tasks, one for each body form. All three tasks are executed in parallel in the lexical context of their creation. Tasks can be explicitly named using the (:tag name &body code) special form which binds an instance of the internal task object to a variable that is lexically bound in the complete plan. In addition, one parent task is created by the def–goal macro itself and the function update–navigation–command might create a number of internal tasks.

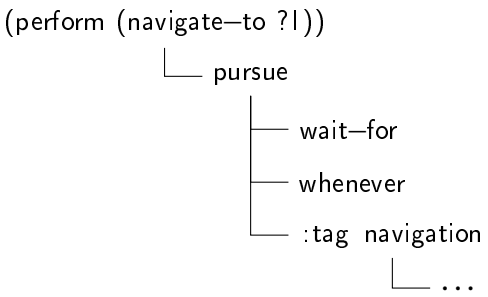


Figure 2.2: *The task tree if the simple navigation plan shown in Listing 2.2.*

The tasks in our example form a *hierarchy* as shown in Figure 2.2, i.e. the three tasks created by pursue are direct *sub-tasks* of the task generated by def–goal. Every task but the top-level task, which is created when the user executes a complete top-level plan, have exactly one parent task and 0 to *n* sub-tasks. The tasks that are important for reasoning, i.e. that contain semantic information, are explicitly referenced in a tree, the *task tree* which is defined by this parent-child relationship. Each task in the task tree can be referenced with a *path*. For instance, the path to the task named navigation in the example above is:

```
((TAGGED NAVIGATION) (GOAL (PERFORM (NAVIGATE-TO ?LOCATION))))
```

For accessing task objects, for instance to specify explicit partial ordering constraints or to synchronize on status changes of the respective tasks, the CRAM Plan Language provides two methods

- (task path) returns the task object with the position in the task tree specified by the absolute path.
- (sub-task path) is similar to task with the difference that path is a relative path with respect to the path of the task the sub-task function is executed in.

The rest of this section provides more detailed semantics and implementation details of the forms and terms introduced above.

2.1.3 Fluents

Fluents are the main mechanism for reacting on changing external input such as sensor data. In common libraries such as GUI libraries or communication libraries, the most widely used mechanism for reacting on asynchronous input are callbacks. However, callbacks can be cumbersome to use and require complex synchronization mechanisms. In addition, they make it hard to reason about control flow. In contrast, fluents provide a convenient and strait forward way to implement reactivity.

2.1.3.1 Fluents in CRAM Programs

As already mentioned, fluents provide a convenient way of adding reactivity to a CRAM program. In essence, fluents are proxy-objects that provide a notification mechanism and contain a value. Fluents can be combined to fluent networks which are fluents themselves. To create a fluent, CRAM provides the method make-fluent. The following example shows how to create a simple fluent:

```
1 (defparameter *fluent* (make-fluent :value 1))
```

This binds the variable *fluent* to a newly created fluent object that is initialized with the value 1. The user can read and set the value with the method value, for instance:

```
1 (value *fluent*)              ;; -> result is 1
2 (setf (value *fluent*) 2)     ;; sets the fluent to 2
3 (value *fluent*)              ;; the value is now 2
```

Normally, a CRAM program communicates with a robot through a middleware that, in the common case, executes an asynchronous callback whenever new sensor data is sent over the network. Fluent values are changed in these callbacks while the actual robot control program is executed in parallel. For instance, if the robot moves, i.e. its location changes, these changes can be received by the CRAM executive and a callback is executed. In this callback, the fluent that keeps the current position of the robot can be updated.

The robot control program can wait for a fluent to become non-nil by using the function wait-for:

```
1 (wait-for *fluent*)       ;; Returns immediately if the fluent
                            ;; value is non-NIL, blocks otherwise
```

As already mentioned, fluents can be combined to so-called fluent networks. In CRAM, the arithmetic functions +, −, * and / and the comparison functions <, >, eq, eql and not are overloaded to return a fluent network if at least one of their parameters is a fluent. For instance, to wait for the value of a fluent to become greater than 5, the following code can be used:

```
1 (wait-for (> *fluent* 5))
```

Please note that the expression (> *fluent* 5) returns a fluent network, i.e. the return value of that expression is always non-NIL while the actual value of that fluent network is either NIL or T. For instance, the following when expression will always be executed, no matter if the fluent's value is T or NIL:

```
1 (when (> *fluent* 5)
2   ...)
```

To provide more flexibility beyond the pre-defined fluent-network operators mentioned above, the system also provides the function fl-funcall that behaves like Common Lisp's funcall function with the difference that it returns a fluent network whose value is the result of a call to the function passed to fl-funcall . Apart from a function object, fl-funcall takes an arbitrary number of fluent and non-fluent parameters and the value is re-calculated by calling the function whenever one of the passed fluents changes its value. For instance, to wait for the square of a fluent value to become greater than 9, we can use the following code:

```
1 (wait-for (> (fl-funcall #'expt *fluent* 2) 9))
```

To execute a block of code whenever a fluent value turns true CRAM provides the macro whenever. Normally, whenever is used in code blocks that are executed in parallel to an actual control program. Internally, it is implemented as an endless loop that first does a wait-for on a fluent, then executes its code body and continues with waiting for the fluent again.

Finally, to just wait for a fluent to change its value, the special fluent network generator pulsed can be used. It does not contain a meaningful value but triggers wait-for only once for each change of the value of the fluent it is monitoring. Pulse fluents are special because their value is not referential transparent, i.e. it changes as soon as it is read. Particularly, this implies that pulse fluents should only be instantiated

temporary, i.e. they should not be bound to variables or passed around in function calls or return values.

In particular when used together with whenever, it can happen that a fluent changes its value, i.e. is pulsed, while the body of the whenever form is still running. Whenever a fluent is updated, a pulse is generated for the updated fluent as well as for all dependent fluents, i.e. all fluent networks derived from it. In the case of fluents constructed with pulsed, we consider three different cases of handling missed pulses, i.e. pulses that happened outside a whenever or wait–for form:

- Missed pulses are handled exactly once, no matter how many pulses occurred.
- Missed pulses are always ignored, i.e. they are never handled. wait–for returns only after the next pulse.
- All missed pulses are handled, i.e. wait-for returns exactly as often as the fluent value changed.

Let us consider a simple example: the user wants to handle events that are received at a high frequency, e.g. produced by a tracking system or continuous position updates of the robot's localization algorithms. If input data is received at a fixed frequency but cannot be processed fast enough, the user will either want to handle missed pulses exactly once or ignore missed pulses completely without queuing the input data. In this case, the program would bind the most recently received event to a fluent. If missed pulses are ignored, new events will be processed as soon as possible but already received data will completely be ignored. If missed pulses are handled once, the program will process as many input messages as possible, always using the most recent message but not waiting for a new event to be received. If input data is not received at a constant rate or if it can be processed fast enough in most but not all cases, it might make sense to use an input queue and handle all missed pulses since wait-for will return exactly as often as input data has been received.

2.1.3.2 Implementation Details

In this section we explain the implementation details of fluents and the method wait–for and the macro whenever.

Fluents are the most low-level data structure of the CRAM Plan Language. They are used for implementing higher-level language elements, in libraries and for synchronization. The implementation is based on classical multithreading synchronization mechanisms, i.e. mutexes [Dijkstra, 1965] and condition variables [Hoare, 1974].

The fluent interface. The data structure for representing a fluent is the base class fluent. Its definition is shown in Class 2. As can be seen, each fluent can be named.

Class 2 The definition of the class fluent, the base class for all fluents in CRAM.

class FLUENT

slot NAME:	The name of the fluent. If not specified, it a unique symbol is generated.
slot ON-UPDATE:	Hash table of callback functions that are executed whenever the fluent changes its value.
slot PULSE-COUNT:	Internal counter to check if pulses happened.
slot CHANGED-CONDITION:	Condition variable that is notified whenever the fluent changes its value.
slot VALUE-LOCK:	Mutex to lock the fluent when its value is changed.

In addition, it contains a slot that keeps a list of callback functions for observing and propagating updates along a fluent network graph. The slots changed–condition and value–lock are used for synchronizing slot access and to wait for pulses. The pulse count indicates how often the fluent has been pulsed. As already mentioned, the above listing shows the base class for all fluents, i.e. it does not contain any data structures for storing actual data. The reason is that fluents that are storing data are special and not all fluents really store data. For instance, CRAM implements two flavors of fluent networks, caching networks which have a data slot and non-caching fluents that compute their value on demand. The interface for fluents provided to the user is defined as a number of generic functions:

- The method value that returns the current value of the fluent. It can be computed on demand or just return the value of a slot. The method is required to be thread safe and is declared as follows:

```
(defgeneric value (fluent))
```

- The method pulse to notify all fluent-networks and threads that are waiting for the fluent. It is declared as follows:

```
(defgeneric pulse (fluent))
```

  Changes of the fluent's value always pulse the fluent.

- The method wait–for that blocks the current thread until the value of the fluent becomes non-NIL. It is declared as follows:

```
(defgeneric wait-for (fluent &key timeout))
```

  The optional timeout parameter causes wait–for to always return after either the timeout expired or the fluent's value turns true. On timeout expiration, wait–for will return NIL, otherwise it will return T.

- The macro whenever defines an endless loop whose body is executed whenever the value of the passed fluent is non-NIL. To improve readability, we only show a simplified version of its implementation that however still keeps the most important properties of the macro:

```
(defmacro whenever ((condition-fluent) &body body)
  `(let ((fluent ,condition-fluent))
     (loop do
       (wait-for fluent)
       ,@body))))))
```

  As can be seen, the macro executes its body in an endless loop whenever wait–for returns. That means whenever never returns unless the user either explicitly calls return or evaporates the thread in which whenever has been executed, for instance by running it in a pursue form.

In addition, a number of methods for registering, removing and accessing callbacks that are executed on value updates are defined. They are defined only for internal use though.

The implementation of wait–for. The wait–for method blocks until a fluent's value becomes non-NIL. That means the implementation needs to block the current thread until the value of the fluent changes, check if the value is non-NIL and return to waiting if the value is still NIL. As can be seen in Class 2, each fluent has a condition variable and a lock object, i.e. a mutex. When wait–for is called, the fluent blocks on the condition variable and waits for it to be notified. As soon as the corresponding wait method for the condition variable (in our case sb–thread:condition–wait) returns, the value of the fluent is checked and wait–for returns if it is non-NIL.

The pulse method. The method pulse calls sb–thread:condition–broadcast on the condition variable to notify waiting threads for a change. In addition, it calls all callback functions that are set in the slot on–update. Update callbacks are function objects that take exactly one parameter, the current value of the fluent. For instance, if the value of a fluent is changed, the pulse method is executed and all update callbacks are executed with the new value of the fluent.

Value fluents. Derived from the base class fluent, we define the most basic implementation of a fluent, a fluent that is just a proxy object for an arbitrary lisp object. Its definition is rather strait forward and adds only two slots to the parent class, a value and the slot test which can be used to control how to detect value changes. That allows to only signal a pulse if a new value assigned to the fluent is really different from its previous value. The definition of the class is shown in Class 3. Please note

Class 3 The definition of the class value–fluent.

class VALUE-FLUENT	
slot VALUE:	The value of the fluent.
slot TEST:	Comparison function to test for value changes. The fluent is only pulsed when a call to this function with the new and the old value returns NIL.

that this is a simplified version of the actual source code. For readability, the two mixin classes fl–cachable–value–mixin and fl–printable–mixin are left out. These mixins are just

markers for implementing generic readers and printers and to indicate that the fluent's value must not change without a notifying pulse. This is in particular important for the implementation of fluent networks which can only cache their values if their only dependencies are cachable fluents.

Value fluents are the only fluents that can be directly constructed by the user. All other kinds of fluents are derived from value fluents, e.g. pulse fluents or fluent networks. For constructing a value fluent, the system defines the function make–fluent that has the following (simplified) signature:

```
(make-fluent &key name value)
```

In addition, it provides a few additional keyword parameters to select the underlying fluent class to be used and to control how the fluent should be recorded when recording execution traces (see Section 3.2.1). Recording of fluents can be disabled completely or a maximal tracing frequency can be specified.

Fluent networks. Fluent networks are fluents with their value being computed based on the values of other fluents. A fluent network's value changes whenever a parent fluent changes its value. The implementation in based on callback functions that are executed whenever a fluent value changes, i.e. the fluent is pulsed.

Class 4 Class of fluent networks that directly depend on fluents that are not cachable.

class FL-NET-FLUENT (**superclasses:** fluent)
slot CALCULATE-VALUE-FUN: The function object to calculate the value of the fluent.

We discriminate two different kinds of fluents, fluents with values that can be cached (i.e. cachable fluents) and fluents that cannot be cached. In the current implementation, only pulse fluents are not cachable. The corresponding classes for fluent networks are defined in Class 4 and Class 5.

Class 5 Class of fluent networks that only depend on cachable fluents and hece are cachable themselves.

class FL-NET-CACHABLE-FLUENT (**superclasses:** fluent fl-cacheable-value-mixin)

slot CALCULATE-VALUE-FUN:	The function object to calculate the value of the fluent.
slot VALUE:	The cached value of this fluent network.

Both classes contain a slot that is bound to a function object for re-calculating the fluent network's value. This function object is executed whenever a fluent the network depends on changes its value though a callback in its on—update slot.

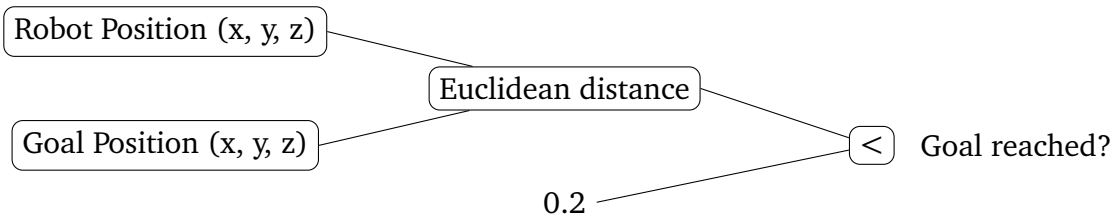


Figure 2.3: *Simple example fluent network for a fluent that is true when the robot is closer than 20cm to its goal.*

Figure 2.3 shows a simple fluent network for checking if the robot reached a navigation goal. The corresponding CRAM code for it is shown in the following listing:

```
1   (< (fl—funcall #'euclidean—distance *robot—pose* *goal—pose*)
2      0.2)
```

The goal position and the robot's current position are both fluents. They are combined by using the special function fl—funcall that returns a new fluent network. The comparison function < is overloaded in CRAM to return a fluent network if one of its parameters is a fluent.

Whenever a fluent operator (see Table 2.1 for a complete list of fluent operators) is used, a new fluent network is created that calculates its value according to the semantics of the operator. Values are re-calculated whenever the network fluent is pulsed. More specifically, fluent network generation first iterates over all parameters to find

<, >, =, eq, eql, member, +, -, *, /, not	Overloaded versions of the corresponding common lisp functions for creating fluent networks without changing the semantics.
fl-and	Fluent network that holds T if all dependencies are non-NIL. In contrast to Common Lisp's and, fl—and cannot be implemented as a macro and does not evaluate to the last parameter.
fl-or	Fluent networks that holds T if any of its dependencies is non-NIL. In contrast to Common Lisp's or, it is not a macro and evaluates all forms.
fl-funcall	Returns a fluent network that calculates its value by calling an arbitrary function with the values of all dependencies.
fl-apply	Same as fl—funcall but with the same signature and s semantics as apply.

Table 2.1: *Functions for creating fluent networks.*

dependency fluents. If no fluents are found, the creation function immediately calculates its value and returns. If at least one fluent is found, it registers a new callback function in the on—update table that pulses the fluent network. This leads to immediate re-calculation of the network's value and notification of all wait—for functions that are blocking on the fluent network.

Pulse fluents. Pulse fluents are special fluents that change their value to true when their dependency fluent has been pulsed. They are special because they change their value on reads, depending on the selected missed pulse handling. For instance, when handling missed pulses only once, the value of the pulse fluent might be T at the first read, but NIL at all subsequent reads until the monitored fluent is pulsed again. While this might not be intuitive behavior it is the only way to implement handling of pulses that occur outside of a wait—for call. The signature of the fluent operator to create a pulse fluent is:

```
(pulsed fluent &key (handle-missed-pulses :once))
```

The parameter handle–missed–pulses can be:

:once. For one or more pulses of the dependency fluent, the pulse fluent's value becomes true only once. If the dependency fluent has been pulsed before the pulse fluent is created, the value is true initially.

:always. The pulse fluent's value is true exactly as often as the dependency fluent has been pulsed.

:never. Similar to once with the difference that the pulse fluent's value is NIL if the dependency fluent has been pulsed before the pulse fluent's creation.

The implementation uses a reference pulse count set to the pulse count of the dependency fluent, and a processed pulse count. The class definition of the pulse fluent is shown in Class 6.

Class 6 Definition of the class pulse–fluent, a special class for handling pulses of fluents.

class PULSE-FLUENT (**superclasses:** fluent)

slot HANDLE-MISSED-PULSES: Identifier that specifies how to handle missed pulses. Must be set to either :once, :always or :never.

slot PROCESSED-PULSE-COUNT: Pulses that were handled already. To decide if a call to wait–for should return immediately, this value is compared to the slot reference–pulse–count.

slot REFERENCE-PULSE-COUNT: Pulse count of the dependency fluent.

The slot reference–pulse–count is always set to the pulse count of the dependency fluent and is updated by the on–update callback. Depending on the handling of missed pulses the slot processed–pulse–count is initialized either by setting it to 0 (for :always and :once) or to reference–pulse–count in case of :never. The calculation of the pulse fluent's value is done whenever the method value is called. The value is true if processed–pulse–count is smaller than reference–pulse–count. In the case of :always, processed–pulse–count is incremented by one with each call to value. For :never and :once it is set to reference–pulse–count which leads to a value of NIL in subsequent invocations of the value method.

2.1.4 CRAM Plan Language Expressions

The CRAM Plan Language is a domain specific language that extends Common Lisp with special forms for parallel and sequential execution with varying error handling semantics and for synchronizing, evaporating and suspending threads of execution. One specific feature is that error handling works even across thread boundaries, i.e. an exception can thrown in one thread and be transferred to another thread to be handled there. All features of the CRAM Plan Language are implemented in ANSI Common Lisp using with the addition of a few SBCL specific extensions for multithreading and synchronization. Lisp's powerful macro system is used to compile CRAM code directly to Common Lisp which is then compiled to native machine code by the SBCL Common Lisp compiler.

This section will discuss the different language forms provided by the CRAM Plan Language and discuss their properties. Finally, it will discuss the underlying implementation.

2.1.4.1 Sequential execution

Sequential execution is the standard evaluation mechanism in most imperative, functional, object oriented and logic programming languages, including Common Lisp. The CRAM plan language extends Common Lisp's sequential evaluation with additional exception handling semantics. All forms for sequential evaluation provided by CRAM' are discussed in this section.

seq. The form `seq` is identical to Common Lisp's `progn`. It executes each form sequentially and returns the result of the last form. If an error occurs, execution does not continue to the next form but a corresponding error handler is invoked. The reason for providing a wrapper instead of using Common Lisp's `progn` are mostly historical. In RPL, the predecessor of the CRAM Plan Language, the form was call `seq` and the CRAM Plan Language tries to reimplement RPL as close as possible. The following example just executes `form-1` and `form-2` in order and returns the result of `form-2`:

```
1 (seq
2   (form-1)
3   (form-2))
```

try-in-order. Some tasks are achieved by trying different algorithms sequentially until one succeeds. The from try-in-order executes the first form and returns its result if it does not throw an exception. If an error occurs, it continues with executing the second form until all forms are tried or one succeeds without throwing an exception. If all forms fail, seq throws an exception of type composite-failure that contains all failure objects of the failed forms. The following example executes form-1 and returns its result if form-1 is not failing. Otherwise, it continues with executing form-2:

```
1 (try-in-order
2   (form-1)
3   (form-2))
```

try-each-in-order. The form try-each-in-order is similar to try-in-order with the difference that it iterates over a list and binds a variable to each element of that list until no error is thrown. In other words, instead of trying each sub-form one by one until one succeeds, it executes the same body for different bindings of a variable until one succeeds. The following code snipped illustrates how to search for objects at different locations until it is found:

```
1 (try-each-in-order (object-location '(:table :counter :cupboard)
    )
2   (or (find-object-at object-location)
3       (fail)))
```

The return value of try-each-in-order is the result of the function body if no failure has occurred. If the body fails for all iterations, an instance of composite-failure that contains all failure objects is thrown. Please note that try-each-in-order is new in the CRAM Plan Language and was not provided by RPL.

2.1.4.2 Parallel execution

In addition to sequential execution of forms, the CRAM Plan Language provides special support for parallel execution of expressions. Basically, parallel CRAM programs are translated into programs that use SBCL's multithreading library *sb-thread*. Lisp programs are basically nested S-Expressions which form a syntax-tree where each expression has exactly one parent expression and every expression can have multiple child expressions. This hierarchy allows for error handling across thread boundaries. If a child form throws an error, the exception object is rethrown by the parent thread object. If an error is rethrown, all child threads that are direct siblings of the parent object are evaporated, i.e. killed. One important property of all CRAM Language forms that execute their child expressions in parallel is that they must not terminate unless all of their child threads terminated. This also includes cases where forms are evaporated. Each form must ensure that if it is evaporated, it also evaporates all its children. Otherwise, evaporation could lead to lots of parent-less threads that just consume resources.

par. The CRAM form par allows to execute all forms in parallel. It fails if any child form fails, i.e. throws an exception, and rethrows the respective condition object in case of such a failure. par terminates when all child forms finished executing. Essentially, this corresponds to a fork/join pattern for parallelization. par spawns threads and joins

all of them to wait for their termination. The following example executes the two functions function−1 and function−2 in parallel and waits for both to finish:

```
1 (par
2   (function-1)
3   (function-2))
```

par returns the return value of the last form in the par block, i.e. its behavior with respect to return values is similar to seq.

pursue. In contrast to classical parallel execution and synchronization on the termination of all threads, the pursue form terminates as soon as one of the child forms terminates. The other child forms are evaporated. pursue has similar error handling semantics as par, i.e. it fails if one of the child forms fails and rethrows the condition object. The most prominent use case for pursue is a monitor pattern, i.e. start two threads where one monitors some state and finishes if a goal state has been reached while the other thread executes commands to achieve the goal state. The following example shows a simple navigation routine that executes and re-calculates commands to reach a goal until a goal condition (encoded in a fluent) has been reached:

```
1 (pursue
2   (wait-for *goal-reached-fluent*)
3   (loop do
4     (let ((command (recalculate-navigation-command)))
5       (send-navigation-command command)
6       (sleep 0.02))))  ;; To send commands at 50 Hz
```

If pursue succeeds, it returns the result of the first form that terminated, i.e. the form that caused the pursue block to finish.

try−all. The try−all form executes all sub-forms in parallel but does not rethrow errors unless all child forms failed. With respect to termination it is similar to pursue. try−all terminates as soon as one of its child forms succeeds. One use case is the

execution of different perception algorithms. As soon as one was able to detect an object, try–all should terminate while an error should be thrown if all algorithms were unable to detect the object. In that case, similar to try–in–order, a condition of type composite–failure is thrown. The following example illustrates the use of try–all:

```
1 (try-all
2   (detect-object-using-color)
3   (detect-object-using-depth))
```

The return value of try–all is the result of the first child form that finished.

2.1.4.3 Error handling

The error handling mechanism of the CRAM Plan Language is based on Common Lisp's condition system. It allows for exceptions across thread boundaries since CRAM Plan Language forms provide a clear parent-child-relationship through the hierarchy formed by s-expressions.

Failures are a central concept in CRAM programs and are supposed to be used for signaling problems such as *"cannot reach object"* or *"cannot see object"*. All CRAM failures are derived from the base condition class plan–failure and the function fail is used to signal a plan failure.

In Common Lisp, error conditions can be handled in three ways:

- Perform a non-local exit. In that case program execution continues at the exit point. This is similar to try-catch-blocks in languages such as Java and C++ where program execution also continues after the catch block if a handled exception is not rethrown.
- Invoke a restart. This executes a function that also performs a non-local exit. However, restarts are normally provided by the code that is throwing an error and are an elegant way to provide a structured API for recovering from failures.

- None of the above. In that case, the Lisp debugger is invoked and the user can investigate the stack and manually invoke a restart.

To deal with the highly multithreaded nature of CRAM programs, the error handling mechanism provided by the CRAM Plan Language is slightly different from Common Lisp although it is implemented based on Common Lisp's condition system.

For providing failure handling code, the CRAM Plan Language provides the macro with-failure-handling. Its signature is defined as follows:

```
(with-failure-handling (&rest handler-clauses)
  &body body-forms)
```

Then specification of handler-clauses within with-failure-handling is similar to Common Lisp's labels or flet macros. First, it expects a list of failure handlers. Each handler form consists of the type, then the name of a variable the condition object is bound to in the lexical extent of the failure handling body and then the body forms. The signature of handler clauses is as follows:

```
(failure-type (variable-name) &body failure-handling-body)
```

The body forms of failure handlers are executed in an anonymous block, i.e. return can be used to perform a non-local exit from the with-failure-handling block. Retries are a very common recovery strategy in CRAM programs. with-failure-handling thus provides the (local) function retry that is bound in all failure handlers and the actual body forms. retry restarts the execution of the body forms. If neither retry nor return is called in failure handlers, the condition object is rethrown and propagates up the call stack to be handled by the next with-failure-handling form. If none is found, the debugger is invoked.

Due to the highly multithreaded nature of CRAM programs, Common Lisp restarts are not supported currently. Although restarts would work inside a single thread, it is harder to implement them in order to work across thread boundaries and the implementation is left to future work.

When the fail function is evaluated, first all thread-local error handlers are executed. If none of them performs a non-local exit, the error is rethrown in the parent thread and can be handled there. Rethrowing in another thread causes the original signaling thread to terminate and the stack that contains information about where the error was thrown is lost. As already mentioned, failures in CRAM are the preferred way to signal different kinds of execution errors. However, they are different from Common Lisp errors which represent runtime errors, failed assertions, type errors etc. While for CRAM failures, the debugger is only invoked when they have not been handled by any with–failure–handling form, Common Lisp errors invoke the debugger if they reach the boundaries of a thread, in addition to being rethrown. This allows for examining the stack and the local variables of the stack frame where the error has been generated.

2.1.4.4 Tagging, partial ordering and syncrhonization and suspension

One special feature of the CRAM Plan Language is that tasks are first-class objects and can be bound to variables and interacted with. This allows for performing actions such as suspending a task or waiting for a task to finish as well as partially ordering tasks. This section explains tagging of tasks and interactions with task objects. In particular task suspension requires special handling to deal with code blocks that must no be suspended (e.g. safety critical code such as low-level control or monitoring code) or that require code to be executed right before a task is suspended (e.g. stopping a motor).

with–tags. The with–tags form establishes a lexical context in which task objects generated from specially marked code blocks are bound variables. The following code snippet illustrates tagging of tasks and task suspension.

```
1 (with-tags
2   (pursue
3     (whenever (*collision*)
4       (with-task-suspended (navigation)
5         (recover-from-collision)))
6     (:tag navigation
7       (navigate-to-goal))))
```

The with-tags forms uses a code-parser to find tags, i.e. blocks that are started with the symbol :tag. The symbol following :tag specifies the name of the variable that is bound to the task object constructed from all following forms in the :tag block. This variable can then be used by forms such as with-task-suspended or task methods such as status to get the current status fluent of a task. The internals of tasks will be explained in the following section.

with-task-suspended. The form with-task-suspended executes a code body with a specific task and all of its sub-tasks suspended, i.e. sleeping. When with-task-suspended's body is finished, the task that was suspended continues with its execution at the point where it has been suspended unless it uses on-suspension or retry-after-suspension.

without-scheduling. To suppress any CRAM scheduling mechanisms at all, critical code can be executed in a without-scheduling block. It prevents a task from being evaporated or suspended and should be used with care. Please note that Common Lisp interrupts can still occur. If the user needs to disable them, SBCL's sb-sys:without-interrupts macro has to be used.

on-suspension. To execute code right before a task is suspended, for instance to stop a motor, the form on-suspension can be used. The following code shows how it is intended to be used:

```
1 (on-suspension (set-velocity 0.0)
2   (loop do (set-velocity 1.0)))
```

Whenever a suspension occurs, the on-suspension form is executed to stop the robot's motors. As soon as the task is woken up again, the program continues with the loop and returns to sending repeated velocity commands.

retry-after-suspension. When suspended, in particular low-level code often needs to unwind, i.e. terminate cleanly. The form retry-after-suspension causes its body to be completely unwound right before the task is suspended. All unwind-protect forms are executed. When woken up again, the complete code body is re-executed from the beginning.

partial-order. Partial ordering of tasks can lead to great performance improvements compared to purely sequential actions. For explicitly specifying ordering constraints on parallel tasks, the CRAM Plan Language provides the macro partial-order. The signature of partial-order is as follows:

```
(with-partial-order (&body body-forms)
  &rest ordering-constraints)
```

All forms are executed in parallel unless explicitly ordered using ordering constraints. An ordering constraint has the form:

```
(:order constraining-task constrained-task)
```

An ordering constraint will prevent the constraint task from being executed until the

constraining task terminates. The tasks used in ordering constraints must be task objects. The following example illustrates how the macro is used:

```
1 (with-tags
2   (partial-order
3       ((:tag task-1
4          (plan-1))
5        (:tag task-2
6          (plan-2))
7        (:tag task-3
8          (plan-3)))
9     (:order task-2 task-1)))
```

In the example, three tasks are executed. The ordering constraint causes task-1 to be executed after task-2 terminated. task-3 is unconstrained and is executed in parallel to task-1 and task-2.

Tagged tasks are one way of getting the task object of a specific piece of code. However, tags are only visible in the lexical context of the with-tags macro. If partial ordering constraints should be specified for sub-tasks that are hidden in a different stack frame, e.g. inside a function that is called from the partial-order form, the function task to get the task object from an absolute path in the task tree and the function sub-task for task objects from relative paths are provided by the system. For instance, suppose we do not want to execute the task task-3 before the (hidden) task navigate of the function plan-2 finishes. The path to this task would be:

```
((tagged task-2) (plan-2) (tagged navigation))
```

Using the function sub-task, the corresponding ordering constraint is shown in the following code snippet:

```
1 (:order
2     (sub-task '((tagged navigation) (plan-2) (tagged task-2)))
3     task-3)
```

Please note that paths in CRAM are specified in inverse order, i.e. the first element in a path specifies the name of the sub-task to be referenced and the next element specifies its direct parent.

2.1.4.5 Definition and Execution of CRAM Programs

While it is possible to call normal lisp functions in CRAM programs, they would not appear in the task tree, i.e. they cannot be used to synchronize on or for reasoning about plan execution. In order to generate a task tree node and a corresponding task path entry, in CRAM, functions should be defined using the def-cram-function macro. Its signature is similar to Common Lisp's defun macro for defining functions. The following code demonstrates its use:

```
1 (def-cram-function navigate (goal)
2   (pursue
3     (wait-for (goal-reached goal))
4     (loop repeat (send-navigation-command
5                   (calculate-navigation-command goal)))))
```

If the user needs more flexibility, for instance if she is defining new macros that create code which should appear on the task tree, CRAM provides the macro with-task-tree-node . The macro is rather complex and allows to generate arbitrary lambda functions to be executed and added as nodes on the task tree. It will be discussed in Section 2.1.4.6.

All CRAM language extensions require a minimal environment to be set up in order to execute. This environment includes variables that contain the current task, the path to the current task object and the corresponding node in the task tree. For executing

CRAM forms in the Lisp REPL, the system provides the macro top–level. All CRAM forms, e.g. pursue or par must be executed in the dynamic extent of such a top–level form. The following example illustrates its use:

```
1 (top-level
2  (par
3   (branch-1)
4   (branch-2)))
```

While top–level is convenient at development time, in actual robot control programs an entry point that is an actual function is required. To define top-level cram functions, the system provides the macro def–top–level–cram–function:

```
(def-top-level-cram-function prepare-breakfast (persons)
  ...)
```

The semantics are similar to top–level with the difference that the task-tree created during execution is bound to that function, i.e. it will not be overwritten until the function is re-executed. Using named top-level functions provides a much more structured way to execute CRAM code than top–level does and top–level should only be used when interactively developing an application.

2.1.4.6 Implementation details

The Common Lisp macro system is extremely powerful because, in contrast to for instance C++, Lisp macros are not based on pure text replacement. Instead they are similar to functions with the difference that they are executed at compile time and transform Lisp code. Macros enable a programmer to not only extend a programming environment by adding new functions, methods and classes but also to add new special forms that require fine control of the evaluation of code. To implement forms such

as par or pursue that execute code in parallel, operating system threads need to be created from sub-forms before evaluating them. Hence, we need to implement CRAM's higher-level special forms as macros.

Algorithm 1 sketches the code the pursue macro is expanded to. Please note that that the code of the sub-forms are passed to the macro as a nested list bound to a parameter. The code in Algorithm 1 only shows the result that is executed at execution time. We use it as an example to show the underlying data structures and how evaporation, suspension and, more generally, communication between different tasks is implemented. As can be seen, when executing pursue, first a set of task objects is

Algorithm 1 The code generated by the pursue macro. It executes all forms in parallel and terminates as soon as one of the sub-forms terminate.

```
Ts ← ∅
for form in sub-forms do
  T ← create task for form
  Ts ← Ts ∪ T
end for
while no task in Ts is in a terminal state do
  wait for change in the status in any task in Ts
end while
for all T in Ts do
  if T is in a terminal state then
    S ← status of T
    R ← result of T
  else
    Evaporate T
  end if
end for
if S is :succeeded then
  return R
else
  throw R
end if
```

created. For each sub-form of pursue, one task object is created and each task starts executing immediately. Then pursue starts to monitor the states of all tasks until one enters a terminal state. Finally, all tasks that did not terminate yet are evaporated. If the terminated task failed, the failure is rethrown, otherwise, the result of the terminated task is returned.

Task objects. The underlying data structure for implementing all concurrent CRAM macros is the *task* object. Tasks are implemented on top of SBCL's *sb-thread* library and provide the functionality for dealing with failures and return values and for implementing evaporation and suspension. Class 7 shows the definition of the class abstract—task which is the base class of all task objects in CRAM.

Class 7 The class ABSTRACT-TASK. It is the base class for all task objects.

class ABSTRACT-TASK

slot NAME:	The name of the task.
slot THREAD:	The thread object of the task or NIL if the task is not running yet.
slot PARENT-TASK:	The task object of the parent task.
slot CHILD-TASKS:	The list of child tasks.
slot STATUS:	The status fluent of the task. The status must be one of the following values: :created, :running, :suspended, :succeeded, :failed or :evaporated.
slot RESULT:	List of result values of the task or the condition object if it failed.
slot THREAD-FUN:	The function object of the task or null if it has not been bound yet.
slot PATH:	The path of the task.
slot CODE:	The s-expression of the task's function.
slot CONSTRAINTS:	List of constraint fluents. The task is not allowed to start before all values are non-NIL.
slot MESSAGE-QUEUE:	The mailbox of the task. It is used for communication between the parent task and its children.

A task object can be instantiated with or without a function object bound to the slot *thread-fun* and the constructor takes the additional argument :run—thread that indicates if a thread should be started at construction time (default), given a function object has been specified. The execute method is used to start executing a task object. If run—thread was set to false when constructing a task, the execute method has to be executed explicitly. Multiple execution of a single task object is not supported by CRAM and trying to execute a task more than once would result in a runtime error.

Task status and events. When a task is started up, it enters an event loop that allows other tasks to change the execution status of the task. Events are sent to a task

by adding them to the task's mailbox, essentially a queue for event objects. During execution, the task periodically enters the event loop.

Entering the event loop periodically in multithreaded systems that are based on *pthread* or similar systems is not completely trivial because there needs to be a way to "inject" code either whenever an event is sent and needs to be processed or at defined points. While the former solution is easier to implement by using SBCL's interrupt—thread method that allows to interrupt another thread and execute an arbitrary closure in it, it can cause major problems because a thread might be interrupted at a critical point, e.g. when it is performing motor control or communicating over the network. In particular task interruptions right after creation can lead to undefined state because an operating system thread might be interrupted while it is still initializing. Thus, in CRAM we use the latter approach. SBCL provides a timeout mechanism through deadlines for all calls that perform IO. Deadlines are established in the dynamic extent of a specific code block and when a task blocks on IO and the deadline expires, a condition is raised that can be handled by the task and invoke the event loop. Then, the deadline can be deferred and the deadline condition is raised again after the new deadline expires. The drawback of this approach is that only lisp functions that are performing IO are interrupted. However, the runtime of tasks that do not terminate quickly is most often constrained by IO.

The status slot of a task is a fluent to allow to use the fluent mechanism introduced before for synchronizing on the status of tasks and needs to be set to one of the following values.

- :created The task object has been created but was not executed yet.
- :running The task is running.
- :suspended The task has been suspended and is waiting for being woken up.
- :succeeded The task terminated successfully.
- :failed The task failed, i.e. an error has been thrown that propagated up the stack of the corresponding thread object without being handled.
- :evaporated The task has been evaporated, i.e. it was killed during execution. It does not have a meaningful result value in that state.

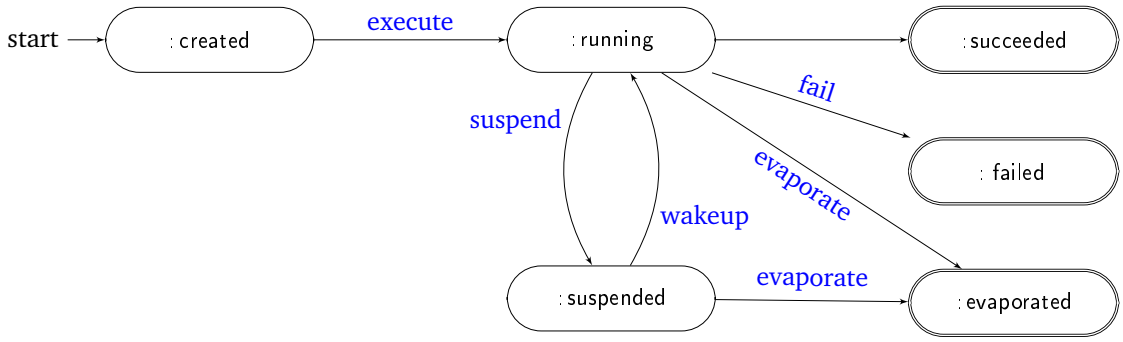


Figure 2.4: *Allowed state transitions during the life time of a task object.*

Status transitions are always performed in the task's event handler. Figure 2.4 shows the allowed state transitions during the life time of a task object and the corresponding events. The initial state is :created. When the execute method is invoked which happens directly at creation time, a new operating system thread that executes the function object bound to the *thread-fun* slot is created. In addition, the task is registered as a child task in the slot *child-tasks* of its parent object. As soon as the thread starts running, the task enters the :running state. There exist three terminal states, :succeeded that is entered if the thread terminates nominally, :failed when an unhandled failure is detected and :evaporated when the task is shut down externally while it is still running. In addition, the task can enter the :suspended state which indicates that it is not executing any code but just waiting to be woken up. While transitions to :failed and :evaporated are caused by explicit events, nominal termination of the thread function and the transition to :succeeded is not caused by an event.

Implementation of evaporation and suspension. Common implementations of multithreading libraries do not allow for preemption of threads without explicit support for it in client code. The reason is that if a thread is terminated, it is important that the termination happens at well defined exit points in code to prevent undefined behavior. For instance, when sending data over a socket, termination must not happen in the middle of the transmission of a package. Otherwise, communication might run out of sync.

As already mentioned, scheduling in the CRAM Plan Language is based on events that are periodically processed by entering the event loop whenever an IO deadline occurs. One important aspect here is that executing the event handler function is done

by Common Lisp's condition system. It does not cause a stack unwind but the handler is executed in the stack frame of the function that is blocking on IO. In case a task receives an evaporate event, the event handler performs a non-local exit by invoking the restart teardown that is established at the main function of the task. The stack is unwound, all protection forms defined with unwind—protect are executed and the task terminates with status :evaporated. In addition, the :evaporate event is propagated to all children of the task to evaporate them as well.

Handling of suspend events is a little more tricky because we need to explicitly unwind in case the currently executed code is inside a on—suspension or a retry—after—suspension form. Both forms use a special (dynamically scoped) variable to register an unwind-handler that executes unwind forms and performs a non-local exit. When a suspend event is received, the system first checks if an unwind needs to be performed and executes the corresponding unwind function. Then it blocks on the task's mailbox to wait for new events, e.g. a wakeup event. If a wakeup event is received, the event handler function terminates which causes the original code to continue.

The task tree. The task tree is a data structure that allows for accessing task objects based on a unique task path and for applying plan transformations, i.e. for replacing the code that is executed when the control flow reaches a specific point in plan execution. While the parent-child relationship between tasks and their sub-tasks already create an implicit tree of task objects, most of these tasks are unimportant for reasoning about plans. Plan transformations normally are not required to replace code at arbitrary locations but at well-defined points such as functions, tags or special macro forms. In particular plan transformation rules as presented in [Müller, 2008] require information such as the s-expression that was executed, the binding of parameters and the task path. Additionally, to support actual plan transformation, i.e. the replacement of parts of the code of a plan *at run-time*, a task tree node need to explicitly store code replacements besides the original code. Class 8 shows the definition of the task-tree-node class. As can be seen, task tree nodes reference objects of type code. They contain information such as the s-expression but also the function object that is to be executed when the corresponding task runs, the parameters to this function object (bound at run-time, when the task tree node is actually executed) and the corresponding task object. Class 9 shows the definition of the code class:

Class 8 The definition of the task-tree-node class. It contains information for reasoning about plans, for matching transformation rules and for actually transforming code.

class TASK-TREE-NODE

slot CODE:	An instance of the class code that contains the code of this tree node.
slot CODE-REPLACEMENTS:	A list of code replacements, normally set by transformation rules. Only the first code replacement is used but other code replacements might be kept for keeping a history of plan applied plan transformations.
slot PARENT:	The parent task-tree-node instance.
slot CHILDREN:	List of child nodes.
slot PATH:	The path of this task-tree-node object in the tree.

Class 9 The definition of the code class. It contains information about the code that is executed when the control flow reaches its corresponding task—tree—node object.

class CODE

slot SEXP:	The s-expression of the code.
slot FUNCTION:	The Lisp function object to be executed.
slot TASK:	The task object that corresponds to the task—tree—node of this code object.
slot PARAMETERS:	Parameters for the function slot. This slot is only bound at run-time when the corresponding task object is executed.

CRAM language forms such as with—task—suspended and partial—order require task objects for their operation. These task objects can either be bound to local variables in their lexical scope, e.g. by using with—tags or they can be acquired using the functions task that takes an absolute task path and sub—task that takes a relative task path. In particular partial—order might require to reference task-objects that are not created yet. However, task and sub—task will create task objects on demand if none could be found for a given path. If the control flow then reaches the corresponding task tree node, instead of creating a new task object, the already created object is used.

Currently, only a small sub-set of all CRAM forms creates task tree nodes: def—cram—function , def—top—level—cram—function and all tags. The reason is that in most cases, a finer

grained task tree would just complicate reasoning and plan transformations without any additional benefit.

Task paths are lists that contain elements naming the corresponding task tree nodes. For instance, a top-level CRAM function with name foo creates the entry (top–level foo) in the path. If it calls a CRAM function bar, the path to the function would be:

```
((BAR)(TOP–LEVEL FOO))
```

Please note that the path must be read from right to left, i.e. the path part of the root node of the task tree is always at the right most position. The reason is that prepending to a list can be done without copying the complete list in Common Lisp and other functional languages that represent lists by cons-cell like data structures (i.e. pairs of pointers).

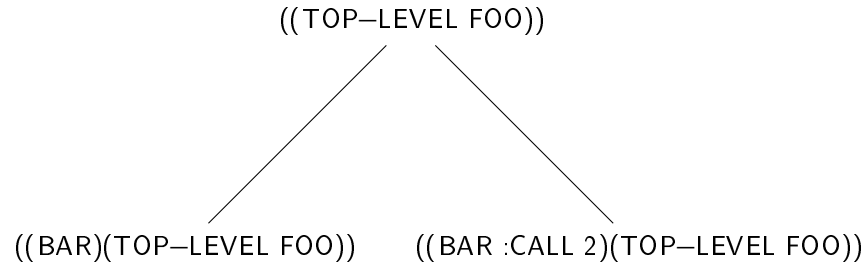


Figure 2.5: *Code tree of the* CRAM *program shown in Listing 2.3. The same function is executed twice. Each execution branch has a unique path.*

In one execution run, i.e. in one single execution of a top-level form, task tree nodes are never re-used. In other words, a task tree node can be executed only once. Otherwise, a single branch of execution could not be uniquely identified by the corresponding task path anymore. Instead of executing one task tree node multiple times, a new node is created for each iteration and an iteration specification is added in the task path. For instance, if the function bar is called twice from the top-level function foo, the second call will create a new task tree node with path

```
((BAR :CALL 2)(TOP–LEVEL FOO))
```

Figure 2.5 shows the complete task tree generated by the code in Listing 2.3.

Listing 2.3: *Simple cram plan that will generate two nodes for function* bar *in the task tree.*

```
1 (def-top-level-cram-function foo ()
2   (bar)
3   (bar))
4
5 (def-cram-function bar ()
6   (do-the-work))
```

In the current implementation, only def-cram-plan, def-top-level-cram-plan, tags and goals (as explained in Section 2.5.1) create nodes on the task tree. For user-defined macros that should be visible in the task tree, the system provides the macro with-task-tree-node . Its signature is shown in the following code snippet:

```
(defmacro with-task-tree-node ((&key path-part name
                                     sexp lambda-list
                                     parameters)
                               &body body)
  ...)
```

The path of a task tree node is generated prepending the current path with the path-part parameter. The name parameter is used for naming the actual task object and the sexp parameter allows to specify the s-expression of the code that is executed. If unspecified, the value of the parameter body is used. In particular when adding code replacements, it is necessary to provide a mechanism to pass variables in the lexical context of the code where the with-task-tree-node macro is used to the actual code. Otherwise, it would be impossible for replacements which are defined in a different lexical environment to access parameters they need. The parameter lambda-list allows to explicitly bind variables in the lexical context of body to parameters. When a with-task-tree-node block is executed, a new task object is created that executes body.

Internally, apply is used to call the task's function object with lambda–list on the values specified with the parameters argument.

2.2 Reasoning and the CRAM Prolog Interpreter

In contrast to, for instance, systems like KNOWROB, CRAM does not use an actual Prolog compiler such as SWI-Prolog [Wielemaker et al., 2012] but implements its own reasoning engine that is similar to Prolog.

CRAM's reasoning engine is intended to be used for every reasoning task that needs to be solved during plan execution, planning, plan parameterization but also in lower-level components, e.g. process modules. In particular reasoning about plan execution and designators requires to access Common Lisp data structures such as instances of classes and structures. In classical Prolog, data must be represented by atoms, numbers and predicates. Although it is possible to represent class instances by atoms and use predicates to bind slot values, this solution is inconvenient at best, not only because it makes introspection hard. Exporting Lisp data structures to a Prolog run-time is non-trivial. In contrast, when using a Prolog interpreter that is natively written in Common Lisp, Lisp data structures can directly be accessed.

Classical Prolog allows for an infinite number of solutions. Normally, queries are executed using a Prolog shell which stops, prints a solution and waits for further user input. The user then can request the next solution or terminate the query. This mechanism is unfeasible when integrating a Prolog reasoning engine in a program. Popular Prolog compilers such as SWI Prolog provide a foreign function interface that allows for issuing a query and request solutions subsequently. However, only one query per thread is possible at a time. In robot control programs, it is often necessary to try out a different solution for a query after executing one action to find alternatives if the action failed. For instance, when searching for objects, the Prolog engine can be used to infer search locations. If an object has not been found at the first location, the second solution should be tried. Generating all solutions is not always feasible, either because there exist an infinite number of solutions or calculating one solution is computationally very expensive. The reasoning engine used in CRAM uses *lazy lists* for representing all solutions to a query. The complete computational context of a query is "frozen" in a generator function that is used for generating each single element of

the lazy list. That way, a call to Prolog can return a data structure representing the set of all queries and the user can compute one solution after the other, on demand. The main difference to only allowing for one reasoning engine per thread is that in a single program, the result of many queries can be bound to variables and new solutions for any of them can be generated when required and it is even possible to compute one solution in one thread and the next solution in another thread.

2.2.0.7 Syntax

CRAM's reasoning engine is a Prolog interpreter completely implemented in Common Lisp. In contrast to classical Prolog engines, it uses Lisp syntax for Prolog programs and queries. The following code block shows the implementation of the predicate *member* in classical Prolog syntax:

Listing 2.4: *The definition of the member predicate in classical Prolog.*

```
1 member(X, [X|_]).
2 member(X, [_|Y]) :- member(X, Y).
```

Classical Prolog distinguishes between facts and rules. Line 1 in Listing 2.4 shows a fact definition that states that X is a member of a list when it is the first element of that list. Line 2 shows the definition of a rule that states that X is a member of a list if it is a member of the list without the first element.

Facts are essentially rules with no rule body, i.e. leaf nodes in the Prolog prove tree. However, due to Lisp's prefix notation, there is no syntactical difference between the two concepts in CRAM Prolog. The corresponding definition of the member predicate in CRAM's Prolog engine looks as follows:

Listing 2.5: *The definition of the* member *predicate in* CRAM*'s reasoning system.*

```
1 (def-fact-group member ()
2   (<- (member ?x (?x . ?_)))
3   (<- (member ?x (?_ . ?y))
4     (member ?x ?y)))
```

Apparently, the most important difference is Lisp's Polish prefix notation. Instead of putting the parenthesis after the name of a functor, it is specified as the first argument after the opening parenthesis. Since Lisp is not case sensitive, in contrast to classical Prolog, variables are indicated by a preceding question mark instead of writing them upper case. In contrast to classical Prolog, goals in a rule definition are combined by an implicit *and*. Additionally, instead of using a semicolon for or as in classical Prolog, CRAM's implementation uses the explicit keyword *or*. For instance, to state that either *A*, *B* or *C* can be true, we can write the following Prolog goal in CRAM:

```
1 (or A B C)
```

Fact groups are a mechanism for grouping predicates that form basic compilation units for interactive programming. As can be seen in the definition of the member predicate above, several variants of the same predicate co-exist in Prolog and they might even be defined recursively, i.e. reference each other. Recompiling a fact group removes all predicates previously defined in it and re-adds all predicates in the fact group. If a predicate is removed, recompilation of the fact group will also remove it from the Prolog prove environment. Fact groups are defined using the macro def-fact-group. Its signature and implementation details will be explained in the following section.

2.2.1 Implementation of the Prolog Interpreter

The core of each Prolog interpreter is the unify function. Unification of two patterns that contain variables means to find *bindings* for all variables so that both patterns are equal. For instance, unification of the patterns (1 a ?y) and (?x a b) will result in the following set of bindings:

$$\begin{aligned} \text{?x} &= 1 \\ \text{?y} &= b \end{aligned}$$

Essentially, unification is a generalization of pattern matching where both sides can contain variables. Unification decides if there exists a solution for assignments of the unbound variables in both patterns so that both patterns are equal and returns these bindings.

As mentioned already in the previous section, CRAM Prolog predicates are normally defined as facts and rules in a fact group. Let us re-consider the definition of the predicate member in Listing 2.5. Rules are defined with <−. The first expression after this definition sign is called the *rule head* and the following patterns are called the *rule body*. When a rule is defined, it is added to CRAM Prolog's database to be used in the inference process to prove goals.

CRAM's Prolog interpreter is accessed by the Lisp function prolog which has one mandatory parameter, the Prolog goal to prove and one optional parameter for a set of initial bindings:

```
1 (prolog <goal> [bindings])
```

For instance, to check if an element is a member of a list using Prolog, the previously defined member predicate can be used as a goal in a call to the prolog function:

```
1 (prolog '(member 1 (1 2 3 4)))
```

The function tries to prove the goal based on the predicates defined in the system and returns a list of binding sets if it can find solutions. If not, it returns NIL. The result of the example above will be (NIL) which indicates that the goal holds and but bindings were computed since there were no free variables in the goal.

To prove a goal, the CRAM Prolog interpreter essentially performs the following steps:

1. Find all rule heads that can be unified with the goal.
2. For each possible solution of unification, expand the rule body and prove each sub-goal one by one by unifying them with all rule heads and expanding them. Accumulate the bindings or backtrack if a sub-goal cannot be proven.
3. When all sub-goals have been proven, yield that specific solution as an element of the returned lazy list.

To understand the exact inference process, let us consider the goal (member 2 (1 2 3)) that just checks if *2* is a member of the list *(1 2 3)*. The first step is to find all rule heads that can be unified with it. If the Prolog database only contains the definition of Listing 2.5, only the second definition can be unified. ?x will be bound to 2 and the variable ?y to (2 3). The rule's body is a recursive evaluation of the member rule, this time with different bindings. Unification will this time hold for both rules definitions. The first one does not have a rule body and thus a solution was found. Since both rule definitions match, a choice point is generated, i.e. when a second solution is requested, the system continues with another recursive sub-goal expansion of member , this time with ?y bound to (3). Then, no solution cannot be found because none of the definitions of member match, i.e. we found all solutions for the goal.

2.2.2 Implementation of Predicates

As mentioned already, classical Prolog facts and rules in the CRAM Prolog system must be defined inside fact groups. Fact groups form the basic compilation unit for

predicates. The system keeps a database of all fact groups. When resolving a predicate, it searches through that database and tries to find rule heads that can be unified with the current Prolog goal. When a fact group is recompiled, it completely replaces its previous version. This is important because in Common Lisp, users often develop interactively and recompile certain expressions again and again. If the system would handle predicates separately, it could not decide if the user wants to add a new predicate or which one to replace a predicate is recompiled. Classical Prolog handles this by considering Prolog files as compilation unit. The basic syntax for the def–fact–group macro is as follows:

```
(def-fact-group name (&rest exported-facts)
  &body fact-definition)
```

name is a symbol naming the fact group. Only one fact group with a given name can be defined since the fact group as a compilation unit is identified by its name.exported–facts symbols specify predicates that can also be defined in other fact groups. Multiple predicates can be exported. The reason for enforcing explicit declaration of redefinable predicates is to prevent errors caused by predicates that are named equally by accident. Finally, the fact–definiton fields contain the definition of the actual facts. Facts definition have the following form:

```
(<- (predicate-name &rest arguments)
  &body body)
```

The rule head always needs to be a list with the name of the predicate to be defined as first element. The rest are optional arguments. All body forms must also be lists that are unified with rule heads during the Prolog inference process.

One important feature that is widely used in CRAM's Prolog interpreter is the easy integration with Common Lisp. Instead of implementing predicates based on facts and rules, the system also supports so-called Prolog handlers. A Prolog handler is a lisp function that is called by the Prolog engine to prove a certain predicate. Prolog handlers are defined with the macro def–prolog–handler:

```
(def-prolog-handler (name (bdgs &rest parameters))
  &body body-forms)
```

The form of the macro is similar to defun with the difference that the first parameter bdgs is always the name of the variable that is bound to the current Prolog prove environment, i.e. the current variable bindings. The rest of the parameters are bound to the parameters of the predicate named by name. The lisp function gets the current set of variable bindings and must return a list of (extended) variable bindings. That way, the predicate can add choice points and new variable bindings to the current prove environment. A return value of NIL indicates a failure and the Prolog interpreter backtracks to the last choice point.

Listing 2.6: *The definition of the Prolog handler for the predicates true, false and lisp–pred.*

```
1  (def-prolog-handler true (bdgs)
2    (list bdgs))
3
4  (def-prolog-handler fail (bdgs)
5    nil)
6
7  (def-prolog-handler lisp-pred (bdgs pred &rest args)
8    (when (apply (symbol-function pred)
9                 (mapcar
10                  (lambda (var) (var-value var bdgs))
11                  args))
12     (list bdgs)))
```

Listing 2.6 shows simple examples, the definition of the Prolog handlers for the predicates true and false. true always succeeds and false always fails.

Many built-in predicates and functors, e.g. and, or, findall and cut are implemented as Prolog handlers. However, the implementation of Prolog handlers is rather error prone since all different cases for bound and unbound variables and unification need to be handled manually. Thus, the system provides the Prolog handlers lisp–fun which allows for calling arbitrary Common Lisp functions and bind their return value to a variable and lisp–pred (see Listing 2.6) to use Lisp functions as Prolog predicates. Listing 2.7 shows an example how lisp–fun and lisp–pred can be used to interact with Common Lisp from CRAM Prolog programs.

Listing 2.7: *The implementation of the predicate lisp–type for unifying the type of the value bound to a Prolog variable.*

```
1  (def-fact-group lisp-types ()
2    (<- (lisp-type ?var ?type)
3      (bound ?var)
4      (not (bound ?type))
5      (lisp-fun type-of ?var ?type))
6
7    (<- (lisp-type ?var ?type)
8      (bound ?var)
9      (bound ?type)
10     (lisp-pred typep ?var ?type)))
```

As can be seen, two versions for the predicate lisp–type are implemented, one for querying the type of a value bound to the Prolog variable ?var and one for asserting that the variable has a specific type.

2.3 Symbolic Plan Parametrization

To make robot control programs as general and flexible as possible, it is important to defer decisions on parameters such as the location of the robot base to perform an action as long as possible, as for instance done in least commitment planning [Weld, 1994]. In CRAM plans, plan entities such as actions, locations and objects are specified using designators. CRAM designators are a compact way of specifying properties of these entities using key-value pairs. For instance, in CRAM plans, we use the following set of properties to describe a red cup that is on the table:

```
((type cup) (color red) (at location-on-table))
```

where location-on-table is bound to a location designator with the properties:

```
((on counter-top) (name kitchen-table))
```

Basically, each additional key-value pair adds a constraint that limits the space of valid solutions of the designator. For valid solutions for an object designator that has the only property (type cup) is any object with the correct type. By adding an at constraint, the system is free to chose any cup that is at the correct location. Location designators do not describe single points in space but just define constraints for a location. Any location fulfilling all constraints must be considered as a valid solution.

2.3.1 Designator Concepts

Currently, CRAM implements three different classes of designators: action designators, object designators and location designators. While they differ in the way they are used in the system and converted to lower-level parameterizations for the robot's

functional components such as perception or navigation, they share a number of common properties.

Equation and side effects. Since CRAM programs are normally highly concurrent, one of the most important properties for all variables and in particular variables that are shared between concurrent processes is that they are read only for the user. Once created, an object should not change any of its user visible data slots. This naturally holds for designator objects as well. Once instantiated, the properties of a designator object must not change anymore and once resolved, a designator solution must not change anymore. Instead, if two designators describe the same entity with different solutions, two separate designators must be created which can be linked together by *equating* them. If two designators are equated, they describe the same entity. Often, multiple solutions might exist for a single set of designator properties that are all valid. For instance, we can find an infinite number of solutions for a designator that describes locations on a table. The designator API provides the functionality to request a new solution if the system decides that a solution is not appropriate for the current action. If the corresponding API function is used, it returns a new solution for the designator of interest that has been equated to the original designator.

Effective designators. Designators can be bound to arbitrary data objects. By *resolving* a designator, we generate this data object and bind it. Once this binding is established, it must not be changed anymore. However, it is not necessary that designator bindings are generated deterministically. When searching for an object of type cup twice, the second solution is not necessarily equal to the first one. Not all designators can be resolved or designator resolution might require external actions. For instance, the resolution of object designators involves navigation to a location from where an object might be visible and the invocation of perception routines.

Designators that have a solution bound to it are called *effective designators*. In addition to the actual data object, effective designators contain a time stamp for storing the time when the data object was created, i.e. when the designator turned into an effective designator.

The temporal ordered set of all designators that are equated to each other allows us to reconstruct the history of an entity over the execution of a CRAM program and it allows us to track the system's believe about object identities. For instance, let us

consider an object designator that is describing an arbitrary object (i.e. type object) that is first picked up from the table and then put down at the counter top. Such a pick-and-place plan first needs to find the object, then grasp it, move to a put-down location and put down the object. Initially, we describe the object and its location with designators with the following descriptions:

$$\begin{aligned} o_0 &= \text{(object ((type object) (at } l_0\text{)))} \\ l_0 &= \text{(location ((on table)))} \end{aligned}$$

Please note that the object designator is a non-effective designator. The robot navigates to a location close to the table and starts searching for objects. If it detects an object, for instance a cup, a new designator is created and equated to the original designator. This new instance is an effective designator since the result of the perception subsystem with lower-level information about the object is bound to the new designator's data slot. This designator solution, i.e. the low-level data structure returned by perception might include 3D models of the object, color information, texture or whatever is relevant in the context of the CRAM system. The properties of the newly created designator do not need to be equal to the original designator neither do they need to include any properties of it. The equate relation only states that two designators are describing the same entity. In our example, the properties of the newly created object designator and its location might be as follows:

$$\begin{aligned} o_1 &= \text{(object ((type cup) (at } l_1\text{)))} \\ l_1 &= \text{(location ((pose <pose-of-object>)))} \end{aligned}$$

The location designator l_1 contains the pose of the object as it was detected by the perception routines. The new object designator contains enough information in its low-level data structures to enable the perception system to redetect the object and to allow the grasping routines to compute a grasp and pick up the object. After grasping the object we know that it is now in the robot's gripper. The system generates a new non-effective designator from the object designator used for grasping by copying all properties but the "at" property. The location is bound to a location designator describing a location in the robot's gripper.

$$o_2 = \text{(object ((type cup) (at } l_2\text{)))}$$
$$l_2 = \text{(location ((in gripper)))}$$

The location l_2 basically states that the object is now connected to the robot's gripper and will move together with it. It is not equated to the object's previous location because although the object might still be at the location where it was detected, after grasping its location is semantically different and we want to reflect this in a new, different, location designator. After putting down the object, again a new object designator that is equated to the previous one is generated, again with a new location designator:

$$o_3 = \text{(object ((type cup) (at } l_3\text{)))}$$
$$l_3 = \text{(location ((pose <pose-of-object>)))}$$

The location l_3 is equated to the location designator passed to the put-down plan since it is a solution for the put-down location. The temporally ordered set of equated object designators originating from o_0 contains information about how the knowledge about the object referenced in the plan developed over time. Basically, the object designator o_3 not only references the object but also encodes that it is exactly the object that was picked up and put down by the robot. If the robot should perform subsequent actions on exactly this object, the object designator o_3 has to be used in a plan. If a different object should be used, a new object designator has to be created that does not contain any history information.

The Designator Interface. Class 10 gives an overview of the class definition of all designators. The CRAM designator implementation provides a number of API functions to create, equate and resolve designators. In the following we will give a brief overview of the API provided by the designators package.

make—designator. To construct a new designator, instead of using Common Lisp's make—instance , the user should use the method make—designator. Its signature is as follows:

```
(make-designator type properties &optional parent)
```

Class 10 The class definition of the class designator which is the base class for all three currently supported designators.

class DESIGNATOR	
slot PROPERTIES:	The list of designator properties, i.a. a list of tuples of key-value pairs that describe the constraints on the entity represented by this designator.
slot EFFECTIVE:	Indicates if this designator is an effective designator.
slot DATA:	A pointer to the low-level data structure bound to an effective designator.
slot TIMESTAMP:	The time stamp indicating the resolution time of this designator.
slot PARENT:	The designator this designator has been equated to.
slot SUCCESSOR:	The next designator in the chain of equated designators.

The method returns a newly constructed designator with the given type (either action , object or location) and the given properties. If parent is specified, the newly created designator is immediately equated with it.

make–effective–designator. This method should be used to construct a new *effective* designator for a given designator. Since a designator is either already effective at creation time (action and location designators) or is made effective based on an already created non-effective designator (object designator), the method make–effective–designator always requires a reference designator that serves as a creation template. The signature of the method is as follows:

```
(make-effective-designator reference-designator
    &key new-properties data-object timestamp)
```

The keyword argument new–properties allows to replace the reference designator's properties and the mandatory keyword argument data–object sets the data slot required in all effective designators. The optional timestamp parameter allows to use a time stamp that differs from the current system time.

reference. The method reference returns the data object of an effective designator. The data object might be generated at the first call of the reference method by different resolution methods as explained in Section 2.3.2.

```
(reference designator)
```

equate. The method equate has exactly two parameters and equates both designators. Its signature is as follows:

```
(equate parent successor)
```

desig–equal. The method desig–equal returns T if two designators are in the same set of equated designators, i.e. if they are describing the same entity. Its signature is as follows:

```
(desig-equal designator-1 designator-2)
```

current–desig. The method current–desig returns the newest designator in the set of equated designators specified by the parameter of current–desig.

```
(current-desig designator)
```

newest–effective–designator. The method newest–effective–designator returns the newest effective designator that has been equated to the designator specified by the method's parameter.

```
(newest-effective-designator designator)
```

As can be seen in Class 10, the designator class contains the two data slots parent and successor. These slots are used for implementing designator equation. Basically, all designators which are equated to each other are members of the same doubly-linked list, implemented using the data slots parent and successor. The equate method first searches for the last designator in this parent designator's list by iterating over it until it finds a designator which has an unbound successor slot. Then it finds the first element in the successor's list, i.e. the designator which has an unbound parent slot. Finally, the successor slot of the last parent is set to the first successor and the successor's parent slot is set to the last parent. That way, the two sets of equated designators are merged. To avoid circular equations, the system enforces that only designators are equated which have not been equated yet.

2.3.2 Designator Resolution

While the public API for using designators is fixed and consistent for all three currently supported designator types, the designator resolution mechanism is highly flexible and modular and must be implemented by a user of the CRAM system. Object designators, action designators and location designators describe entities in entirely different domains and thus the mechanisms for resolving them are different from each other. For instance, while action designators can normally be resolved immediately without requiring any sophisticated knowledge about the current environment, object designator resolution most often requires interaction with the environment. The robot has to move close to the believed location of an object and move the robot's camera in order to detect the object. Location designator resolution on the other hand often requires deep knowledge about the geometry of the environment and in general much more valid solutions can be found for a location designator than for, e.g. objects. For instance we can find an infinite number of solutions for locations on the table while we have only a low number of cups in a kitchen.

Object Designator Resolution. Object designators describe objects that need to be detected by a perception system such as ROS' tabletop object detector [2] or COP [Klank, 2012]. The result of these perception systems is then converted into a new object designator that is equated to the original designator. The resolution of object designators thus requires interaction with some of the robot's sub-systems, including navigation to drive to a location from which the object can be detected, the robot's pan tilt unit to point the camera on a possible location of the object and the actual perception subsystem. Currently, the only constraint on the implementation of an object designator resolution mechanism is that the data slot of an effective object designator needs to be bound to an instance of the class object−designator−data or a class derived from it. The process of resolving an object designator is normally as follows:

1. Move the robot and its cameras to point to the right location in order to be able to detect the object.

[2] http://ros.org/wiki/tabletop_object_detector

2. Parse the designator's properties and generate the actual parameters for the perception sub-system. For instance, select a set of perception algorithms that are likely to find the object.

3. Call the perception subsystem.

4. Use the output of the perception subsystem (possibly many detections) to select one result that matches the designator's properties.

5. Convert the detection result to an instance of object–designator–data or one of its subclasses.

6. Create a new effective designator from the data instance.

7. Equate the original designator and the new effective designator.

This process must be at least partly implemented in the robot's high-level control program since the robot often needs to actively move to be able to detect objects. This is not necessary for resolving action and location designators.

Action Designator Resolution. Action designator resolution is depending on task knowledge and reasoning about actions to perform. It is implemented by using the solutions for the Prolog predicate (action–desig ?designator ?solution). This allows the user to implement the process of generating a solution for an action designator completely in Prolog. When the reference method is called on an action designator, it generates the lazy list of all solutions for that designator by calling Prolog to find solutions for the predicate action–desig. If no solution can be found, an error is thrown because the system does not have any means for resolving the designator. Otherwise, the first solution is used as a solution for that designator. The remaining solutions are used to generate equated designators when further solutions for the designator are requested, e.g. when a submodule that executes a designator decides that the solution is invalid in the current execution context.

One of the simplest action designators is the action designator for navigating the robot to a specific location. The following example shows the possible set of properties for such an action designator:

$$\begin{aligned} a_0 &= \text{((type navigation) (goal } l_0\text{))} \\ l_0 &= \text{((to see) (obj cup}_0\text{))} \end{aligned}$$

Suppose that the desired solution for the action designator a_0 is the location designator l_0 bound to the goal property. The corresponding Prolog predicate action–desig can then be defined as follows:

```
1 (def-fact-group navigation-action-designator (action-desig)
2   (<- (action-desig ?designator ?solution)
3     (desig-prop ?designator (type navigation))
4     (desig-prop ?designator (goal ?solution))))
```

Please note the use of the helper predicate(desig–prop ?designator ?property). The predicate holds for all properties that are specified in the designator instance bound to ?designator.

Location Designator Resolution. Location designator resolution is the most complex resolution mechanism in the current CRAM system. The reason is that it needs to be flexible enough to allow for the implementation of different generation mechanisms, e.g. simple heuristics, more complex sampling mechanisms or reasoning and planning. Some generation mechanisms might only generate one solution while others might be able to generate an infinite number of locations and the precedence order of generation mechanisms has to be defined. The system must be modular enough to allow for plugging in different modules for location generation, e.g. for using a semantic map or for using inverse kinematics or an inverse reachability map to find locations to reach objects.

The current mechanism for generating poses consists of three steps:

1. Generate a lazy list of solution candidates.
2. Draw one solution candidate.

3. Verify that the solution candidate is a valid solution. If not, draw another solution.

The system provides interfaces for registering generation functions and validation functions. That allows the user to provide libraries that add a certain functionality to designator resolution, e.g. the use of an inverse reachability map to find locations to grasp an object.

More specifically, the user can use the function register–location–generator to register a generator function. Generator functions are functions that are executed one after the other and that receive the location designator to be resolved as a parameter. They return a (lazy) list of *solution candidates*. All results are then concatenated and form one single lazy list of possible solutions for that location designator.

To specify validation functions, the user can use the function register–location–validation–function . Such a validation function is a function that receives two parameters, the designator to be resolved and a solution candidate. The validation function must return one of the following values:

- :reject The solution is invalid and should be rejected immediately.
- :accept The solution can be accepted. Only if none of the validation functions returns :reject the solution is accepted though.
- :unknown The validation function cannot decide if the solution is valid or not.
- :maybe–reject The solution is rejected if no other validation function explicitly accepts it, i.e. if all other validation functions return :unknown.

To validate a designator solution, the system iterates over all registered validation functions and calls them one after the other. Only if all functions either return :accept or :unknown, a the solution is accepted as a solution for the location designator to be resolved. If at least one validation function returns :maybe–reject, the solution is only accepted if at least one validation function explicitly returned :accept and no function returned :reject . Otherwise it is rejected.

Both API functions for registering generators require a priority parameter that is used to control the order of evaluation. A smaller number indicates that the function should

be evaluated earlier. In particular generator functions need to be ordered because their return value can be a lazy list with infinite length. Such solutions need to evaluated last because any subsequently generated solutions would not be considered. The priority parameter for validation functions allows for performance optimization. For instance, if a function terminates very quickly, it might make sense to evaluate it first, in particular when it rejects many solutions.

To make the functionality of the system clear, let us consider a simple example. Assume we want to resolve location designators for the robot to reach a specific object. A corresponding location designator specification might be defined as follows:

$$l_0 \quad = \quad ((\text{to reach}) \ (\text{object } o_0))$$

A location for the robot's base must be a location that the robot can navigate to, i.e. a location that is marked as free in a 2D navigation map as used in most localization and navigation programs, e.g. ROS' move_base[3]. Thus, we can define a generator function that only yields poses that are inside free areas on the 2D map. A second constraint is that the object's pose must be reachable by the robot. The validation function might then use an inverse kinematics module to verify if the object is reachable from that pose. More specifically, in the example, we define one generator function and two validation functions. The generator function generates poses that are on the free space and not closer than the robot's radius to occupied space in a 2D map. The first validation function just verifies if a pose is in the same set of navigatable poses since other pose candidates generated by a different generator function might lead to unreachable poses or to collisions with the environment. The second validation function calls inverse kinematics to verify that the object is actually reachable.

2.4 Process Modules

One of the main goals of the CRAM software toolbox is to allow for implementing general high-level robot control programs that can run on different robot platforms. The system needs to provide the means for abstracting the specific properties and interfaces of the actual robot hardware in use. In addition, the system needs to support

[3]http://ros.org/wiki/move_base

execution of plans in simulation and projection of plans which is essentially a light-weight simulation mechanism.

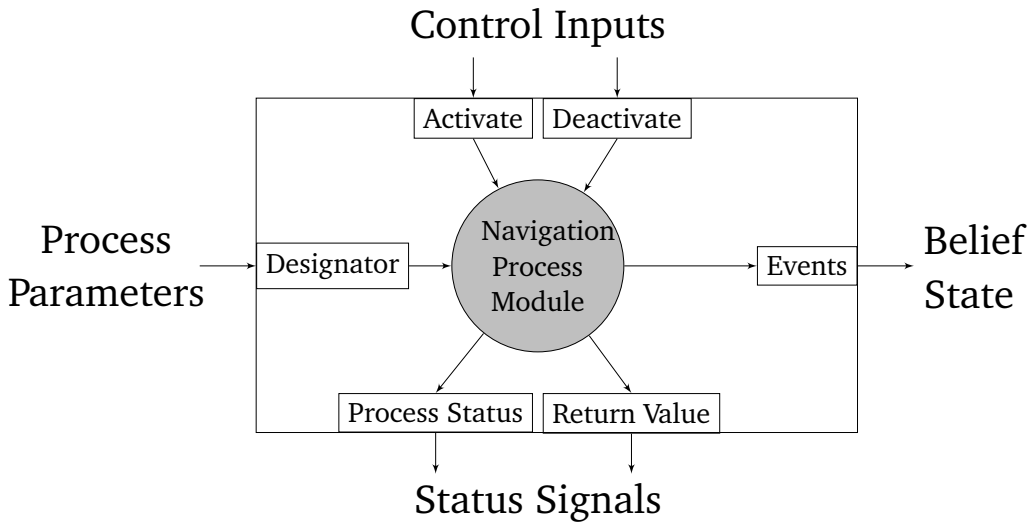


Figure 2.6: *Process module encapsulating a navigation control process. The input is an action designator specifying that the action is a navigation action and the containing the goal location as represented by a location designator, e.g.* (a location (to see) (obj cup)).

One of CRAM's components is a framework for implementing so-called process modules. Process modules provide the interface between high-level control programs and the robot's hardware. All interaction of the robot with the world thus must be encapsulated in process modules. The CRAM *Process Modules* framework defines a clear and simple but also flexible interface for encapsulating interaction with the robot's hardware. Process modules are started up when the robot starts to execute plans and shut down in the end. During the robot's operation, all process modules run in parallel to each other and the high-level control program and wait for input in the form of an *action designator*. Action designators are symbolic descriptions of the actions to be performed. For instance, to move the robot to a location specified by the location designator l_0, a module providing navigation functionality might accept an action designator with the following description:

$$a_0 \quad = \quad ((\text{type navigation})\ (\text{goal}\ l_0))$$

Besides the input which must be an action designator, all process modules provide a

status fluent that allows to query if the module is running, free and waiting for input or if the last action succeeded or failed. In addition, they can pass a result value to the calling process, normally the high-level plan that triggered an action. Process modules return result values similar to common functions. This functionality is mainly used in process modules such as perception where detected objects are returned to the calling process. Additionally, a second mechanism is used to allow for transparently updating knowledge bases and belief states. Process modules send notification events for each change in the environment they were causing. For instance, each detection of an object generates an object–perceived event and whenever the robot moves a robot–state–changed event is generated. More complex sequences of events are generated by manipulation. When grasping an object, the robot might move one arm first to a pre-grasp location, then to the object to be grasped. Each of these actions causes one event to notify all process in a CRAM system that the robot changed its state, i.e. moved. The granularity of events and the frequency in which they are generated is up to the user's needs and may vary in different domains.

To implement a process module, the user has to provide at least the following components:

1. Action designator resolution methods for all action designators supported by the process module.

2. The predicates matching–process–module and available–process–module. The predicates hold for all action designators that can be executed by the process module if it is available in the current environment and on the robot.

3. The actual implementation of the process module. The input is the action designator and the implementation must emit events using a specific method, on–event .

Special care has to be taken if the task that sent an input to the process module is suspended while the process module executes the action. In that case, all actions triggered by the suspended process must stop as well. If a task is suspended while it is executing either the method pm–execute or monitor–process–module, the process modules framework ensures that the process process module cancels the executed or monitored action and restarts it with the same action designator as soon as the task is woken up again. That way, given a process module handles cancellation correctly,

the user can be sure that the robot stops moving as soon as the control program is suspended and continues when it is woken up again. The library handles non-local exits, for instance caused by evaporations, in a similar way: actions executed by an evaporated task are canceled and the process module must stop executing it.

While the basic CRAM Process Module library only defines the interfaces for implementing the process module itself, for starting and stopping it and for sending commands and waiting for results, CRAM contains other libraries that define common events and common error objects. In this section only the core functionality of process modules will be discussed. Events and error objects will be shown in Chapter 3.

2.4.1 The CRAM Process Module Interface

The CRAM Process Module library provides functions for starting and stopping process modules, for sending input and for monitoring execution. At the moment, two implementations for process modules exist, synchronous and asynchronous process modules. The base class is the class abstract—process—module which is shown in Class 11. It is mainly used internally for implementing the basic API.

Class 11 The class process—module. It is the base class for all process modules.

class ABSTRACT-PROCESS-MODULE

slot NAME:	The name of the module. The name is used to uniquely identify a process module.
slot INPUT:	The input fluent of the process module. Whenever a new value is assigned, the process module starts to execute the specified action.
slot STATUS:	The status fluent of the process module.
slot CANCEL:	A boolean fluent that is used to trigger cancellation.

The status fluent's value is one of the following:

- : offline The process module is not running.
- :running The process module is processing input.
- :waiting The process module is waiting for input.

- :failed Similar to :waiting with the difference that execution of the most recent action designator failed.

While the status fluent can be used to monitor the module and, for instance, check if it has been started up, it is mainly used internally and should not be accessed by the user.

More important than the definition of the base class is the definition of the API. In the following we will explain the most important functions for running process modules and sending input.

pm–run. All process modules need to implement the method pm–run. This method must not terminate. The systems starts a new operating system thread for each process module to be run and executed the process module's pm–run method in it. When the process module is shut down, its thread object is shut down. This also causes the pm–run method to be unwound.

with–process–modules–running. The system provides the macro with–process–modules–running as an easy to use interface for making sure process modules are running the execution of a body. The following example shows how the macro can be used:

```
(with-process-modules-running
      (pr2-manipulation-process-module
       move-base-navigation-process-module)
  (make-breakfast))
```

As can be seen, two process modules, one named *pr2-manipulation-process-module* and one named *move-base-navigation-process-module* are started up before executing the method make–breakfast. After executing the macro's body, the two process modules are shut down. The signature of the macro is as follows:

```
(with-process-modules-running (&rest module-specifications)
  &body)
```

Module specifications can either be symbols naming the process modules to be started up or pairs of the form (alias process-module-name) to allow for renaming process modules. While this feature is used rarely, it allows for hot-swapping or replacing the process modules that are used by a plan in the caller, i.e. it allows for changing process modules without changing actual plans.

pm-execute. The method pm-execute allows the user to send an action designator to a process module. Its signature is:

```
(pm-execute process-module input-designator)
```

The method verifies that the indicated process module is running and sends the input designator to it. The process module interface does not define at which point the pm-execute method terminates, i.e. it might block until the action is finished or terminate immediately to allow for background execution of the action.

monitor-process-module. The method monitor-process-module allows the user to block until a process module finishes execution. Non-blocking process modules might be executing multiple designators at the same time and monitor-process-module can either monitor the execution of all designators or just a specific subset. The signature of the method is:

```
(monitor-process-module process-module &key designators)
```

pm-cancel. The method pm-cancel is used to cancel the execution of a single action designator.

pm-status. To get the current status of a process module, the user can use the method pm-status. It returns one of the following symbols: :running, :failed, :waiting or :offline .

2.4.2 Synchronous and Asynchronous Action Execution

Although the method pm-execute might block until the process module finished executing an action designator, it is not required by the process module interface to do so. This allows for sophisticated execution patterns where not only one action designator can be processed by a process module and where the robot can continue executing its plan until the system infers that it needs to wait for a process module to finish. For instance, if two sub-plans that are running in parallel need to move one of the two robot's arms, in many situations it is possible to execute both action designators in parallel given they specify motions for different arms.

Synchronous process modules. The simplest and probably most robust implementation of process modules are synchronous process modules. The corresponding Common Lisp class in CRAM is process-module. Such a process module can only process one action designator at a time. The system ensures that callers block until the process module is free, if it is already processing an action designator. In this implementation, pm-execute is a blocking call and waits until the process module finishes the action. In case of an error, it rethrows the error in the calling task. The major advantage of synchronous process module is that they are easy to reason about by the user. Their implementation is rather strait forward since the user does not need to deal with synchronization or monitoring of errors. The CRAM process modules library provides the

macro def–process–module that allows for the definition of synchronous process modules in a way that is very similar to the definition of normal functions. Listing 2.8 shows the implementation of a simple navigation process module that could, for instance, just send a goal to a ROS navigation node.

Listing 2.8: *Very simple implementation of a synchronous navigation process module.*

```
1 (def-process-module navigation (input)
2   (let ((goal-location (reference input)))
3     (or (drive-to-navigation-goal (reference goal-location))
4         (fail 'navigation-failed))))
```

Similar to defun, the macro def–process–module requires the name of the process module as its first parameter. In contrast to functions, process modules do not support arbitrary lambda lists but accept only one parameter, the input designator which must be an action designator. The signature of the macro def–process–module is as follows:

```
(def-process-module (input-designator) &body body)
```

In the example above, the input designator is first dereferenced. We assume that the result is the goal location represented by a location designator. In our example, the method drive–to–navigation–goal accepts the goal pose which is the result of dereferencing the goal location designator. In case it returns NIL which indicates a navigation failure, the module creates a condition object and throws it.

The user does not need to provide any special handling for cancellation. If a synchronous process module is canceled while running, the system causes a non-local exit of the process-modules body and executes all unwind forms.

Asynchronous process modules. While synchronous process modules are easy to implement and easy to reason about, they might cause major performance flaws. For instance, if the robot should grasp two objects, one with each arm, it is not required to execute both actions sequentially if the two objects are reachable for the robot. Asynchronous process modules also allow for more sophisticated movement schemes and the combination of several trajectory goals into one smooth motion. Suppose the robot should reach for a cup, grasp it, lift it and carry it. To achieve a smooth motion without having the robot to stop after each single action, we can either define one big macro-action encoded as one single action designator or we can provide a way to sending the different action designators one by one, store them in a queue and let the process module combine them. The former solution can be implemented using a synchronous process module but is much less general since one action designator has to be defined for each combination of the simple actions. This solution also makes the high-level plans for manipulation much simpler, i.e. less expressive since the system only has means to reason about high-level plans. For implementing the latter solution, process modules must be asynchronous since the pm—execute method must not block to allow high-level plans to continue executing subsequent actions and send more action goals that can be combined to one continuous motion by the process module.

To summarize, the implementation of asynchronous process module should show the following properties to increase plan performance and improve trajectory execution of manipulation actions:

- pm—execute should not block if it can execute an action designator immediately. This is the case if the process module is not currently processing any action or if the new action designator requires the control of a different hardware unit. For instance, one action designator for the right arm and one for the left arm might not cause pm—execute to block while two actions for the right arm could cause it to block.

- pm—execute should not block if the action can be combined with an already processed action, e.g. by executing the actions subsequently.

- pm—execute should block if the action cannot be executed yet, e.g. when the navigation process module is active, manipulation might need to wait for the robot

to stop. pm–execute should terminate in this case as soon as it starts executing the action.

- The user must be able to check for errors and get return values of a process module at well-defined synchronization points.

The CRAM Process Modules library implements the above constraints in the class asynchronous–process–module. Using asynchronous process modules is slightly more complex than using synchronous modules since the complete functionality must be based on asynchronous handling of input designators. Additionally, a mechanism to decide if the method pm–execute needs to block must be provided and the user needs to explicitly handle cancellation.

The implementation of asynchronous process modules provides all methods required by the interface as defined in Section 2.4.1. However, compared to synchronous process modules that can conveniently be defined using the macro def–process–module, the definition is slightly more complex. Instead of a single macro, the implementation uses CLOS and the interface that needs to be implemented by the user consists of three methods:

- (on–input process–module input–designator): This method is executed whenever the user calls pm–execute on the process module process–module. Please note that even if inputs arrive in parallel, calls to on–input never happen concurrently. The user must make sure that on–input terminates because subsequent input designators can only be processed after the method terminated.

- (on–cancel process–module input–designator): The implementation of this method is optional. It is called whenever the user executes the method pm–cancel.

- (synchronization–fluent process–module): This method must return a fluent that indicates if a call to the method pm–execute needs to block. Before actually sending an input designator to the process module, pm–execute waits for the returned fluent to become non-NIL.

Additionally, two methods for allowing the user to either cleanly terminate a process module or to trigger a failure are provided:

- (finish–process–module process–module &key designator): Signals the successful termination of the execution of the action specified by designator.
- (fail–process–module process–module error &key designator): Signals a failure while processing designator. The failure is rethrown when the user calls monitor–process–module .

In order to implement an asynchronous process module, the user first needs to create a new class that inherits from asynchronous–process–module. Then at least the two methods on–input and synchronization–fluent must be implemented. If actions might run for a longer time, the user should also implement the method on–cancel to allow for clean cancellation of actions.

In contrast to synchronous process modules, it is not easily possible to define the behavior of asynchronous behavior on suspension of a caller. Since asynchronous process modules are processing in parallel to actual plan execution, plans need to explicitly add statements to cancel specific action designators on errors or suspension.

2.5 The CRAM Plan Library

In this section we will explain what CRAM plans are and how they are written. We will give an overview of the plans that have been implemented in the current system and how the underlying implementation works.

2.5.1 Transparent Plans

We define CRAM plans as robot control programs that cannot only be executed but also reasoned about. CRAM plans are carefully designed programs that contain semantic annotations that can be used to infer their purpose in an underlying logic. We call symbolic plan annotations *goals*. They describe the purpose of a specific code part symbolically and the semantics of these symbolic plan annotations is well defined. It can be used to infer the purpose of certain pieces of code, if they were executed as expected and for projecting a plan in order to find common flaws in execution. As a more specific example, let us consider a function that should grasp an object. Instead

of using def–cram–function, we specify a pattern in a first-order logic that indicates that the purpose of the plan is that the robot holds the object in its gripper when the plan finishes. We name the goal "(achieve (object–in–hand ?object))".

Listing 2.9: *Simplified plan for picking up an object.*

```
(def-goal (achieve (object-in-hand ?object))
  (with-designators
      ((grasp-action (action `((type trajectory) (to grasp)
                               (obj ,?object))))
       (lift-action (action `((type trajectory) (to lift)
                              (obj ,?object))))
       (loc (location `((to execute) (action ,grasp-action)
                        (action ,lift-action)))))
    (at-location (loc)
      (perform grasp-action)
      (perform lift-action))))
```

A simplified version of the complete plan is shown in Listing 2.9. Please note that for readability reasons the code snippet does not contain any error handling.

Goal definitions are different from normal function definitions in Common Lisp or the definition of CRAM functions because the selection of the code to execute is based on *pattern matching* instead of simply looking up the function name and executing the corresponding code. The reason is that similar goal patterns might require different plans to be executed. For instance, the goal (achieve (loc Robot ?location)) should move the robot's base to a specific location while (achieve (loc ?object ?location)) requires the robot to navigate to a location close to the object, grasp it, move to a location to reach the put-down location and place the object. In CRAM, a goal consists of two parts, the goal *name* and a pattern that is called *occasion*:

```
(def-goal (name &rest occasion) &body body)
```

The occasion can be an arbitrary list with variables being prefixed with a question mark. Examples for occasions that are defined for the achieve goal include (loc Robot ?location), (loc ?object ?location), (object-in-hand ?object) etc. The complete list of goals is discussed in Section 2.5.2.

Before defining a goal, it must be declared by using the macro declare-goal. Besides declaring the goal, the macro allows the user to execute code before the actual goal code is executed and to prevent execution of any code, for instance in cases where the goal has been achieved already. The signature of declare-goal is:

```
(declare-goal (name (&rest occasions)) &body body)
```

The macro is implemented to define a function named according to the declared goal. First, body is executed in an anonymous block to allow for early exits by using return. This allows the user implement checks if the goal has to actually execute any code or if the corresponding occasion already holds. That way, the user has fine control over the execution of actual goal functions. This functionality is used in the current implementation to only achieve goals that do not hold yet. For instance, if the goal (achieve (object-in-hand ?object)) is executed and the object is already in the robot's gripper, no additional actions will be performed an the call to achieve will just terminate successfully. In case the code specified in body did not cause an exit by calling return or an equivalent exit, the function generated by declare-goal tries to select the actual code to be executed that was defined using def-goal by matching the occasion specifications against the defined goals. All variables in the occasion specifications, i.e. all symbols that are prefixed with a question mark, are bound in the lexical scope of the goal by using the result of the pattern matching step. This approach is similar to functional languages that support pattern matching, e.g. languages from the ML family, e.g. Haskell or OCaml.

Using pattern matching in goals suffers from one problem: which goal to select if several goals match. CRAM's implementation behaves similar to other functional languages in that case. The matching algorithm processes goal patterns in the order of their definition until one matches. That means the first matching goal is used. Special care has to be taken when goals with similar occasions are defined in different files because it might not be clear in which order the files are loaded, depending on how file dependencies are defined. It is highly recommended that goals for such patterns are always defined in the same file.

2.5.2 Goals in the Current CRAM System

In this section we will give an overview of the high-level plans and goals that are implemented in the current CRAM system. CRAM provides a complete plan library with goals for picking up and putting down objects, for opening and closing articulated objects such as drawers and doors and for perceiving objects and higher-level logical states (i.e. occasions) in the world. The plans are implemented in a general way and require four process modules to be implemented: a perception process module, a manipulation process module, a navigation process module and a process module moving the robot's sensor head to in order to allow the perception process module to find objects.

(loc ?obj ?loc)	Make sure that the object specified by the object designator ?obj is at the location specified by the location designator ?loc.
(loc Robot ?loc)	Make sure that the robot is standing a the location specified by the location designator ?loc.
(object–in–hand ?obj)	Grasp the object specified by the object designator ?obj if not already grasped.
(object–placed–at ?obj ?loc)	Place an object that is in the robot's gripper at the location specified by the location designator ?loc.
(object–opened ?obj)	Make sure that the articulated object specified by the object designator ?obj is open.
(object–closed ?obj)	Make sure that the articulated object specified by the object designator ?obj is closed.

Table 2.2: *Occasions that can be achieved in the current implementation of the* CRAM *plan library.*

achieve. The most important goal for expressing changes in the world that are to be performed by the robot is achieve. Its signature is as follows:

```
(achieve occasion)
```

Informally, a call to achieve that terminates successfully indicates that the system believes that the corresponding occasion holds in the robot's belief state. A list of occasions defined in the current implementation is shown in Table 2.2. Please note that achieving an occasion does not necessarily mean that the robot has to execute any actions. For instance, if the robot knows that an occasion already holds, achieve will terminate successfully immediately.

at–location. In contrast to achieve, the at–location goal has a different form. It is not implemented using declare–goal and def–goal but as a macro based on the lower-level task tree macro with–task–tree–node. The reason is that at–location specifies that a specific code block must be executed at a specific location. For instance, the following code preforms a grasp action at the location specified by pick–up–location:

```
1 (at-location (pick-up-location)
2   (achieve '(object-in-hand ?obj)))
```

The at–location macro monitors the current location of the robot and only starts to execute the body forms when the robot is at the specified location. In case the robot moves away during execution of the body forms, at–location evaporates and restarts them as soon as the location constraint is established again. This behavior is especially useful when plans are executed in parallel. For instance if the robot is supposed to grasp two objects that are both reachable from one location, i.e. the location of the

robot in one pick-up plan is also a solution for the location designator of the second plan, both grasp actions might start executing in parallel.

with-designators. Similar to Common Lisp's let, we define the macro with-designators. It should be used to create all designators that are used in a plan and establishes the corresponding variable bindings in the current lexical context. Its signature is as follows:

```
(with-designators (&rest binding-spec) &body body)
```

Bindings in the with-designators macro require a variable name the designator will be bound to, the class of the designator (e.g. action, location or object) and the properties of the designator. Bindings are specified as follows:

```
(name (class properties))
```

The with-designators macro is important because it allows to find all designators defined and used in a plan. Per default, the macro uses the function make-designator to create a designator. But to allow for more sophisticated methods to create the designator, for instance to re-use already instantiated designators, with-designators allows to replace the create method. This feature is particularly important if designator solutions are generated in a background process in parallel to plan execution by projecting the plan as explained in Chapter 5.

perceive-state. To decide if a certain occasion has to be achieved or if it holds already, the robot most often needs to perceive all objects referenced by an occasion. For instance, to check if the occasion (loc ?obj ?loc) holds, the object referenced by ?obj has to be detected. The signature of the goal to verify if a certain occasion holds is:

```
(perceive-state occasion)
```

The goal performs all steps necessary to check for the occasion, including moving the robot's base, moving the sensors and detecting all objects involved.

perceive-object. All manipulation actions require recent detections of the objects to be manipulated since objects might be moved by humans or the robot's self-localization might drift which means that previous detections might not be accurate enough anymore. To detect all objects that match an object designator, to re-detect an already detected object or to find one object that matches an object designator, CRAM provides the goal perceive-object. The following forms are currently implemented:

- (perceive-object 'all ?obj): Return a list of all (unequated) object designators that can be found and match the input object designator.
- (perceive-object 'a ?obj): Return one (equated) object designator that matches the input object designator.
- (perceive-object 'the ?obj): Re-perceive the input object designator. ?obj must be equated to at least one effective designator. Otherwise an error is thrown. The goal returns a new, equated object designator if the specified object could be found again.

perform. The interface between goals and the robot's hardware is provided by process modules which are parameterized by action designators. Given an action designator, the system is able to deduce all process modules that can execute it. This mechanism uses the CRAM Prolog engine and predicates that define the capabilities of all process modules. The goal **perform** takes an action designator, infers which process module should execute it and sends the designator to the respective process module. Its signature is as follows

```
(perform action-designator)
```

If multiple process modules can execute the designator, they are tried in the order of their definition until one process modules successfully executes the action. If all process modules failed, a composite failure containing all failure objects is thrown.

2.6 Related Work

Domain specific programming languages and language extensions are quite popular in the context of robotic applications, planning and artificial intelligence. Early examples are Firby's RAP [Firby, 1987] and McDermott's RPL [McDermott, 1993] which is the direct ancestor of the CRAM Plan Language. In fact, the CRAM Plan Language tries to closely reimplement RPL with a strong emphasis on the support for modern multi-core processors and middle wares such as ROS.

Newer examples for domain programming languages particularly used in robotics and artificial intelligence are URBI and one of its components, *urbiscript* [Baillie, 2005] and *Euslisp* [Matsui and Inaba, 1991]. *urbiscript* is a dynamic programming language with a C-like syntax with support for concurrent code constructs similar to RPL and the CRAM Plan Language, however with different syntax of course, and event based programming. *Euslisp* on the other hand is a Lisp dialect that provides geometry routines to allow for a tight integration of spatial and geometric reasoning. It is does not provide special constructs for reactivity and concurrent programming.

Besides domain specific languages, state machines are widely used for high-level robot control. One recent example for a library to implement state machines, including support for parallel processes is *SMACH* [Bohren and Cousins, 2010], which is a Python library.

TREX [McGann et al., 2008], a teleo-reactive executive implemented at the Monterey Bay Aquarium Research Institute, is a complex and powerful library for integrating planning, projection of future effects and reactivity.

The Prolog engine included in CRAM is derived from a Prolog interpreter implemented in Common Lisp as shown in [Norvig, 1992]. However, CRAM extends the interpreter with incremental computation of Prolog solutions using lazy lists and delayed computation. Additionally, the cut operator and mechanisms for easy integration with Common Lisp through Prolog handlers, i.e. Lisp functions that compute Prolog solutions and which are invoked by the Prolog engine are provided by the CRAM Prolog interpreter.

Although CRAM is not a cognitive architecture that tries to mimic the human mind, it can still be classified as a cognitive architecture since its main purpose is to provide the tool set to implement cognitive behavior by integrating reasoning, planning and robot control. Examples for other cognitive architectures are *ACT-R* [Anderson, 1993] and *Soar* [Milnes et al., 1992].

In CRAM, robot actions are implemented using the CRAM Plan Language while reasoning is implemented using the CRAM Prolog interpreter. However, the Prolog interpreter is designed to tightly integrate with robot control programs. The intention behind this approach was to allow for maximal performance and flexibility by using a compiled full-featured programming language on the one hand but not lose the power flexibility of modern reasoning engines. Approaches such as *Golog* [Levesque et al., 1997] and its successors *ConGolog* [Giacomo et al., 2000] and *cc-Golog* [Giacomo et al., 2000] demonstrate more integrated solutions that implement temporal calculi, namely the situation calculus in Prolog and use it for controlling robotic agents. However, Golog and its derivatives are restricted to pure symbolic reasoning and are restricted to Prolog data structures. Handling of time, i.e. actions with durations, synchronization and parallelism is problematic. However, cc-Prolog adds a *waitFor* statement inspired by the corresponding RPL function which enables it to deal with actions that have a duration.

2.7 Discussion

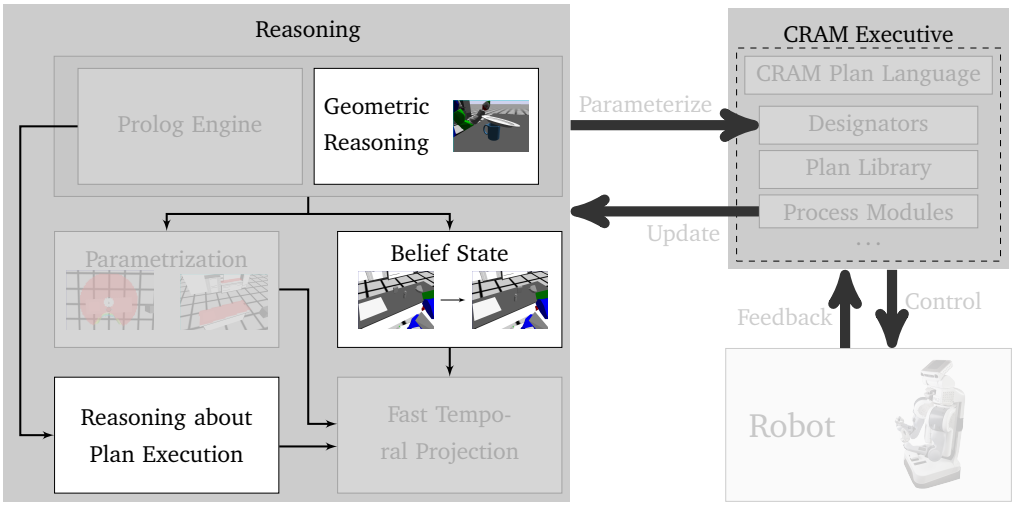
In this Chapter, we gave the implementation details of the CRAM core components, namely the CRAM Plan Language, the CRAM Reasoning Engine, Designators, Process modules and the CRAM plan library. We have shown how these components are in-

tended to be used to implement cognitive behavior on robotic agents and how these components fit together.

The CRAM Plan Language is the basis for implementing CRAM plans and the CRAM plan library and provides a domain specific language for expressing highly parallel control programs. The CRAM designators library provides a powerful mechanism for describing symbolic plan parameters and translating them to actual commands that can be executed by robotic agents. Designators are highly extensible and flexible and allow for integrating different reasoning mechanisms and most importantly integrate with the CRAM Reasoning Engine. CRAM's Prolog engine is especially designed for integration in programs, i.e. it allows for multiple queries in parallel and for the incremental generation of solutions. While CRAM plans as implemented in the CRAM Plan Library are highly general and not specific to a single robot platform, Process Modules provide a generic interface for communicating with the actual hardware and are highly platform specific.

Chapter 3

CRAM Reasoning Extensions



Besides providing a powerful domain specific programming language and libraries for implementing cognitive high-level executives for robotic agents, one of the most important features of CRAM is the tight integration of different reasoning engines and the executive. On the one hand, CRAM's reasoning systems have full access to all knowledge provided and gathered by the executive and on the other hand, using CRAM's reasoning mechanisms for decision making during plan execution is a core part of CRAM's executive.

One of CRAM's core libraries is a full-featured Prolog engine, entirely written in Common Lisp. A description of the Prolog engine can be found in Section 2.2. In this section, we will give a more detailed description of extensions and applications of the reasoning engine, including spatial and physics-based reasoning, the representation of the robot's belief state in CRAM and reasoning about plans and plan execution.

The work presented in this Chapter has been published previously in [Mösenlechner and Beetz, 2009], in [Mösenlechner et al., 2010] and in [Mösenlechner and Beetz, 2011].

3.1 Physics-Based Reasoning

A robot that is operating in a complex, dynamic environment needs to make many decisions while performing its tasks, for example, which objects to use in order to fulfill its goals, where to search for these objects, where exactly to place them and where to stand to be able to perceive, grasp or put them down. Classical hierarchical architectures for implementing mobile agents such as the 3T architecture [Bonasso et al., 1997] use a symbolic planner on the top-most level. In such planners, the representation of the environment is purely symbolic which makes the planning problem feasible but abstracts away from many important aspects. For instance, occlusions are hard to reason about on a purely symbolic level although the visibility of objects is a hard constraint for being able to manipulate an object.

The variation in pick-and-place actions is minimal, given a robust implementation of the different sub-actions and the integration of a powerful reasoning engine which allows to avoid symbolic planning at all in many cases. CRAM provides a physics-based reasoning engine that integrates a geometrically accurate representation of the world to allow for inferring plan parameters based on the current state of the world, visibility, stability and reachability.

In this section, we will describe extensions to CRAM's reasoning engine that integrate geometric aspects of the environment. Although these extensions allow for finding objects that are reachable for the robot from its current location or for finding all visible objects, these extensions do not provide the functionality to generate solutions for poses that fulfill certain such properties. However, an extension to location designators as presented in Section 2.3 will be shown in Chapter 4.

The idea behind the implementation of CRAM's geometric reasoning engine is to couple CRAM's symbolic reasoning engine with a geometrically accurate world database and compute the truth values and bindings of predicates on demand, whenever the reasoning engine needs to prove a specific predicate. When performing pick-and-place

tasks in changing environments, the most important aspects to take into account in order to infer plan parameters are stability, reachability and visibility.

Stability basically means that the robot and objects should not be placed at locations where they would be colliding with other objects and where they are standing stable, i.e. not flipping over. Reasoning about stability is not only important when placing objects on supporting planes such as table tops or drawers but also when stacking objects. For instance, being able to infer which objects can be stacked and in which order (e.g. cups can be stacked on plates but not the other way around) opens up many opportunities for plan optimization.

Since robots in our domain are mainly performing manipulation tasks, reasoning about reachability is clearly another important feature that needs to be supported. The robot needs to be able to infer if an object is reachable when it is standing at a specific location and it needs to be able to infer if grasping the object is possible at all or if the object is blocked by other objects standing in the way. While motion planning [Latombe, 1991] solves the problem of deciding if and how an object is reachable and grasp planning [Saxena et al., 2008, Hsiao and Lozano-Pérez, 2006] allows to find stable grasps for picking up an object, both operations are computationally extremely expensive. However, as one of the fundamental applications of CRAM is its use in robot executives, decision making needs to be fast. This is one reason for not implementing or integrating full motion and grasp planners. Instead, the system relies on simple heuristics and inverse kinematics calculation for only a few key points to allow for fast decisions although sacrificing correctness. However, this is normally not a serious problem because the robot's execution uses a motion planner or a similar component which will signal an error that is handled by the executive in case an object really is not reachable. In fact, the CRAM reasoning system tries to find solutions for which it is likely that a motion planner will find a valid solution quickly.

Reasoning about visibility is probably one of the most important features CRAM provides. In order to detect objects, the robot needs to be able to see them. When placing objects the robot needs to be able to infer if an object will occlude other objects that are needed later in a plan. The implementation in CRAM's reasoning component is based on off-screen rendering. The basic idea is to render a scene from the robot's perspective and check if an object is visible, how much of it is visible and which objects are occluding it. The rendering engine accesses the internal representation of the

robot's environment to get 3D models of objects and their position in order to render an exact representation of the robot's belief about the world.

The system integrates the Bullet physics engine [1] to implement predicates such as *stable* or *contact* for stability reasoning and to store a geometrically and physically accurate representation of the robot's environment. For inferring facts about visibility, the system uses OpenGL to render the world database and to reason about reachability, the system uses standard inverse kinematics (IK) solvers provided by many robot software packages including ROS.

Predicates to interact with CRAM's world database	
(world ?w)	Unifies *?w* with a world database.
(assert ?w (object ?t ?n ?p . ?_))	Asserts an object of type *?t* and name *?n* at pose *?p*.
(assert ?w (object–pose ?n ?p)	Moves the object named *?n* to pose *?p*.
(retract ?w (object ?o))	Retracts an object *?o* from the world database *?w*.
(object ?w ?n)	Asserts that the object with name *?n* exists in world *?w*.
(object–pose ?w ?n ?p)	Unifies *?p* with the pose of object named *?n* in world *?w*.
(object–type ?w ?n ?t)	Unifies *?t* with the type of the object named *?n*.
Predicates to interact with the robot model	
(link–pose ?w ?r ?l ?p)	Unifies *?p* with the pose of the robot link named *?l* of the robot model named *?r*.
Predicates to reason about stability	
(contact ?w ?o1 ?o2)	Holds if objects *?o1* and *?o2* are in contact in world *?w*.
(stable ?w ?o)	Holds for all objects *?o* that are stable in world *?w*.
(stable ?w)	Holds if all objects are stable in world *?w*.
(above ?w ?o1 ?o2)	Holds if the object named *?o1* is above the object named *?o2* in world *?w*.

[1] http://bulletphysics.org/

(below ?w ?o1 ?o2)	Holds if the object named *?o1* is below of the object named *?o2* in world *?w*.
(supported–by ?w ?t ?b)	Holds if the object named *?t* is supported by the object named *?b*.
Predicates to reason about visibility	
(visible–from ?w ?p ?o)	Holds if the object *?o* is visible from pose *?p*.
(visible ?w ?r ?o)	Holds if the object *?o* is visible in one of the cameras of the robot object named *?r*.
(occluding–objects ?w ?p ?o ?occ)	Unifies the list of object names that are occluding the object named *?o* with *?occ*. *?p* is either the name of a robot object or a specific pose from which the scene is observed.
(occluding–object ?w ?p ?o ?occ)	Holds for all objects *?occ* that are occluding the object named *?o*.
Predicates to reason about reachability	
(reachable ?w ?r ?o)	Holds if the robot *?r* can reach object *?o*.
(reachable ?w ?r ?o ?m)	Holds if the robot *?r* can reach object *?o* with all manipulators specified by the list *?m*.
(blocking ?w ?r ?o ?b)	Holds for each object *?b* that is blocking the trajectory to grasp object *?o*.
(blocking ?w ?r ?o ?m ?b)	Holds for each object *?b* that is blocking the trajectory to grasp object *?o* with manipulator *?m*.

Table 3.1: *Predicates used to perform physics-based inferences*

In the remainder of this section, we will explain the predicates for reasoning in an accurate three-dimensional representation of the world and how they are implemented. A list of basic predicates can be found in Table 3.1.

3.1.1 Physics Engine Integration and the World Database

To implement the Prolog world database and predicates that reason about physics, the CRAM reasoning engine integrates the Bullet physics engine. Bullet is an industry strength, widely used and well maintained physics engine with its main application in computer games. New simulation environments for personal robotics applications, for instance Morse[2] and Gazebo[3] use it already or plan to use it soon. However, game physics engines also have limitations, mostly caused by their original application domain, computer games where accuracy was always less important than nicely looking effects and good performance. Bullet, like other game engines such as ODE[4], has problems with simulating closed kinematic loops that happen, for instance, when grasping objects. Also, the simulation of small objects is relatively unstable. Fortunately, in the CRAM reasoning system, dynamics simulation of actual grasps is not really required and the instability problem can be overcome by scaling the complete world. Essentially, it is more important for the CRAM reasoning engine to predict *that* an object will flip over than where exactly it will fall. The authors in [Weitnauer et al., 2010] show that parameter such as friction can be tuned to improve simulation accuracy though.

The implementation of CRAM's physics reasoning engine is split up into three layers of abstraction:

1. A wrapper library for Bullet in Common Lisp. The library consists of a C++ part that wraps all classes and methods required from Bullet into a C library that is linked into Common Lisp using SBCL's foreign function interface. Additionally, a Lisp library is implemented to provide a native look and feel of the wrapped Bullet library. It defines Lisp classes and methods similar to Bullet's classes and integrates with Lisp's garbage collector.

2. A Common Lisp library that adds more semantics to the Bullet library. This includes different object classes that contain more meta data than Bullet requires, for instance names, collections of rigid bodies to represent whole objects, the grounding of objects in Knowrob's semantic map, etc. Additionally many utility

[2] http://www.openrobots.org/wiki/morse/
[3] http://gazebosim.org/
[4] http://www.ode.org/

functions are provided for instance for loading objects from other ROS components and files.

3. Predicates that provide the interface for spatial reasoning and reasoning about physics. These predicates access the functions, methods and classes of layer 2.

The physics engine. For dynamic simulation of rigid objects, a physics engine needs to provide data structures for storing rigid bodies, their parameters required for simulating dynamics and their current dynamics state. In each simulation step, these data structures are updated according to the rules of the physics engine.

Since reasoning about visibility, reachability and other geometric properties requires at least some of the object properties which are also required by the physics engine such as the position of the object or its three dimensional model, CRAM uses and extends Bullet's data structures directly as a world database.

Besides classes for collision checking and a constraint solver to compute how rigid bodies that are connected to each other through joints behave, a Bullet world class requires a gravity vector. It provides methods for adding new rigid bodies and for removing them and for performing one simulation step. A rigid body is essentially a data structure that represents an object. The rigid body data structure contains the object's motion state, i.e. the object's position in the world. Additionally, the a rigid body object contains the mass and the inertia matrix of the object required for dynamics simulation, the object's current linear and angular velocities and a list of collision shapes. Additionally, Bullet's rigid body class provides interfaces for changing the object's simulation and collision behavior, i.e. it allows for disabling or enabling simulation of the object and to configure which objects can collide with which other objects. For collision checking, each rigid body must provide a collision shape. The following (incomplete) list shows the most important collision shapes supported by Bullet. They are all available in CRAM's physics-based reasoning engine:

- *Box*: this shape represents a box defined by a length, width and height.
- *StaticPlane*: a static plane of infinite size. It is defined by a normal and a constant representing the plane's distance from the origin.
- *Sphere*: a sphere, defined by specifying a radius.

- *Cylinder*: a cyllinder defined by the width, length and height of its bounding box.
- *Cone*: a cone defined by a radius and a height.
- *CompoundShape*: compound shapes allow to combine different collision shapes to one. That way it is possible to construct complex collision shapes for rigid bodies that consist of several shapes. The positions of the child shapes relative to their rigid body's pose are fixed.
- *ConvexHull*: a convex hull shape is a set of points that define the convex hull of the object. Usually, the points are the vertices of a three-dimensional mesh.

Additionally, Bullet provides support for constraints, i.e. joints to connect several rigid bodies. Although joints are important for simulating the dynamics of a robot since they essentially represent the connection between two parts of the robot and the point where motor forces are applied, they are not really important for most reasoning tasks in CRAM.

The representation of objects. While CRAM's world representation includes Bullet's complete world representation, it extends the Bullet data structures with the concept of objects. Objects are sets of rigid bodies that semantically belong to each other. For instance, the different parts of a robot (i.e. the robot's links) compose the object "robot". They need to be positioned independent of each other. Since this is not possible by using compound collision shapes, multiple rigid bodies must be used.

Class 12 The definition of the class `object` that is the base class for all objects in CRAM's reasoning world.

class OBJECT

slot NAME:	A (unique) symbol naming the object.
slot RIGID-BODIES:	A hash table that maps rigid body names to instances of Bullet's rigid body class.
slot POSE-REFERENCE-BODY:	The name of the object's main body. It is used to determine the pose of the object. Per default, this rigid body is used by the object's `pose` method.
slot WORLD:	A pointer to the world instance this object belongs to.

The definition of the class object is shown in Class 12. It is the base class for more complex objects required for the reasoning tasks that come up when executing pick-and-place actions on robots, for instance semantic maps [Tenorth et al., 2010a]. In CRAM's reasoning engine, semantic maps are classes that are derived from the class object and extend it with a reference to the semantic map as it is stored in OWL [Bechhofer et al., 2004] and provided by KNOWROB. The reasoning system additionally provides support for articulated objects, i.e. objects that have, for instance, doors or are drawers, and can be opened and closed. The class robot−model is another class derived from object that represents robots. A robot consists of links and joints that are normally defined outside the reasoning system. In ROS, they are usually defined using URDF[5], an XML specification with information about the robot's kinematic chain, sensors and links, including their 3D models and inertia. Links essentially correspond to rigid bodies in Bullet and joints connect these links. However, since the CRAM reasoning system does not require dynamic simulation of the robots, joints specified in URDF files are not added to the corresponding Bullet representation but are just used for calculating forward kinematics and for positioning the links. Although the robot model collides with other objects, it is essentially represented as a static object that is not simulated by the physics engine.

Grasping and releasing objects is the most important action in pick-and-place tasks. Since in particular game physics engines often have problems simulating kinematic chains and since we do not perform any dynamic simulation of the robot model in the CRAM reasoning engine, the robot−model has explicit support for attaching objects. When we want to express that an object is grasped by the robot, we explicitly connect it to a specific link of the robot model object. This has two effects:

- The system disables physics simulation for the object to prevent it from.
- Whenever the link the object is connected to moves, the object is moved accordingly.

The system defines an additional object class that should be used for household objects such as knives, forks, cups and so on. These classes represent additional semantic information that is needed for our reasoning tasks. For instance, in general, collisions between objects and in particular between the robot and other objects, are not de-

[5]http://ros.org/wiki/urdf

sirable. An exception are household objects that are either standing on objects of the semantic map or stacked on each other. Additionally, objects such as a cup can have additional semantic properties defined by an object hierarchy in a knowledge base. For instance a cup also is of type *container*.

Storing and restoring the world. When using the CRAM's world representation for reasoning, it is important to not change the global world database to avoid destroying the current representation of the world. Additionally, we want to be able to make snapshots of the world to analyze previous states of the world and integrate them in reasoning. To implement these features, the system provides mechanisms for storing the complete static and dynamic state of the world and later restoring it. When storing the world database, for each object the list of all its rigid bodies, their positions and their linear and angular velocities is stored. Additionally, the reference to the bodies' collision shape, the mass and inertia of the object and its collision properties are stored. In Bullet, collision shapes cannot change once instantiated. It is therefore allowed to reuse collision shapes in several rigid bodies and different world instances. Since collision shapes can be reused and are not serialized when storing a world database, this operation is relatively fast and can be used in the reasoning engine without a significant negative impact on the overall performance of the Prolog engine. All predicates for physics-based reasoning require an explicit world database instance bound to a variable and perform simulation in a copy of that world database.

3.1.1.1 Prolog predicates for physics-based reasoning

The main interface for the user to interact with the Bullet based world database are predicates implemented in CRAM's Prolog interpreter. To allow for reasoning in different world databases in the same Prolog code, all predicates that use information from a world database have an explicit world variable. However, the system has a default world database that is used when a predicate's world variable is unbound. Similar to Prolog's fact database, the physics-based reasoning engine provides `assert` and `retract` to add, update and remove objects or change their position. For instance, to create a cup at a specific location, we can evaluate the following Prolog expression:

```
(assert ?w (object mesh cup ((-1.8 2.0 0.912) (0 0 0 1))
                  :mesh mug :mass 0.2))
```

The expression looks rather complex, but to add objects, the system needs at least to know where to put the object, what type it is, how the object should be named, its mass and some additional information about the object type. In the example above, we add a new mesh with name *cup*. The mesh is indicated by the symbol *mug* and the mass is set to 200 grams.

The signature of the `assert` predicate for changing the world database is as follows:

```
(assert ?world ?fact)
```

Currently, `?fact` can be bound to a number of different patterns that allow for adding objects, changing object poses, changing joint states of robot objects and attaching objects to links on the robot.

Adding and changing objects. The most important interaction with the environment is the ability to add objects and update their location. For instance, integration of perception makes extensive use of the assertion of new objects or updates their location if necessary. The corresponding fact for asserting new objects is:

```
(assert ?w (object ?type ?name ?pose . ?args))
```

Note that if `?w` is unbound, the default world database is used. `?type` must be bound to a symbol naming the type of the object to be added. The supported object types and the required additional arguments are shown in Table 3.2. The `?name` variable is mandatory and must be a symbol specifying the object's name in the database. This

name must be unique since it is used to reference the object in predicates. The variable ?pose is also a mandatory parameter that can either be a list of the form ((x y z) (ax ay az aw)), i.e. a vector and a quaternion, or a specific Common Lisp object representing a pose. ?args are type specific parameters such as the object's mass or the extents of for instance a box.

Type	**Parameters**	**Description**
box	mass, size	Box with specified *mass* and 3D vector *size*.
static-plane	normal, constant	Static plane according to the specified parameters *normal* and *constant*.
sphere	mass, radius	Sphere with the given *mass* and *radius*.
cylinder	mass, size	A cylinder with the given *mass* and bounding box specified by *size*.
cone	mass, radius, height	A cone with the given *mass*, *radius* and *height*.
point-cloud	points	A static point cloud constructed from *points*.
mesh	mass, mesh, color, disable-face-culling	An object created from a mesh identified by *mesh*, *mass* and *color*. The mesh identifier can either be a URI or a symbol for a mesh alias, e.g. mug, plate or pot. *disable-face-culling* can be used to fix broken face winding order.
cutlery	mass, color, cutlery-type	Add cutlery of type *knife* or *fork*, the specified *mass* and *color*.
urdf	urdf, color	Add a URDF model (i.e. a robot) from a parsed robot description file bound to *urdf*.

semantic-map	urdf	Add a semantic map. If specified, *urdf* is used for visualization, but the links need to be grounded in KNOWROB's semantic map. If not specified, all semantic map objects are just visualized as boxes.
ros-household-object	mass, model, color	Adds an object from the ROS household object database[6]. The model indicated by *name* is added with *color* and *mass*

Table 3.2: *Currently supported object types and their parameters that can be asserted in the* CRAM *reasoing world database.*

As can be seen in Table 3.2, some objects require a mass parameter and some do not. Objects without a mass parameter or objects for which the mass is set to zero are static objects, i.e. no dynamic simulation is performed. They are only collision objects. Although URDF files might contain inertia and mass information for some parts of a robot, these parameters are ignored. The reason is that the goal of the system is not to provide a complete dynamic simulation of the robot but to allow for light-weight reasoning.

Once an object has been added to the world database, the only allowed interaction with it is changing its pose. In other words, an object cannot change its type after it has been created. The pose of an object can be updated as follows:

```
(assert ?w (object-pose ?name ?pose))
```

The variable ?name has to be bound to the name of the object as specified when the object was asserted and ?pose is the object's new pose. It can be either a list with lists

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

containing the coordinates and the rotation of the object or a pose object, similar to the assertion of objects.

To remove objects from the world database, the system provides a `retract` predicate that acts on world databases. The signature for removing objects is:

```
(retract ?w (object ?name))
```

After the retraction form has been evaluated, the object matching the specified name will be removed from the specified world database.

Updating a robot object. Robot objects are somehow special because the robot is a central concept in reasoning about actions, visibility and reachability. Although the robot object inherits all properties from the class *object*, it extends it by the concept of links and by support for attaching and detaching objects.

Specifically, to update the robot's state in a world database from the position of the real robot's joints, the system provides a special assertion form:

```
(assert ?world (joint-state ?robot-name ?joint-states))
```

The variable `?robot-name` needs to be a symbol that names a URDF object. The variable `?joint-names` needs to be a list with tuples containing the name of the joint to set and the position as a number. For instance, to set the joint with name *"torso_lift_joint"* of the robot object *pr2*, the following assert expression can be used:

```
(assert ?w (joint-state pr2 (("torso_lift_joint" 0.33))))
```

Dealing with manipulation is one of the most important aspects in the reasoning system. Since the system is not simulating the robot itself but only uses physics simulation to prove specific aspects and because physics engines are relatively bad in particular at handling grasping, the CRAM reasoning system provides explicit support for attaching and detaching objects to the robot's links. Whenever a robot link moves, all attached objects move accordingly. To attach an object in a specific world database, the following assertion form can be used:

```
(assert ?w (attached ?robot ?link-name ?object))
```

Similar to joint state assertions, the variable ?robot has to be bound to the name of a robot object. ?link-name specifies the name of the link the object should be attached to as defined in the URDF file corresponding to the object. Finally, the variable ?object has to be bound to a symbol naming the object that is to be attached.

To detach an object from the robot, the reasoning system provides a retraction for the attached relation respectively. Its signature is as follows:

```
(retract ?w (attached ?robot ?object)
```

The effect is that the object instance bound to ?object is detached from all links of the robot object ?robot. If ?object is unbound, all objects will be detached. To detach a specific object from a specific link or to detach all objects from a specific link, a second version is provided:

```
(retract ?w (attached ?robot ?link-name ?object))
```

If ?object is unbound and ?link–name is bound, all objects connected to the link are detached.

Querying the world database. Often, it is required to enumerate all objects and iterate over them, to verify if an object for a given identifier exists, what the type of an object is, if it is a household object, i.e. an object that can be manipulated or what its pose is. The CRAM reasoning system provides a number of predicates to query information about objects in the database.

The predicate object holds for each object name in the database. It allows for querying all objects and for checking if a given name references an existing object in a world database. Its signature is as follows:

```
(object ?world ?name)
```

To assert or query the type of an object or all pairs of object names and the corresponding type, CRAM provides the predicate object–type with the following signature:

```
(object-type ?world ?name ?type)
```

Please note that an object can have several types. One solution for an object type is always the CLOS class of the object, for instance *robot-object* or *household-object*. In addition to being a *household-object*, a mug also has type *mug*. For normal household objects such as cutlery, the additional types are defined in the methods that are responsible for adding the object when the (assert ?w (object ...)) predicate is used. Additionally, the system supports creating objects from ROS' household objects database. Besides the 3D meshes, the database also contains semantic tags describing the different types of household objects. The object–type predicates uses these if possible to generate additional solutions.

Specifically for getting all types of a household object, CRAM provides the predicate household–object–type with the following signature (similar to object–type):

```
(household-object-type ?world ?name ?type)
```

The predicate only holds for objects that are sub-classes of *household-object*.

For accessing the pose of an object, the predicate object–pose is provided (similar to the corresponding assertion). The signature is as follows:

```
(object-pose ?world ?name ?pose)
```

The predicate unifies the pose of the object's reference body with the variable ?pose.

To check if a robot object has a specific link or to enumerate all links of a robot object, the predicate link with the following signature is defined:

```
(link ?world ?robot-object ?link)
```

Finally, for getting the pose of a specific link of a robot object, the system provides the predicate link–pose with the following signature:

```
(link-pose ?world ?robot-object ?link ?link-pose)
```

This predicate is in particular useful for querying the pose of sensor frames, as required for instance for visibility reasoning.

Basic predicates for physics-based reasoning. The main predicates for physics-based reasoning are predicates for querying contacts between objects and for inferring stability. Contacts are proven by using Bullet's collision detection engine. Bullet provides classes and methods for querying so-called collision manifolds, i.e. data structures that contain the contact points of two colliding objects. The predicate for representing contacts between two objects in CRAM's reasoning engine is `contact` with the following signature:

```
(contact ?world ?object-1 ?object-2)
```

It holds for all colliding objects in the world database `?world`. Since some objects and most importantly the robot and the environment based on a semantic map consist of many links, a second version of the `contact` predicate is provided that allows for reasoning about collisions between two objects with a specific link name. The link can be either part of the first object or of the second object. Its signature is as follows:

```
(contact ?world ?object-1 ?object-2 ?link)
```

If both objects have several links one solutions for each link is generated. To explicitly state that only the links of one object should be considered, the predicate `link` can be used.

As mentioned already, the implementation of the predicate is based on *collision manifolds* provided by Bullet's physics engine. Depending on the value of the two object variables and optionally the link, either all contacts, all contacts of a given object or only the contacts between two specific objects are taken into account. More specifically, if both objects are unbound, the list of all manifolds, i.e. of all contacting rigid bodies is used and reduced to the actual object instances that are colliding. Then, solutions for all contacting objects are generated. If only one object variable is bound, only contact manifolds that contain rigid bodies of this specific object are used. If both

object variables are bound, the system simply verifies if the two objects are colliding with each other. If the second version with a link is used, additionally solutions for all contacting links (which are represented as rigid bodies internally) are generated. Otherwise, two objects are considered to be in collision if any of their rigid bodies collide.

The predicate stable is based on comparing the pose of a single object in two world databases where one world database is the original database and the other one a copy of it after simulating the world for a specific number of seconds (currently 5). If the difference in the pose of an object is above a specific threshold, it is considered unstable. The signature of the stable predicate is as follows:

```
(stable ?world ?object)
```

An alternative implementation of the stable predicate with the following signature for asserting that the complete world, i.e. all objects, are stable is provided:

```
(stable ?world)
```

The implementation uses the predicate object—pose—different that holds if the pose of an object is different in two world databases. Its signature is as follows:

```
(object-pose-different ?world-1 ?world-2 ?object)
```

The stable predicate for a specific object first copies the world database. Then it calls the Bullet function simulate on it to advance the world for five seconds. Finally, the pose of the object in the original world and the simulated world are compared. Listing 3.1 shows the implementation of this version of the stable predicate.

Listing 3.1: *The implementation of the stable predicate for a specific object.*

```
1 (<- (stable ?world ?object)
2   (world ?world)
3   (copied-world ?world ?copy)
4   (simulate ?copy 5)
5   (object ?world ?object)
6   (not (object-pose-different ?world ?copy ?object)))
```

As can be seen, first a valid world database is asserted, followed by a predicate that creates an exact copy of the world database by serializing it and deserializing the result into a new database object. The next step is to simulate the copied world. The predicate object holds for every object. If the variable ?object is unbound, it creates a choice point for each object in the world to iterate over all objects. If it is bound, the predicate object only holds if the object actually exists in the world database. The final step is to verify that the object did not move during simulation.

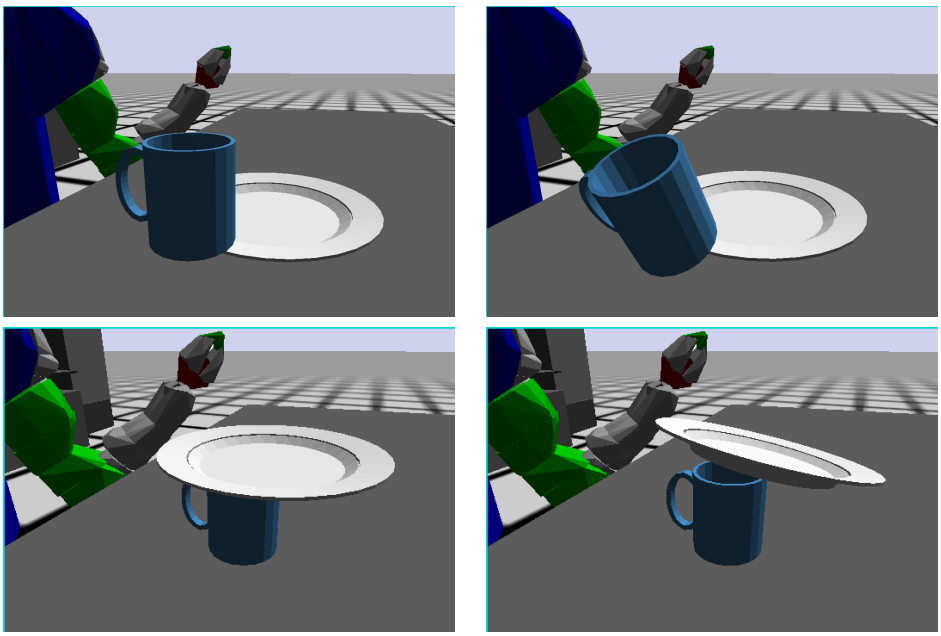


Figure 3.1: *Unstable scenes in the initial configuration (left) and after simulating for 0.5 seconds. In the top row, a mug is standing on the edge of a plate, in the bottom row, the plate has been placed on a mug.*

Figure 3.1 shows a visualization of the world database before and after simulation of two unstable scenes.

Higher-level physics-based predicates. Based on the predicates defined in the previous paragraph, some higher-level predicates for reasoning about simple spatial relations and the *supporting* relation are defined. The supported–by predicate holds for all pairs of objects where object A is standing stable on another object A, i.e. where object A is supported by object B. The predicate supported–by is defined as follows:

```
1 (<- (supported-by ?world ?top ?bottom ?link)
2   (world ?world)
3   (contact ?world ?top ?bottom ?link)
4   (above ?world ?top ?bottom ?link)
5   (stable ?world ?top))
```

As can be seen, the relation holds if two objects are in contact with each other and one is above the other one. Additionally, the supported object needs to be stable.

The system also provides the predicates above and below that hold if the bounding boxes of two objects overlap and the z value of one object is greater than the z value of the other object or smaller respectively. The signatures of the predicates are as follows:

```
(above ?world ?object-1 ?object-2)

(below ?world ?object-1 ?object-2)
```

Please note that the predicate below is just implemented using the above predicate. If object-1 is above object-2, then object-2 must be below object-1.

Various helper and debugging predicates. For debugging purposes, the CRAM reasoning system defines several predicates to visualize and simulate the world.

- (debug–window ?world): creates a new GLUT[7] OpenGL window that visualizes the world database bound to the variable ?world. The user can navigate in the world using the mouse. Most of the screenshots showing the system are created using the debug–window predicate.
- (simulate ?world ?t): calls Bullet and simulates the world for the specified amount of time (in seconds). Note that this predicate changes the world database by updating object positions according to the physics engine, i.e. this predicate is not side-effect free.
- (step ?world ?dt): performs one simulation step with length ?dt in seconds. This predicate is not side effect free either.
- (simulate–realtime ?world ?dt): Simulates the world and slows down simulation to achieve simulation in realtime. Realtime here means that one second simulation time corresponds to exactly one second real time. In contrast, simulate simulates as fast as possible. The main purpose of this predicate is debugging.

3.1.2 Visibility Computation

Reasoning about visibility and occlusions is probably one of the most important and still least addressed aspects in planning and action execution. Although it might be possible to reason about stability on a purely symbolic level by defining naive physics rules, reasoning about visibility without taking into account the geometry of objects and the environment is hard at best. One reason is that not all objects that are occluded or that are occluding other objects are not necessarily specified explicitly in actions. For instance, an object that is placed on the table by the robot might occlude other objects that were not directly involved in the put-down action. This is one variation of the frame problem [McCarthy and Hayes, 1969]. Another reason is that not only the camera parameters and the location of the camera but also the 3D model and properties of the algorithms that are used for detecting objects influence the overall

[7] http://www.opengl.org/resources/libraries/glut/

success of object detection. For instance, some algorithms might be able to detect an object if only 80% of it are visible while others need a complete view of the object, maybe even without any other objects too close. Figure 3.2 shows an example scene where only the mug and the pot are visible for the robot but not the bowl.

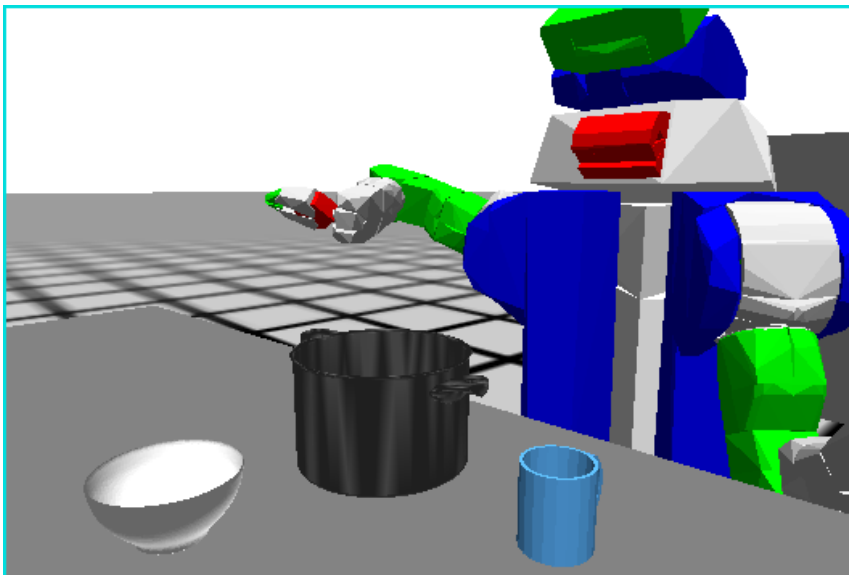


Figure 3.2: *Example scene with a table, the robot and tree objects: a bowl (white), a mug (blue) and pot (black)*

The CRAM reasoning system provides two predicates for reasoning about visibility: visible and occluding. The implementation of both predicates is based on OpenGL. The basic idea is to render all objects in the world database with each object in a different color. The viewpoint is set to one of the robot's sensors. By counting pixels and comparing the number of pixels that were seen and the number of pixels that should belong to the object, we can compute the percentage of an object's visibility.

Computing the visibility of an object. The underlying functionality for implementing all predicates for visibility reasoning is provided by a single Lisp function, calculate-object-visibility. The function requires the world database, the object of interest, the camera position, and some camera parameters such as its size and field of view as input and returns how much of the object is visible as a percentage as well as the list of occluding objects. To work properly, the algorithm needs three rendering runs:

1. Render the object with the camera centered on it to calculate how many pixels of the object should be visible.

2. Render only the object from the actual camera pose to calculate a *mask* of the object in the camera image.

3. Render the complete scene from the actual camera pose.

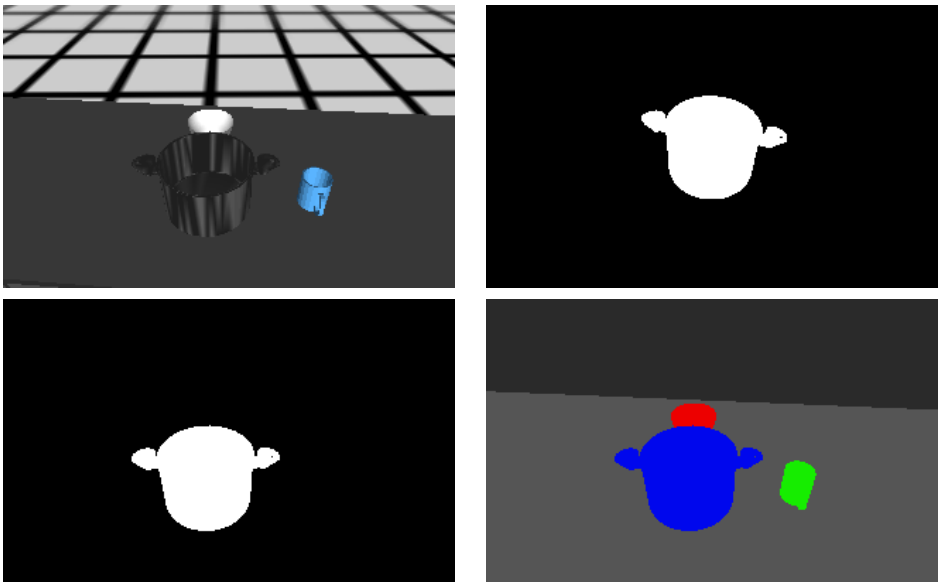


Figure 3.3: *Generated images to infer the visibility (and occluding objects) of the pot. Top left: rendered scene from robot perspective; top right: the pot centered; bottom left: only the pot; bottom right: the complete scene, every object has a unique color. The pot is (partly) occluding the bowl.*

The result of these rendering runs is shown in Figure 3.3. The first rendering run is required to know how much of the object should be visible. For instance, if the object is standing behind the robot, i.e. if it is not visible in the camera sensor at all, just rendering the object with the actual camera position would lead to zero pixels visible but still to 100% visibility if we didn't calculate how many pixels should be visible by pointing the camera directly on the object.

The second rendering run generates a binary mask where pixels that belong to the object are set to one and the other pixels to zero. All pixels that are inside this mask but that do not belong to the object belong to objects that are occluding the object of interest.

The third rendering run draws each object in a different color with lighting disabled. By counting pixels inside the mask of the second rendering run that belong to the

object of interest, the system can compute the percentage of the object's visibility. All pixels that belong to other objects and are inside the mask are occluding the objects and are returned in the list of occluding objects.

Predicates for visibility reasoning. The CRAM reasoning framework provides four predicates for reasoning about visibility, visible-from, visible, occluding-object and occluding-objects. The signature of the visible–from predicate is as follows:

```
(visible-from ?world ?camera-pose ?object)
```

The predicate holds when the percentage of visible pixels is above a certain threshold when the scene is seen from a specific pose, the camera frame. Although this threshold is depending on the sensor and the algorithms used on a real robot, we consider it sufficient in most cases to use a fixed threshold of 90%. Similar to most other predicates, if the variable ?world is unbound, the default world database is used. The variable ?camera–pose must be bound since the system is unable to iterate and backtrack over all possible (6D) poses due to exploding computational complexity. The variable ?object can be left unbound. If it is bound, the predicate just verifies if an object is visible from a specific pose. If unbound, the predicates generates a choice point in the Prolog prove tree with solutions for every visible object. However. in the current system, enumerating all visible objects is rather expensive since for each object, the three rendering runs are required. By using a more specialized implementation of this variant of the predicate, the computational complexity can be reduced though. To just compute all visible objects, it is sufficient to execute the first rendering run for each possible object, leave out rendering run two and execute the third rendering run only once.

Often, the user just wants to query if an object is visible by the robot or which objects are visible without explicitly specifying the camera pose. The CRAM reasoning system provides the visible predicate for that case with the following signature:

```
(visible ?world ?robot-object ?object)
```

It relies on an additional predicate camera-frame which needs to be implemented in the robot's knowledge base that allows to query the names of the robot's sensor frames. The visible predicate is implemented as follows:

```
1 (<- (visible ?world ?robot ?object)
2   (world ?world)
3   (robot ?robot)
4   (camera-frame ?robot ?camera-frame)
5   (link-pose ?world ?robot ?camera-frame ?camera-pose)
6   (visible-from ?world ?camera-pose ?object))
```

As can be seen, the predicate camera-frame is used to find the name of all camera frames. Then the current pose of a camera frame is queried by using the link-pose predicate. Finally, the system uses the visible-from predicate to assert visibility of an object.

The predicate occluding-objects allows to infer the list of objects that are occluding a specific object. The signature of the predicate is as follows:

```
(occluding-objects ?world ?camera-pose ?obj-name ?
    occluding-names)
```

As in most other predicates that access a world database, the variable ?world can be either bound or unbound with the default database being used in case it is unbound. ?camera-pose must be bound. The variable ?obj-name can be unbound which generates solutions for each object. If bound, the variable ?occluding-names must be the exact list of objects that are occluding the object referenced by ?obj-name.

The predicate occluding–objects is polymorph over the second variable. If it is not bound to a Common Lisp pose object, it must be a symbol naming a robot object. In that case, similar to the visible predicate, the predicate camera–frame is used to find all sensor frames of a robot object and find the occluding objects for each of the frames.

To check if a single object is occluding another object, the CRAM reasoning system additionally provides the predicate occluding–object with the following signature:

```
(occluding-object ?world ?camera-or-robot ?obj ?occluding-obj)
```

Please note that in contrast to occluding–objects, the predicate occluding–object will fail if there are no occluding objects while occluding–objects will yield an empty list. That means if the user wants to verify that there are no occluding objects, she can either write

```
(not (occluding-object ?world pr2 mug ?o))
```

or assert that the list of occluding objects is empty:

```
(occluding-objects ?world pr2 mug ())
```

3.1.3 Reachability Reasoning

Besides stability and visibility, the third important aspect when performing actions in an unstructured and complex environment is reasoning about reachability. More

specifically, that means that the robot needs to be able to evaluate locations and verify if objects are reachable, infer which objects are reachable and if other objects might be in the way when reaching for an object. Motion planning [Latombe, 1991] and grasp planning [Berenson et al., 2007] is a well studied area in mobile robotics. However, while motion planning algorithms are able to generate collision free paths, the problem is hard and computationally extremely expensive. Since one goal of the CRAM reasoning system is to provide a fast system to be used for decision making at runtime, finding solutions that lead to high success probabilities of a plan is sufficient. More specifically, the CRAM reasoning system is used to generate solutions for plan parameters such as locations for the robot base to stand while executing an action for which it is likely that grasp and motion planning that is used when actually executing an action will succeed. If planning really happens to fail, the system is able to backtrack and generate a new solution for which motion planning is retried.

To improve performance, we make several assumptions in the CRAM reasoning framework:

1. Avoid grasp planning by defining a fixed set of object specific grasps. For instance, for grasping a mug, side, front and top grasps are used. For grasping a plate, only side grasps are valid.

2. Avoid motion planning by approximating planned trajectories by a low number of trajectory key points.

3. Instead of searching for completely collision-free solutions, rely on motion planners that are used to execute actual grasping actions. With state of the art sensors, motion planners are robust enough to find a collision free path to reach an object although the link configurations corresponding to the trajectory key points used in reachability reasoning are in collision with objects.

The CRAM reasoning system implements two predicates for reasoning about reachability, `reachable` and `blocking`. Both are based on using inverse kinematics calculation to find joint angles to reach for an object. The exact pose to reach is defined by the center of the object's bounding box, a grasp which is essentially the orientation of the gripper and a tool vector.

Valid grasps for reasoning. The CRAM reasoning system relies on a number of predefined standard grasps for reasoning about reachability. Many household objects are rotationally symmetric or almost rotationally symmetric. Examples include glasses, cups and plates. This property implies that many grasps are similar and equally good. On the other hand, the set of valid grasps highly depends on the type of the object. For instance while the robot might be able to grasp a cup from the front, a side or from the top and only one arm is needed, for plates the robot needs to use side grasps and two arms. To be as general as possible, the system uses a knowledge base where information about the grasps that should be used for reasoning and the grasps that are valid for picking up an object are defined by predicates. A grasp is a full pose defined by two components, the grasp orientation and the tool vector defining a direction and the distance between the robot's gripper frame and the pose to reach. In the current system, the actual grasps are not object specific, i.e. the system always uses a subset of initially defined grasps for calculating poses to reach an object.

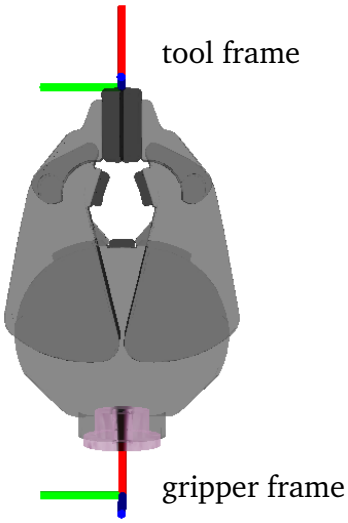


Figure 3.4: *The default tool frame relative to the PR2's gripper frame.*

For objects that are smaller than the robot's gripper, the robot always reaches for the object's center, using a so-called tool. The tool length is adjusted if the bounding box of the object is bigger than twice the default tool length to account for the fact that only the robot's gripper frames should penetrate the object. More specifically, given the center of an object's bounding box defined by the pose matrix O, the actual pose

P to reach is calculated based on the grasp orientation matrix G and the tool vector $\vec{t}$. It is defined as follows:

$$P = O(G\vec{t})^{-1} \tag{3.1}$$

The definition of grasps and the tool length is highly robot specific. The tool vector is constant, i.e. it is used for all grasp orientations. Figure 3.4 shows the default tool frame with a tool vector of length 0.2 and no rotation relative to the PR2's gripper base frame. For the PR2, we define four grasps, two side grasps, one front grasp and one top grasp. The side grasp to use is selected based on the arm the reasoning system is generating solutions for.

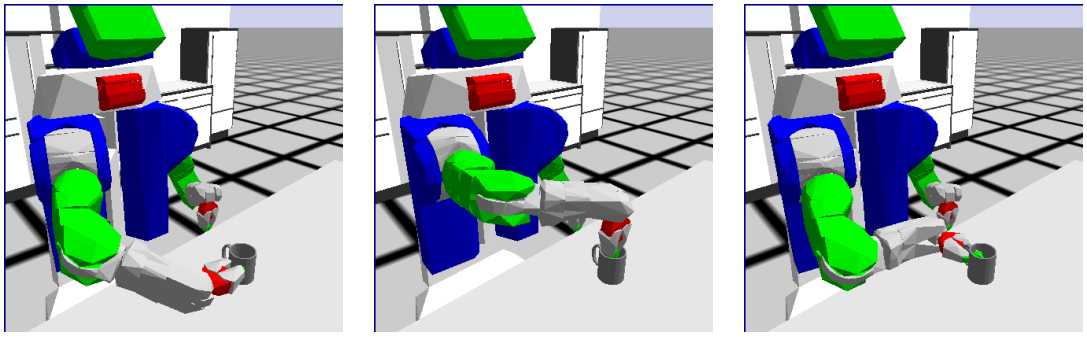


Figure 3.5: *Side and top grasp and front grasp used for reasoning about reachability.*

Figure 3.5 shows the three grasps that are allowed for grasping a cup with the right arm. The corresponding orientations in the robot's gripper frame are defined as follows:

- Top grasp: rotation of -90 degrees around the Y axis.
- Side grasp left: rotation of 90 degrees around the Z axis.
- Side grasp right: rotation of -90 degrees around the Z axis.
- Front grasp: no rotation.

The default tool vector that is used for small objects is:

$$\vec{t} = \begin{pmatrix} 0.2 \\ 0.0 \\ 0.0 \end{pmatrix} \tag{3.2}$$

As can be seen, the gripper's main axis is along X and the default tool length is 20cm. As mentioned already, if the object's bounding box is twice as big as the tool length, i.e. bigger than 40cm, the default tool length is scaled to ensure that only the robot's gripper links are in contact with the object to be grasped.

Predicates for reasoning about reachability. To reason about the manipulation of objects, we define two concepts, *reachability* and *blocking objects*. For stating that an object is reachable, which objects are reachable for the robot or with which arms an object can be reachable, the system provides the predicate reachable. It is defined in two versions, one including the arm with which the object is reachable and one without that information. The two signatures are as follows:

```
(reachable ?world ?robot ?object)
(reachable ?world ?robot ?object ?arms)
```

The implementation is based on the calculation of inverse kinematics, i.e. the problem of finding positions for the robot's joints to reach a given pose. In case of the PR2, the ROS system provides the respective functionality. The predicate holds if the object ?obj–name can be reached by the robot ?robot–name from its current location. All variables can be either bound or unbound. If the variable ?world is unbound, the default world database is used. In the second version, the variable ?arms is bound to a list of symbols that indicate the arms that are required for reaching the object. If multiple combinations are possible, the second version of the reachable predicate has several solutions. For instance, when asserting the reachability of a cup, two solutions can be generated, one with ?arms bound to (:left) and one with ?arms bound to (:right). When reasoning about the reachability of a pot which requires two arms to grasp it, only one solution will be generated with ?arms bound to (:left :right). In contrast, the first version of reachable will have only one solution if a specific object can be reached. That means while the first version only asserts which objects are reachable or if a specific object is reachable, the second version can be used to find out with which arms the object can be reached.

A simple implementation of the reachable predicate might perform the following steps:

1. Get the pose of the object to be reached.
2. Get all valid grasps and required arms.
3. Based on the grasp and the bounding box of the object of interest, compute the pose for the gripper link to reach (with the base frame for the IK solver as reference frame).
4. Verify that there exists a solution for inverse kinematics.

This implementation is sufficient for robots that do not have a movable spine that is excluded from the default kinematics chain used for inverse kinematics computation. However, robots such as the PR2 can lift their torso by about 33cm which influences the reachability of certain poses. Unfortunately, the additional torso joint (a sliding joint), is not included in the PR2's inverse kinematics solver. The consequence is that the reachable predicate needs to try inverse kinematics solutions with different values for joints that are not part of the kinematic chain. As part of the system's knowledge base, we define the predicate robot–pre–grasp–joint–states which yields solutions for joint states to apply before doing the actual inverse kinematics computation. Since asserting joint states changes the world database, the reachable predicate must first copy the database and perform all steps for inferring reachability based on that copy in order to avoid any unwanted side effects. For the PR2, we define three joint states for the robot's spine, the spine lowered completely (value of 0), the joint in the middle (value of 0.165) and the spine at its uppermost position (value of 0.33).

Besides predicates for reasoning about object reachability, the system provides lower level predicates for asserting that the robot can reach a specific point or a 6D pose in space. In contrast to full 6D poses, points do not contain an orientation. For verifying if a point is reachable or for inferring arms that can be used to reach a specific point in space, CRAM provides the predicate point–reachable with the following signature:

```
(point–reachable ?world ?robot ?point ?arm)
```

The only variable that is required to be bound is the variable ?point since the Prolog

inference process is not able to generate a possibly infinite number of points in space as solutions. The implementation of the predicate iterates over all possible grasps and uses the default tool to verify if an inverse kinematics solution exists for reaching the point. For asserting the reachability of a specific pose, we define the predicate pose–reachable with the following signature:

```
(pose-reachable ?world ?robot ?pose ?arm)
```

In contrast to point–reachable, this predicate does not need any grasps since a pose already contains an orientation. The predicate only holds if an inverse kinematics solution could be found to reach the pose ?pose with ?arm. Please note that similar to point–reachable, ?pose needs to be bound while the other variables can be left unbound.

The CRAM reasoning system is also supposed to be used to evaluate potential poses for the robot to stand in order to manipulate objects. Although the system is not designed to perform grasp or motion planning, we still can make certain statements regarding the quality of such a pose. For instance, locations from which the robot can reach an object that should be picked up without being in collision with any other objects is surely preferable to locations where the robot would collide with other objects. The same holds for selecting the arm to use for picking up an object. Please note that the system does not reject locations where the robot is in collision with other objects because a sufficiently good motion planner can still be able to find a collision free path for reaching an object. However, preferring solutions where the robot is in collision with less or preferably no objects, motion plans will be much simpler and the computational time for finding such a motion plan will be shorter.

To reason about and query for blocking objects, the CRAM reasoning system defines the predicate blocking. Similar to reachable it comes in two versions, one with an arm parameter and one without. The signature of the two predicates is as follows:

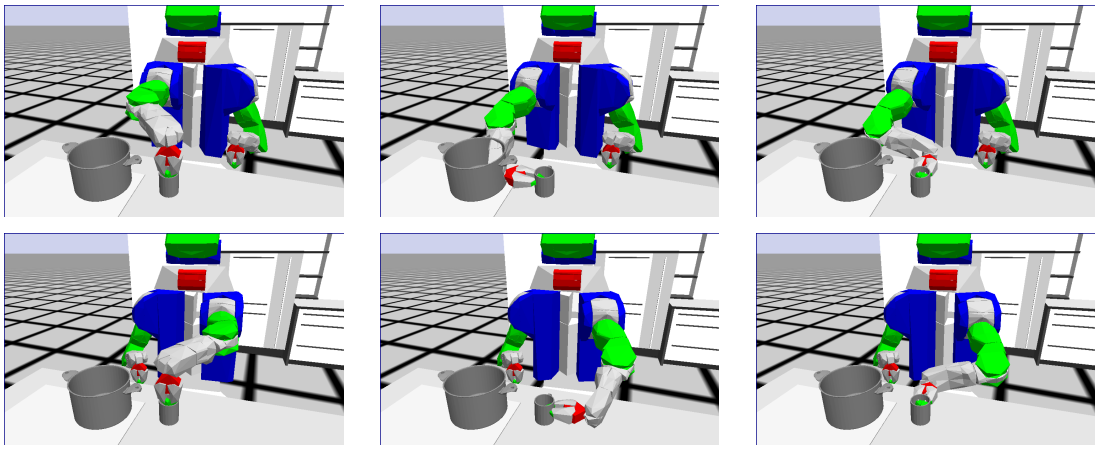


Figure 3.6: *Computation of the* blocking *predicate for the cup. The top row shows the IK solutions for the three possible grasps for the right arm and the bottom row shows the corresponding solutions for the left arm. The robot is in collision with the pot when using the right arm, so the pot is considered a blocking object.*

```
(blocking ?w ?robot ?object ?blocking-object)
(blocking ?w ?robot ?object ?arm ?blocking-object)
```

The first version generates solutions for any object that the robot is colliding with when reaching for ?object and binds the names of the blocking objects to ?blocking-object. One solution for each blocking object is generated. Please note that the system will collect all blocking objects for all usable arms and grasps but each blocking object will only lead to one solution although it might be blocking for different grasps.

The second version generates solutions specific for one of the robot's arms. That way, it is possible to compare the sets of blocking objects for different arms and, for instance, select the arm with the smallest number of blocking objects.

If the user needs a complete list of blocking objects instead of different solutions, the functor setof can be used. The following example shows how to get the set of all blocking objects when reaching for the object named *mug* with the right arm:

```
1 (setof ?blocking-object
2        (blocking ?world pr2 mug :right ?blocking-object)
3        ?blocking-objects)
```

Please note that in contrast to reachable, blocking generates separate solutions for each arm independent of how many arms need to be used to grasp an object.

The implementation of the blocking predicate is slightly more complex than reachable since it needs to perform collision detection. This is done using the contact predicate. In contrast to reachable for which inverse kinematics calls are sufficient, blocking needs to assert the joint states returned by inverse kinematics computation before performing collision checking. To avoid changing the user's world database, it therefore needs to copy the world and perform its computations in that copy.

3.1.4 Belief State Representation using CRAM's World Database

In order to use the geometric world database introduced in this section for decision making during plan execution, the plan execution environment needs to update and keep it consistent with the environment. As explained in Section 2.4, process modules provide the interface between high-level plans and the robot's hardware and perception routines. Only process modules are allowed to change the robot's state or the environment and all perception routines are also encapsulated in process modules. To notify all parts of a CRAM system about changes, process modules send events that contain symbolic information about the corresponding change.

Essentially, process modules can gather information about the world through sensors and perception algorithms running on sensor data, they can move the robot's links, i.e. cause changes in the robot's joint states, or they can interact with the environment by grasping objects and releasing them again. Additionally, the robot can open and close doors and drawers, i.e. change the state of articulated objects. For each of these possible interactions, CRAM defines an event that contains additional information about the change, for instance which object has been detected and by which sensor or the opening angle of an articulated object. For keeping CRAM's world database

consistent, the system subscribes to these events and performs the corresponding assertions in the database. For instance, when the robot grasps an object, two events will be generated, one for notifying the system that the robot moved, i.e. that its links changed their position, and one for indicating that the object is attached to the robot's gripper now which means if the robot moves, the object will move accordingly. In the following, the currently defined events and the implementation of the corresponding event handler for updating the default world database are explained.

robot−state−changed. Whenever the robot changes the position of one of its links or whenever it changes its location in the environment, process modules are supposed to send the event robot−state−changed. Please note that the maximal frequency of events is not restricted. However, the rate of events should be kept as low as possible to save computational resources and memory. For instance, in the current implementation, the navigation process module sends only one robot−state−changed event after the navigation process finished. Manipulation only sends events after an arm trajectory has been executed. Besides a time stamp, the robot−state−changed event does not provide any additional information.

If the reasoning system receives a robot−state−changed event, it re-initializes the position of all robot links from the robot's joint states and the localization module. In a ROS system, TF[8] is used which provides the complete set of transformations between different coordinate frames in the ROS ecosystem.

object−attached. As mentioned already, when manipulating objects, it is important that the system has information about the objects in the robot's gripper at a given point in time. For instance, to decide which arm to use for grasping an object or if it is possible at all to grasp a specific object, for instance for bi-manual manipulation, information about attached objects is important. If the robot changes its state, all attached objects must move accordingly to keep the world database consistent. The corresponding event in CRAM is the event object−attached. The event contains information about the object that has been attached (i.e. the object designator), the name of the link it has been attached to and a symbol naming the manipulator it has been attached to, for instance : left or right.

[8] http://ros.org/wiki/tf

The corresponding action to be executed when an object–attached event has been received is the attached assertion as defined above:

```
(assert ?world (attached ?robot ?link-name ?object))
```

Please note that in the predicate above the variable ?object must be a unique symbol identifying the object to be attached while the object bound to the event is an object designator. The event handler has to translate the designator to the corresponding object instance in the world database. An object designator might reference different object instances over time. However, the data slot in an effective designator must reference an instance of type object–designator–data which contains an object identifier. The object–perceived event must provide a mechanism to map this identifier to an actual instance name in the world database and the same mechanism must be used by the handler for the object–attached event.

object–detached. The implementation of the object–detached event is similar to object-attached. The events contain the same data. In fact, in the current implementation, both events are derived from the same base class object–connection–event. The implementation of the event handler retracts the attached relation using the corresponding retraction as defined above:

```
(retract ?world (attached ?robot ?link-name ?object))
```

The same mechanism for mapping the object designator in the event to the name of the object instance in the world database used in the handler for attaching objects is used in the object–detached event handler as well.

object–articulation–event . Robots operating in human environments might be able (and required) to open and close articulated objects such as drawers, cupboards or a refrigerator. The CRAM system therefore defines an event for signaling changes to

these objects. The corresponding event is called object–articulation–event and contains the object designator of the articulated object for which the articulation state changed and the new opening distance, either an angle for doors or a linear distance for drawers. This data is then used to change the joint state of the corresponding object in the world database.

object–perceived. Whenever an object has been perceived, the event object–perceived is sent by the perception subsystem. Besides the object designator of a newly perceived object, the event object contains an identifier of the sensor that sent it. While this information is not required for updating the world database, it still is useful for instance to evaluate the quality of the received data. The designator referenced in this event must always be an effective designator, containing a reference of type object-designator-data.

One particular problem that has to be solved for a robust and consistent representation of the robot's environment is *entity resolution*, also known as *anchoring*. It is basically the problem of deciding to which instance in the robot's (symbolic) belief about the world a specific object detection belongs. For instance, if the robot knows the location of two cups that are standing next to each other and a new detection of just one cup is received, the system needs to decide if it should create a new object instance or if it should update its belief about one of the two cups. In the latter case it needs to decide which cup to update. The problem of entity resolution is hard and still not completely solved. However, CRAM implements simple heuristics based on overlaps and knowledge about the algorithms used for detecting objects.

Let us assume the robot has a large database of 3D models of objects and the perception system can detect the objects that are in this database. If any object is detected, the handler for the object–perceived event first asserts a new object instance with the corresponding 3D model stored in the model database. Then the system uses the contact predicate to check for overlaps. If two objects with identical type, i.e. with the same 3D model, are overlapping, the system retracts one of the two instances and equates the designator of the new object with the designator of the old object to indicate that they are both referencing the same object. This approach assumes that the self-localization of the robot is relatively accurate and that that object detection

leads to fairly accurate poses, too. These assumptions are valid for the PR2 using a 3D sensor such as the Kinect and a sufficiently distinguishable set of objects.

To account for objects that have been moved externally, the system also needs to remove object instances that should have been visible but could not be detected anymore. With sensors such as the Kinect, it is possible to find clusters of points that are on a supporting plane with considerably few false positives and false negatives. We thus assume that if an object could not be found with this very simple detection algorithm, it is not present in the world anymore or it significantly changed its position. Using the visible predicate, the system can infer which objects should be visible taking into account occlusions and sensor parameters such as the visual field or a maximal detection distance. That way, it can decide which objects need to be removed from the database.

Unfortunately, although in most cases perception is able to find point clusters that correspond to objects on supporting planes, it is not always possible to get the 3D model of the object, either because the object is not in the database or due to sensor noise or other problems of the perception algorithm. In these cases, the system can still triangulate the detected point cloud and assert it in the world database. While dynamic simulation, i.e. stability reasoning is not possible for these objects, the 3D models can certainly be used for visibility reasoning and for inferring occlusions. These triangulated point clouds can even be used to infer if they can be used as supporting planes when assuming the corresponding objects are static (see Figure 3.7).

3.2 Reasoning about Plan Execution

Besides providing mechanisms for representing the robot's environment on a geometric level and for reasoning in this geometric representation, a second important aspect to implement cognitive capabilities in robots is the semantic understanding of plans and reasoning about them. To enable a robot to infer if a plan or sub-plan was successful, i.e. if it achieved its intended effects, and for detecting unintended side-effects of generated plans, it is important to have means for reasoning about plan execution. Connecting changes in the belief state of the robot to actions it executed is not only important for debugging and analyzing plan execution after actions have been performed but also for predicting possible outcomes of a plan using plan projection and

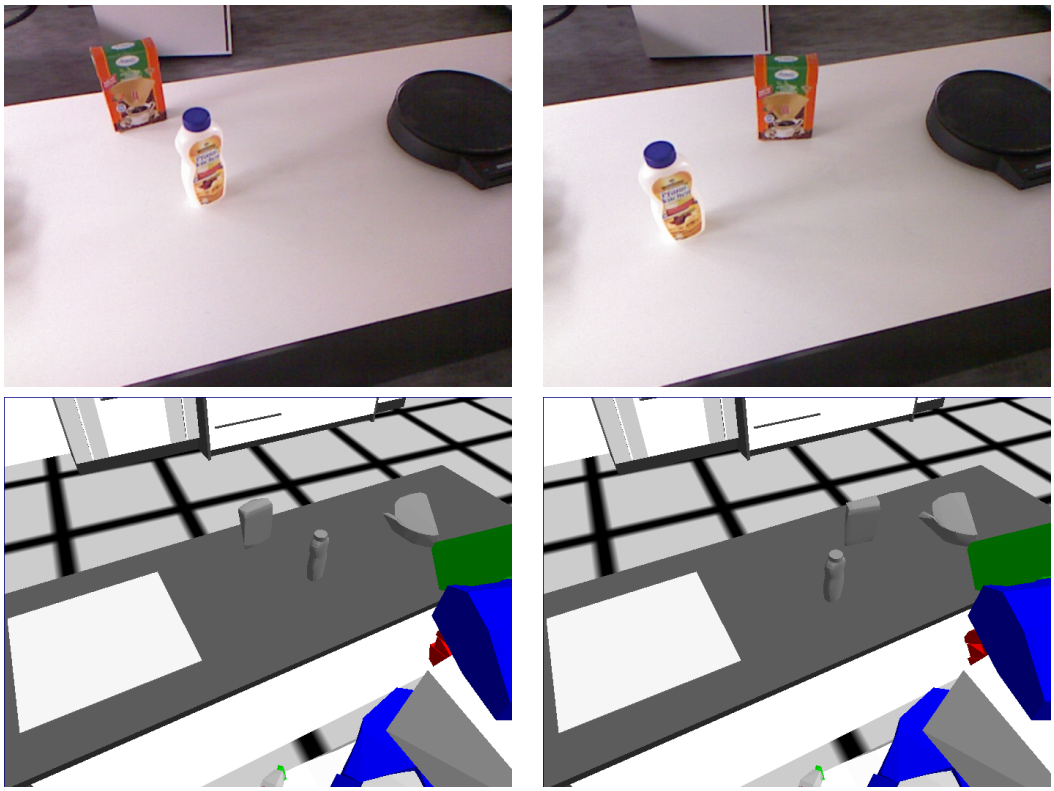


Figure 3.7: *Scenes with objects being triangulated from raw point cloud data. The top row shows the camera image and the bottom row a visualization of the corresponding world database. As can be seen, only half of the pancake maker is visible for the robot and has been triangulated. The picture shows the initial state with the box standing left behind the bottle and a later detection where the box has been moved externally. The system retracted the original instance of the box and asserted a new instance at a different location.*

simulation to analyze and process the simulated results. Also, detailed information about the execution of a plan can be used to generate large sets of training data for learning. Learned models can then be used again for decision making inside plans.

Similar to reasoning about stability, visibility and reachability, the interface for reasoning about plans and plan execution is implemented as CRAM Prolog predicates that query the underlying lower-level data structures. These lower level data structures for reasoning about plan execution are called *execution trace* which can also be stored on a hard disk for later use.

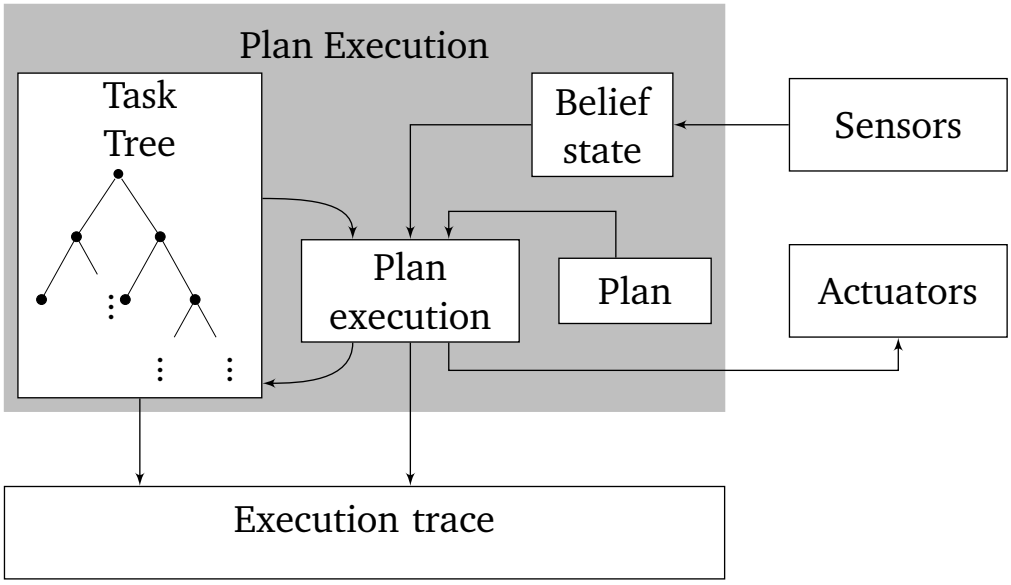


Figure 3.8: *Plan execution and recording of execution traces. The task tree and the internal state of the CRAM plan execution environment (including the belief state) are logged and stored in an execution trace.*

3.2.1 Representation of Execution Traces

As mentioned already, execution traces are basically special knowledge bases on which predicates are implemented, for instance to infer the purpose of plans or to make inferences based on what the robot was believing at a certain point in time. Essentially, the execution trace consists of three parts:

1. The task tree that is generated during plan execution.
2. The values of all fluents at any point in time. This also includes the status fluents of tasks.
3. A time line. It is a sequence of events and the corresponding world states.

Figure 3.8 shows an overview of the generation of execution traces during plan execution.

The task tree. As shown in Section 2.1.4.6, executing a plan always generates a task tree, i.e. a tree data structure with instances of type task–tree–node. Plan execution unfolds the tree, i.e. whenever a declarative form such as a goal (e.g. achieve

or perceive–object), at–location or a plan function is executed, a new node in the task tree is created. The task tree is generated independent of the recording of execution traces and is not only used as part of the execution trace but also for accessing task instances for instance for synchronization using the partial–order macro or calls to wait–for or whenever on a task status fluent. The execution trace contains a serialized version of this task tree.

Recorded fluents. The recording of fluents is more complex because in contrast to the task tree data structure, fluents change their value over time. That means that the CRAM library for recording execution traces needs to provide the functionality to store all values a fluent had over time together with the corresponding time stamp. Fluents already provide a callback mechanism that is used to implement fluent networks that need to update their value when a fluent in the fluent network changes its value. More specifically, the implementation of the setter for the fluent value executes all on–update callbacks. Fluent recording hooks into that mechanism by registering a callback that stores fluent values in a database. However, special care has to be taken when storing the fluent value because fluents can be bound to arbitrary objects such as class or structure instances, lists or arrays that might be altered by the user. Although side effects in CRAM are strongly discouraged, there is no way in Common Lisp to prevent the user from using them. The recording framework for fluents thus needs to make a deep copy of fluent values as soon as a value change happens.

Events and the timeline. The ability to reason about the relation between plans that were executed and the effects in the world (and in the robot's belief state) is one of the most important requirements for execution traces in CRAM. Besides the task tree which contains, in combination with the recorded fluent values, all information about plan execution, representing the belief state and changes in it is a second crucial part in CRAM's execution trace framework. Although fluents can be used to represent the robot's belief and in particular changes in it, they lack semantic information and grounding in an underlying logic. Instead, process modules as introduced in Section 2.4 signal world updates by generating events. The CRAM execution trace library subscribes to these events and records them. Additionally, the belief state based on the geometric world database as introduced in Section 3.1 allows for more sophisticated reasoning and decision making during plan execution. The same events used for updating the world database are also used in the execution trace. By storing the

current world database whenever such an event is received, snapshots of the robot's belief are taken. The temporally ordered sequence of events and the corresponding world database instances form a timeline. The timeline is used to relate effects in the environment, or rather in the robot's belief, to the corresponding parts in the plan.

Persistent storage of execution traces. Execution traces contain a huge amount of valuable information that is not only useful for debugging and verifying if a plan had the intended effects. Additionally, this data can be used by learning algorithms, for instance to learn failure models or decision functions. Many programming languages already provide the functionality to serialize data structures and store them in files. For instance, Python provides the pickle module. Unfortunately, Common Lisp does not have built-in support for marshalling. A number of third party libraries can be found however. The CRAM execution trace uses the library *cl-store*[9] which provides support for different back ends and is sufficiently flexible and extensible.

The execution trace of a CRAM plan contains the task tree with references to low-level thread objects and closures, the world representation with references to a Bullet based world state and fluent values over time with references to arbitrary Lisp objects. Some information, such as thread handles or closures are not relevant for later reasoning about plan execution while lazy lists generated by all CRAM reasoning modules hold important information. All information in the execution trace is encapsulated in the class episode–knowledge. Besides information about the start and end time of the recording, it contains a reference to the task tree object and a reference to a hash table containing all fluents and their value.

While using default serialization handlers for all CLOS objects, the system defines various special handlers which are necessary because cl-store has no means for serializing closures. For task objects, only the name of the task's status fluent and the result value of the task is stored. Code objects store only the s-expression of the executed code, the task object and the arguments passed to the task tree node. Lazy lists are slightly harder to serialize because it is not possible to store the complete computational context to allow for generating additional elements after serialization. Although not a perfect solution, the system thus converts lazy lists to ordinary lists. Instead of expanding the complete list which might either be computationally expensive

[9] http://common-lisp.net/project/cl-store/

or run into infinite loops for an infinite number of elements, lazy lists are just cut, i.e. the generator function is removed and the resulting lists contain only elements that have been computed so far. Storing the Bullet based world database is rather complex since the original Lisp classes contain numerous native system pointers to foreign C++ objects which cannot be stored. Fortunately, the mechanism for storing a world database already generates CLOS classes that mostly contain serializable pure Lisp data. However, for performance reasons and because Bullet allows to re-use them, it still contains references to Bullet shape objects which cannot be serialized easily. Special handlers allow for storing complete meshes or other objects though.

3.2.2 Querying Execution Traces

Accessing the execution trace in CRAM works similar to accessing the geometric world database explained in the previous section. We define predicates for representing aspects of plan execution and to represent the relation between tasks, their parameters, their results, errors or changes of their status. Since time is an important aspect, we add a temporal calculus to the pure first-order logic representation we used so far. The predicates to reason about the task tree together with temporal predicates to reason about timelines provide the tool set to reason about plan execution.

Reasoning on timelines. Occasions have been introduced in Section 2.5.2. An occasion is a logical expression describing the state of the world at a certain point in time. It is grounded by predicates in the CRAM reasoning system and hold over intervals in time. Additionally, we define the concept of events that indicate changes in the (believed) world states, i.e. occasions can only change their truth value when events occur. Events are sent by process modules, i.e. they are not generated by high-level plans but by low-level components of the executive that directly interact with the robot's hardware and environment. This is an important property because that way, the world states that are to be achieved (by high-level plans) are separated from the actual interactions with the environment. While high-level plans can be executed concurrently, events are always generated sequentially and, per definition, do not have a duration.

To access the timeline that is stored in an execution trace, CRAM provides the predicate execution–trace–timeline with the following signature:

```
(execution-trace-timeline ?trace ?timeline)
```

The variable ?trace must be bound to a valid execution trace object. The predicate then unifies the variable ?timeline with the timeline in the execution trace.

To reason about timelines which basically represent the robot's belief state during plan execution and changes in it, we define two predicates. The predicate holds allows for asserting occasions and their duration. It is defined as follows:

```
(holds ?timeline ?occasion ?time)
```

The variable ?timeline must be bound to a valid timeline object. The occasion must be bound to a valid Prolog expression that is proven on specific instances of the Prolog world database at specific points in time. It can contain free variables. The variable ?time is used to control which world states on the timeline have to be considered. It needs to be bound to a pattern of one of the following forms:

- (at ?t) Prove the occasion at a specific point in time indicated by the time stamp ?t. The implementation basically loads the world database that was valid at ?t and tries to prove the occasion using that database.
- (during ?t–1 ?t–2) Prove the occasion in all different valid world databases in the interval specified by [?t–1, ?t–2). The holds predicate succeeds if the occasion could be proven in at least one world database. If the occasion contains free variables and the value of the variables changes between world instances, choice points are generated.

- (throughout ?t–1 ?t–2) Prove the occasion in all different valid world databases in the given interval. The holds predicate succeeds only if the occasion could be proven in all world databases with the same assignment of (free) variables. If the occasion holds in all world databases but some variable assignments change or if the occasion does not hold in at least one world database, holds fails.

With these tree variations of the holds predicate, it is possible to express occasions in the belief state of the robot and relate them to goals. For instance, we defined achieve to only terminate successfully if the robot believes that the corresponding occasion holds in the robot's belief at this time. Suppose the occasion of an achieve expression is (loc Cup Table), i.e. the cup should be on the table and the achieve terminated at time stamp 30.15. The corresponding holds expression for verifying that the occasion really holds is:

```
1 (holds ?timeline (loc Cup Table) (at 30.15))
```

Please note that the ?timeline variable needs to be bound to a valid timeline object. It can be left unbound when the user intends to use holds during plan execution to use the default timeline of the current execution episode. Outside of the CRAM execution context of a plan however, the default timeline is unbound and for each new execution episode, a new timeline is started.

To assert the occurrence of events on the timeline, the CRAM execution trace library implements the predicate occurs with the following signature:

```
(occurs ?timeline ?event ?time)
```

The predicate states that an event pattern ?event was sent by a process module at time stamp ?time. Although events are instances of CLOS objects on the lowest level, they are converted to logical patterns that can be matched by CRAM's reasoning sys-

tem when an execution trace is recorded. Table 3.3 gives a summary of the event patterns currently supported by CRAM.

(robot–state–changed)	The robot or one of its links moved
(object–attached ?obj ?link)	An object has been attached to one of the robots links, e.g. by grasping it.
(object–detached ?obj ?link)	An object has been detached from a robot link, e.g. by opening the gripper.
(object–articulation–event ?obj ?distance)	An articulated object (e.g. door, drawer) has been opened or closed.
(object–perceived ?obj ?sensor)	An object has been perceived in one of the robot's sensors.

Table 3.3: *Event statements.*

The implementation of occurs is much simpler than holds because events do not have any temporal extent and no handling of time intervals is necessary. The implementation of the predicate basically iterates over all (temporally ordered) pairs of event patterns and the corresponding time stamps and unifies them with the two corresponding variables in the occurs predicate. For instance, the following example shows how to find all objects that were detected by the robot during plan execution:

```
1 (occurs ?timeline (object-perceived ?object ?_) ?t)
```

The variable ?timeline has to be bound to a valid timeline object. The result of this predicate will be a set of solutions with pairs of object designators and the time stamps when they were detected.

Reasoning about fluent values. Fluents are a powerful tool to represent values that change over time. Not only do they allow for waiting for value changes, for implementing synchronization and reactive code. They also enable CRAM to store the complete history of their values over time. In particular, fluents are heavily used internally for implementing synchronization mechanisms in task objects. Their main use case in execution traces is therefore to store the history of a task's state and the transitions between them. All fluents have a unique identifier that is used to access them in the

execution trace. The predicate to access the value of a fluent at a specific time is fluent–value–at with the following signature:

```
(fluent–value–at ?trace ?fluent–name ?value ?time)
```

Apart from ?trace, none of the variables in the above example need to be bound but one important property on the binding of the variable ?value has to be noted. The value will not be referentially equal (i.e. equal using the Common Lisp eq function) to any of the fluent's values as referenced in a CRAM program, at least if the value is not a simple type such as a number. For execution traces that were stored in files, this is clear. However, in online execution traces, the reason is that in order to deal with side effects on fluent values, the system has to make a shallow copy of all objects and therefore, while the copied objects contain the same data as the original object, they do not share the same reference.

Reasoning on the task tree. The predicates presented so far allow for accessing the robot's belief state and querying changes in it but they do not access the task tree to provide a relation to the actual robot control programs. CRAM implements predicates to access data of task objects that are referenced in the task tree, i.e. that contain semantic annotations, predicates to reason about their relation (e.g. parent/subtask), predicates to access the value of designators and predicates for querying failure objects.

To assert that a variable is bound to a task object or to enumerate all tasks, CRAM provides the predicate task with the following signature:

```
(task ?trace ?task)
```

It holds for each task object that is stored in the corresponding execution trace.

The system only stores a subset of tasks on the task tree, including tasks with semantic annotations and tagged tasks. Only these tasks can be accessed by the task predicate. Semantic annotations are either created by goals, i.e. procedures created with def–goal or special macro forms such as at–location. To access the semantic annotation of a task, CRAM implements the predicate task–goal with the following signature:

```
(task–goal ?trace ?task ?goal)
```

The variable ?trace has to be bound to an execution trace object while the variable ?task can be either bound or unbound. The variable ?goal is unified with the goal pattern of a task. For instance, let us consider the definition of the pick-up sub-plan:

```
1 (def–goal (achieve (object–in–hand ?object))
2   ...)
```

When it was executed successfully, the following expression will hold on the corresponding execution trace

```
1 (task–goal ?trace ?task (achieve (object–in–hand ?object)))
```

If ?task is unbound, all tasks that grasped an object will be solutions. Additionally, the object designator of the grasped object will also be bound in the corresponding solutions.

Besides the goal of a given task object, other interesting aspects are the start and the end time of the task and its result. In case of a failure, the actual failure object is relevant as well. The corresponding predicates for accessing these task properties are:

- (task–status–at ?trace ?task ?status ?time) Query the status of a task at a specific point in time.
- (task–start ?trace ?task ?time): Query the start time of the task, i.e. the time stamp where its status transitioned to *:running* the first time.
- (task–end ?trace ?task ?time): Query the end time of a task, i.e. the time stamp where its status transitioned to a final status (either *:succeeded*, *:evaporated* or *:failed*).
- (task–outcome ?trace ?task ?status): Query the final status of the task.
- (task–result ?trace ?task ?result): Access the return value of a task. This predicate only holds if the task succeeded.
- (task–failure ?trace ?task ?failure): Query the failure object thrown by the task. This predicate only holds if the task's final status was *:failed*.

Besides the properties of task objects, a second and probably more important aspect is reasoning about the relations between tasks. This includes assertions about the task tree, i.e. parent-child relationships and reasoning about concurrency and potentially conflicting resources. The hierarchical relationship between tasks is interesting because it allows to make statements about the purpose, intentions and roles of certain plan parts. Informally, the intention of an achieve goal is to make its occasion hold in the world. The intention of a perform goal is to execute an action on the robot's hardware, the intention of a perceive goal is to verify occasions and detect objects using the robot's sensors and the intention of at–location blocks is to only execute the body of the expression at a certain location. On the other hand, the purpose of a task is to contribute to the fulfillment of the intention of its parent tasks.

In CRAM we define two predicates to express the relation between a task and its child task, subtask to assert the relation between a parent task and its direct child task, and subtask which holds for all direct and indirect sub-tasks of the parent task, i.e. will hold for the transitive hull of all sub-tasks. The signature of the two predicates is similar:

```
(subtask ?trace ?task ?sub-task)
(subtask+ ?trace ?task ?sub-task)
```

Similar to all other predicates that are computed based on an execution trace, the variable ?trace has to be bound to an execution trace object. The other variables can be either bound or unbound and choice points are generated in the reasoning engine for the different solutions of the relation. For instance, if we bind ?task to an object that has two children which again have two children, the predicate subtask will generate two different solutions while the predicate subtask will generate six solutions.

Helper predicates for accessing data structures. In addition to the predicates for basic reasoning about the robot's belief state and plans, a plan normally utilizes other data structures, particularly designators, failure objects and rarely instances of other CLOS classes. CRAM provides a few helper predicates to access these data structures. For instance, designators are normally used as parameters for goals and can therefore be bound to variables by the task-goal predicate and failure objects can be accessed using the task-failure predicate.

As explained in Section 2.3, designators are objects that contain key-value pairs symbolically describing the entity referenced by the designator and an optional data slot which is called the designator solution. Different designators that are referencing the same entity are equated and form a chain of designators representing the history of an entity in the robot's belief. To access different solutions at different points in time, CRAM provides the predicate designator-solution-at with the following signature:

```
(designator-solution-at ?designator ?solution ?time)
```

Please note that the execution trace does not provide a complete list of designators, i.e. it is not possible to enumerate all designators. The reason is that designators are only referenced by goal patterns or the with-designators CRAM form and do not have an explicit name as fluents do. Therefore, no execution trace object has to be provided

and the ?designator variable has to be bound in the designator–solution–at predicate. Additionally, it is important to access the symbolic description of referenced entities at specific points in time. For that, CRAM provides the predicate designator–property–at with the following signature:

```
(designator–property–at ?designator ?property ?time)
```

The predicate asserts a designator property at a specific point in time and can be used to enumerate all properties at a specific point in time or all time stamps at which a property is part of the designator's description. More specifically, based on the variable ?designator, the predicate designator–property–at first queries all designators equated with it during the course of actions recorded in the execution trace. Then it finds the designator that belongs to the time stamp specified by the variable ?time. Finally, all solutions are generated for which the variable ?property matches the designator's properties.

Failure objects do not have any temporal extent, i.e. once an instance is created, it will not change over time. Therefore, all predicates related to accessing failure objects do not require a time variable. The most interesting aspect of failure objects is its type. To access it, CRAM provides the predicate failure–type with the following signature:

```
(failure–type ?failure ?type)
```

The predicate unifies the failure object's Lisp type with the variable ?type. To access different slots, common CRAM reasoning utilities such as lisp–fun can be used.

3.2.3 Examples for Reasoning on Execution Traces

The reasoning mechanisms defined in this section provide a powerful way to investigate what the robot did, when it did it, why it did it and what the believed state of the world looked like. In this section, examples to demonstrate the expressiveness of the predicates for querying the execution trace will be presented. In the following examples, let us assume the robot executed a table setting plan that included several pick-and-place actions.

To find all locations where the robot was standing while picking up an object, we query all at–location goals that were executed in the context of an object–in–hand goal and access the respective location designator. The following code snippet shows the corresponding Prolog code:

```
1 (and (task-goal ?trace ?task-1 (achieve (object-in-hand ?_)))
2      (subtask+ ?task-1 ?task-2)
3      (task-goal ?trace ?task-2 (at-location ?loc)))
```

If we want to find out if the robot grasped a specific object, for instance a plate, and the initial location of that object, we can make the following query:

```
1 (and (task-goal ?trace ?tsk-1 (achieve (object-in-hand ?obj)))
2      (task-goal ?trace ?tsk-2 (perceive-object ?obj))
3      (subtask+ ?trace ?tsk-1 ?tsk-2)
4      (task-end ?trace ?tsk-1 ?t1)
5      (task-end ?trace ?tsk-2 ?t2)
6      (designator-property-at ?obj (type plate) ?t1)
7      (designator-property-at ?obj (at ?loc) ?t2))
```

First, we find all pick-up plans and bind the variable ?obj to the object designator used for grasping. Then we find all sub-plans that were perceiving that object to find

the point in time where we definitely know the pose of the object. Then we need to query the end times of the two tasks in order to query their designator properties at the respective time stamps. Finally, we assert that the designator references an object of type “plate” and query the location designator describing the location of the object right after detecting it. Please note that if the goal perceive–object failed to find the object, the query for the designator properties would fail as well and the above example would not yield any solutions.

To find all locations the robot navigated to, we can query all perform goals with action designators describing navigation actions. The following code snipped shows the corresponding Prolog query:

```
1 (and (task-goal ?trace ?task (perform ?action))
2      (task-end ?trace ?task ?t)
3      (designator-property-at ?action (type navigation) ?t)
4      (designator-property-at ?action (goal ?loc) ?t))
```

This query will have multiple solutions, one for each location the robot navigated to.

Another interesting aspect of plan execution are failures. For instance, to find all pick-up tasks that failed, we can use the following query:

```
1 (and (task-goal ?trace ?task (achieve (object-in-hand ?_)))
2      (task-outcome ?trace ?task :failed))
```

Often, the system is able to recover from single pick-up failures. In that case the query above would not hold since recovery takes place in the achieve goal object–in–hand. However, object–in–hand contains multiple calls to **perform** which might have failed. To find all pick-up goals that succeeded but required several attempts to pick up the object, we can make the following query:

```
1 (and (task-goal ?trace ?tsk-1 (achieve (object-in-hand ?_)))
2      (task-outcome ?trace ?tsk-1 :succeeded)
3      (subtask+ ?trace ?tsk-1 ?tsk-2)
4      (task-goal ?trace ?tsk-2 (perform ?act))
5      (task-outcome ?trace ?tsk-2 :failed))
```

When executing a complex plan with a high number of steps, it is possible that one action invalidates a goal that has been achieved previously. Although all individual plan steps succeeded, the overall outcome of the plan would still be wrong in that case. Using the execution trace, we are able to detect such errors. Each plan has one single top-level task, the task that does not have a parent task itself. To check if any goal that should have been achieved does not hold in the end of the top-level plan, we make the following query:

```
1 (and (top-level ?trace ?tt)
2      (task-end ?trace ?tt ?t)
3      (subtask ?trace ?tt ?sub)
4      (task-goal ?trace ?sub (achieve ?o))
5      (not (holds ?trace ?o (at ?t))))
```

As can be seen, we use the occasion of achieve goals and verify that the goal actually holds in the robot's belief state when the top level plan finished reusing the goal occasion in the holds predicate. Since holds is implemented based on the physics-based world database which is updated from events generated by perception and low-level actions, we can verify if the goal actually holds by just using knowledge of the timeline and not the task tree. To summarize, this example shows how the semantic annotation of plans using occasions is used to automatically query the robot's belief state in order to verify that a plan actually lead to the its asserted behavior. This allows to detect flaws that are hard to detect using common error signals such as exceptions since sub-plans are not analyzed separately but the complete course of action is taken into account.

3.3 Related Work

In robotics, qualitative reasoning and more specifically reasoning about collisions and the geometric constraints of the robot and its environment, has been used heavily in particular for motion planning [Latombe, 1991]. The combination of motion planning with symbolic planning methods has been shown in [Zickler and Veloso, 2009], [Ruehl et al., 2010] and [Dornhege et al., 2009]. One interesting aspect of [Dornhege et al., 2009] is that the applicability of symbolic actions is verified using geometric reasoning. This approach is similar to our approach of implementing Prolog predicates based geometric reasoning to resolve them on a sub-symbolic level.

The work presented in [Zickler and Veloso, 2009] shows the integration of dynamic aspects in planning using a physics engine. General purpose physics-based simulation using a tableaux based reasoning system is shown in [Johnston and Williams, 2008]. However, the limitations of this system are that no reasoning about visibility and perspective taking is possible.

A system to reason about perspective taking and visibility in the context of human-robot interaction is shown in [Marin et al., 2008].

The authors of [Kunze et al., 2011] present a system for extensive reasoning about robot action execution on a very detailed level. Complete action sequences are simulated and all important aspects of action execution are recorded. Prolog is then used to perform reasoning on the recorded execution scenarios. The use of a physically accurate simulation environment can lead to very accurate results, although limited by the accuracy of the underlying physics engine and the modeling detail of the robot and the environment. This approach is computationally very expensive since all aspects of action execution including control loops and perception are modeled.

As shown in [Weitnauer et al., 2010], the accuracy of commonly used physics engines such as *Bullet*, can be improved by automatically adapting simulation parameters. The authors have shown that the accurate prediction of actions using physics engines is feasible.

The placement of objects based on spacial relations specified in natural language has been presented in [Coyne and Sproat, 2001]. There, the authors use predefined areas, so called spacial tags to resolve properties such as *on* or *under*.

Once a system can reasoning about the execution of its actions, it becomes capable of answering questions about the internal data structures, the course of action and the decision making process that leads to the executed sequence of actions. The only similar system the author knows about that runs fully integrated on a robot platform and is able to reflect about itself is the GRACE system [Simmons et al., 2002] which gave a talk about itself at a conference on artificial intelligence. This included information about it's capabilities and the tasks being executed.

Meta-level architectures such as a Prolog interpreter implemented in Prolog clauses [Sterling and Shapiro, 1994] show some similar characteristics as the reasoning system presented in this work. Programs are interpreted as data that can be reasoned about. Thus, aspects of the interpretation are made explicit. But in contrast to these systems that are applied on disembodied problems inside a more or less static problem domain specified as facts in knowledge bases, our system allows to reason about plans that are executed on a robotic platform that acts in a complex and dynamic environment. Our system allows to handle uncertain knowledge and beliefs, and grounds the knowledge representation in the underlying data structures of the robot.

The first order representation described in this paper can be compared to Temporal Action Logics (TAL) [Doherty et al., 1998]. Whereas TAL's authors define a language for reasoning about action and change, we add a concept of intention. That means, we not only describe what happened but also what the robot wanted to achieve.

Runtime analysis of programs and their execution behavior is heavily used in Just-In-Time compilers such as the Java Virtual Machine for hot-spot optimization [Arnold et al., 2004]. There, program execution has to be recorded and analyzed as well, though on a much lower level than the work presented in this chapter.

3.4 Discussion

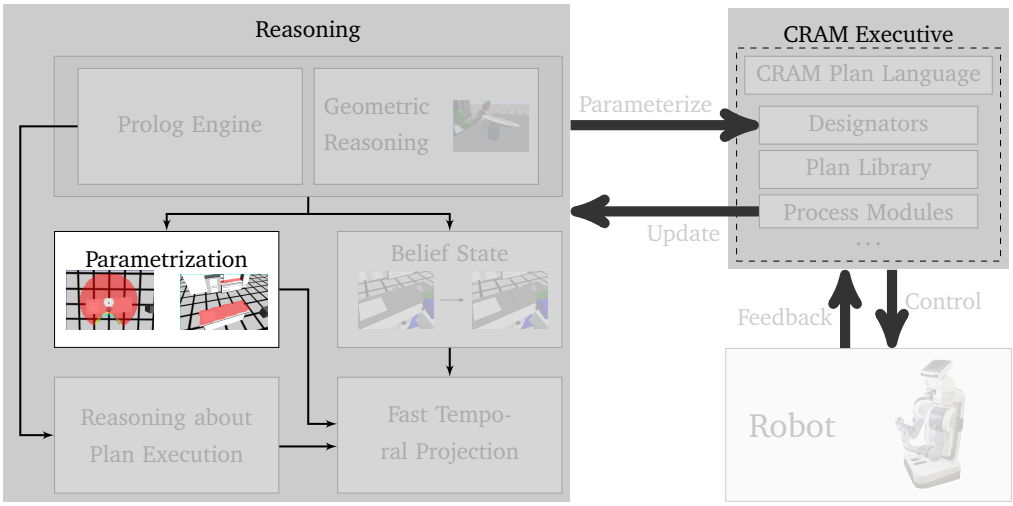
In this chapter, we explained the integration of geometric reasoning and symbolic reasoning by implementing predicates such as *stable*, *visible* and *reachable* using a physics engine, off-screen rendering of the scene from the robot's perspective using OpenGL and inverse kinematics computation. The approach shown in this work allows for quick inference and combines the advantages of symbolic reasoning with more

accurate mechanisms that are based on a geometrically accurate representation of the environment. The system is used for maintaining a concise internal representation of the world which is used to execute geometric reasoning tasks and for fast predictions of the effect of actions.

In addition to reasoning about geometric aspects of the environment, we presented how reasoning about plan execution is implemented. Reasoning about the actions the robot executed, either in reality, in simulation or when projecting a plan is crucial for finding flaws in the robot's actions in order to avoid or repair them.

Chapter 4

Parameterizing Plans



The physics based reasoning system introduced in Chapter 3 allows for computing the truth value of predicates but also to generate solutions that satisfy a specific Prolog expression. However, the generation of solutions is limited to purely symbolic variable bindings. For instance, while it is possible to iterate over all sensors of the robot and verify if an object can be seen in one sensor to find all sensors in which an object is visible, the system cannot generate locations for the robot's base from which the object can be seen. The reason is that the set of sensors is symbolically defined and is rather small while the set of all possible 6D-poses is of infinite size which means that the computational complexity explodes. On the other hand, when executing plans on a robot, parameters such as the locations for putting down objects or for the robot to stand when performing actions is an important but also hard task that needs to be solved for robust and flexible action execution.

The CRAM system provides a non-optimal sampling based solution for generating locations that satisfy a set of symbolic constraints. In contrast to motion planning which tries to find a path of poses from a starting pose to a goal pose, CRAM generates goal poses for placing objects or the robot's base. Parameter generation in CRAM is supposed to be used at run-time. This implies that it needs to be as fast as possible without any negative impact on the execution time of plans. Additionally, for constraints such as reachability, visibility or the stability of poses, it is hard to define objective functions for optimization algorithms. Most often, finding a suitable solution fast is more important and also more feasible than finding a provably optimal solution.

The system described in this chapter is an extension to *location designators* as introduced in Section 2.3 and is integrated using the interfaces defined for location designators as explained in Section 2.3.2. The resolution of a location designator is divided into two steps, a generation step and a verification step. Generators yield sequences of solution candidates in the form of lazy lists and verification functions accept or reject solutions.

Informally, the generation of a pose, for instance *for the robot to reach the cup*, is resolved as follows:

1. Generate a two-dimensional grid based on the location designator constraints with values greater than zero for all potentially valid solutions. The grid should approximate the solution space of the designator properties but does not necessarily need to be completely correct. It is a map representing the solution density of a specific cell where greater values represent more solutions for a specific constraint.

2. Use the grid as a probability density function to generate random samples.

3. Use heuristics to resolve the orientation and Z coordinate (height) of the generated pose. For poses for the robot to stand, this is always zero, i.e. the pose will always be on the floor.

4. Use the physics-based reasoning system to prove that the solution is valid.

As can be seen, the first three steps generate a solution candidate while the last step verifies that the candidate is a valid solution with respect to the location designator to be resolved.

In this chapter we will first describe the system for constructing the two-dimensional grids representing valid poses from the symbolic constraints specified by location designators. Then we will explain the code API and the currently implemented grid generator functions together with the corresponding validation functions.

The work presented in this Chapter has been published previously in [Mösenlechner and Beetz, 2011] and in [Mösenlechner and Beetz, 2013].

4.1 Density Maps for Sampling Solution Candidates

Since backtracking over a possible infinite number of poses to find solutions in a Prolog inference process is not feasible, a different mechanism that has the potential to find valid solutions quickly is important. Location designators contain constraints for the described locations on a symbolic level while, on the other hand, solutions required by the CRAM executive need to be sub-symbolic values, e.g. poses. The constraints specified with designators normally restrict the solution space but still leave many potentially valid solutions. For instance, let us consider the following location designator:

```
(location ((on counter-top)))
```

It constrains the solution space of all points that are on any counter-top in the robot's environment. Figure 4.1 shows a density map for the above designator in TUM's kitchen lab. It can be seen that two areas are counter tops and all locations on them are equally valid. The process of resolving location designator properties and generating poses (i.e. positions and orientations) in space is split up into three parts in order to deal with the computational complexity of the six-dimensional space of poses (three dimensions for the position in space and three for the orientation) and to make

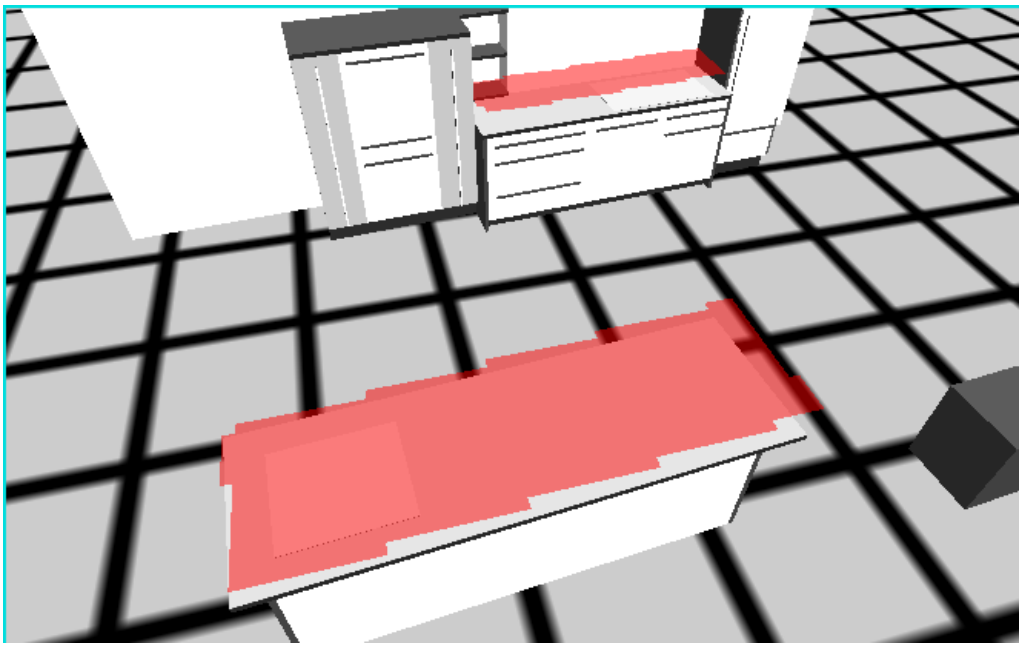


Figure 4.1: *Density Map for locations* on a counter-top. *The red areas indicate values in the density map that are greater than zero meaning that these points are potential solutions for locations on counter tops.*

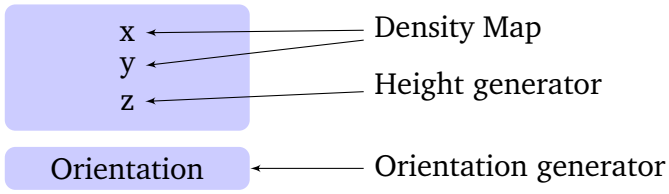


Figure 4.2: *The different generator functions to generate a (sampled) pose in three dimensional space, including its orientation.*

a sampling based approach feasible. As mentioned already, the generation of a pose consists of three steps: sampling points from a two-dimensional probability distribution generated from a combination of density maps, the generation of the z-coordinate by a different generator function that can either be based on heuristics, sampling or a combination of both and the generation of the orientation of the pose by a third generator that can be just a heuristic or also be based on sampling as shown in Figure 4.2. This section will provide the concepts and reasoning related to the first part, sampling based on density maps.

To generate solution candidates, the first step is to convert designator properties, i.e. the symbolic key-value pairs of the designator's constraints to a probability density

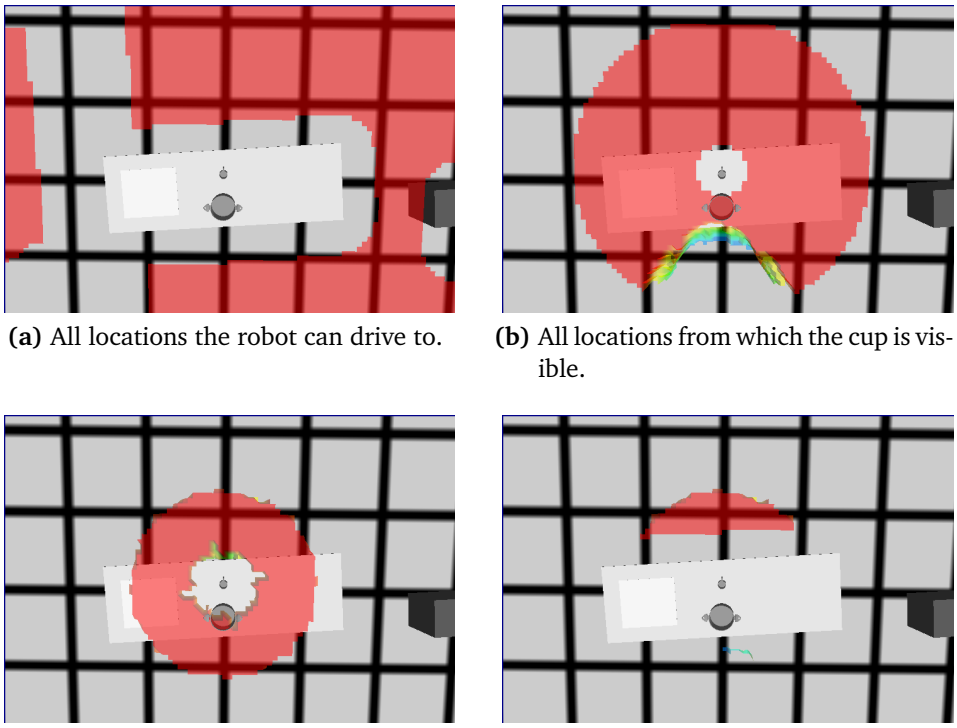
function that can be used by sampling algorithms. A valid solution for a designator must satisfy *all* constraints. In other words, the current system does not support *or* combinations but only *and* combinations of constraints. If a solution does not satisfy one of the constraints it cannot be a solution for the designator. Density Maps, i.e. two-dimensional matrices of positive real numbers allow for representing spacial constraints such as *on*, *in* but also to represent locations from which objects can be reached to a certain extent. Additionally, they can be merged easily and they can be easily converted to valid probability density functions to be used for sampling. Each entry in the density map matrix corresponds to a grid cell in the x-y-plane of the robot's environment. Values of zero indicate that the corresponding grid cell cannot be a solution for the designator to be resolved and values greater than zero indicate potential solution candidates. A greater value indicates a greater probability of the corresponding cell to be a valid solution and therefore increases the probability of the corresponding cell to be selected as a solution.

In our very simple example of locations on a countertop, the corresponding density map is generated by first finding all countertops using a semantic map of the environment. Then these objects are projected down to the x-y-plane and all cells that are inside a counter top are set to equal values greater than zero while all other cells are set to zero. The density map is converted to a probability density function by normalizing its values. To form a valid probability density function, all values must sum up to one and since the density map is a discrete grid, the normalization operation is rather simple. First, the sum of all values is computed and then each value is divided by this sum.

More complicated designators, i.e. designators with more constraints, are resolved by first generating several density maps and then combining them. As an example, let us consider the following example designator:

```
(location ((to see) (to reach) (obj Cup1)))
```

It describes a location for the robot's base from which it can see and reach the object instance named *Cup1*. To resolve it, three density maps are generated: one represent-



(a) All locations the robot can drive to.

(b) All locations from which the cup is visible.

(c) All locations from which the cup is reachable.

(d) Combination of all three density maps.

Figure 4.3: *The three different density maps and the resulting density map that are generated by the designator description(* `location ((to see) (to reach) (obj Cup1 ))`*). As can be seen, the example scene consists of a cup and a pot that is standing close to the cup with the pot occluding the cup.*

ing all locations that the robot can drive to, one representing all locations from which the robot probably can see the object and one from which it is likely to be able to reach the object. Figure 4.3 shows the three distributions. To combine them, the three density map matrices are multiplied component-wise which removes all cells that contain a zero in either of the three density maps. After normalizing the resulting density map, samples drawn from the resulting distribution will satisfy all constraints of the above designator.

One important property of our approach based on the combination of multiple probability density functions is its flexibility and extensibility. The system allows to add new functionality by just loading the corresponding libraries without requiring the user to execute initialization functions. The current system provides some rather general implementations that are purely based on ROS and not specific to a robot platform,

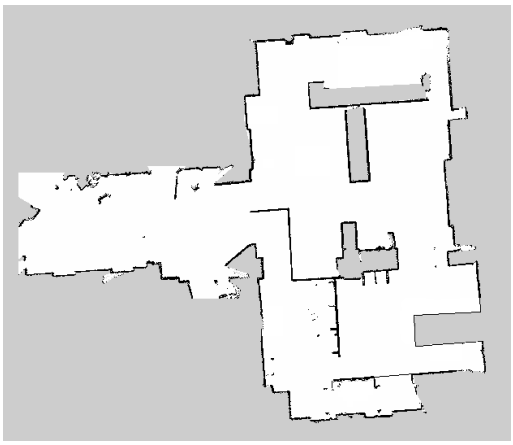
but also modules that exploit more knowledge about the environment or the robot. One example for the former case is a density map implementation based on a two-dimensional map of the environment. This map is provided by ROS in any case since it is used for localization and was build using SLAM. Examples for the latter case are density maps using a semantic map provided by Knowrob to resolve locations on tables, in cupboards, etc. but also to provide a similar functionality as the occupancy grid based density map. CRAM also provides a robot specific density map to resolve reachability. The corresponding module uses an inverse reachability map [Zacharias et al., 2007] to only include locations from which a specific pose can be reached by the robot. The specifics of the different density map providers will be discussed in Section 4.1.1.

4.1.1 Implementation of Density Maps

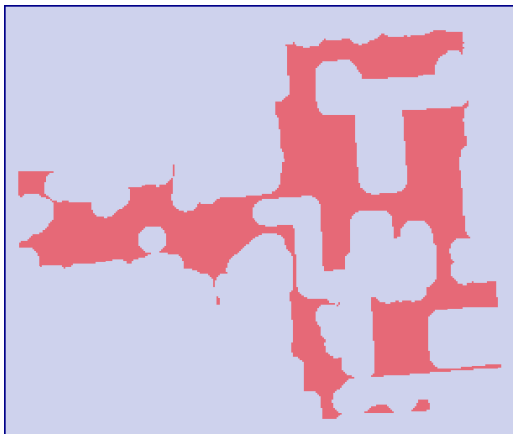
In its current state, CRAM provides a number of density map generators for different designators including designators describing poses on tables, in cupboards, in the refrigerator or in drawers as well as designators describing locations for the robot to stand to reach or to see specific objects or poses. In this section, we will explain the different generator functions and their grounding in detail.

4.1.1.1 Occupancy Grid Based Density Maps

Since mobile robots need to localize themselves in their environment these systems usually already provide two-dimensional occupancy grids (i.e. maps) containing all (known) obstacles. Figure 4.4a shows an example map used in the TUM kitchen lab. In particular, this map contains information about areas that are known to be free, i.e. that the robot can drive on, and about obstacles. We use this information to resolve designators for the robot to stand since one of the (implicit) constraints of such designators is that, when standing at the corresponding location, the robot must not be in collision with anything in the environment. The corresponding density map contains non-zero values at all cells in the occupancy grid that are marked as free and that are not closer to a cell marked occupied than a certain threshold. Figure 4.4b shows the density map that is generated from the occupancy grid shown in Figure 4.4a. Please



(a) Example of an occupancy grid used by the robots for self-localization based on laser sensors.



(b) Density Map generated from the occupancy grid shown in Figure 4.4a.

Figure 4.4: *Occupancy grid used by navigation and the corresponding density map for representing locations the robot can navigate to.*

note that the density map can contain areas that are not reachable by the robot. In other words, the density map can consist of multiple unconnected areas.

Currently, only designators to reach or to see objects are used to position the robot's base. The following list shows the corresponding designator properties:

- `(to see) (obj ?object)`
- `(to reach) (obj ?object)`
- `(to execute) (action ?action)`

The occupancy grid based density map module matches any designator that has these properties and adds the corresponding density map to the designator resolution process.

4.1.1.2 Density Maps using a semantic map

In high-level plans, locations are specified symbolically. This not only includes locations for the robot to stand, but also locations for searching for objects and for putting them down. In a human kitchen environment, the objects of interest are nor-

mally stored in cupboards and drawers or stand on counters or tables. To be able to resolve these symbolic descriptions of locations in the environment, CRAM integrates a semantic map [Pangercic et al., 2012] provided by KNOWROB [Tenorth and Beetz, 2013]. The semantic map is a knowledge base that contains all objects of interest in the robot's environment. Besides these object instances, it also provides the geometric information required to resolve locations or even generate an environment for simulation. It includes the dimensions of the object and their location in space and information about sub-parts and the connecting joints and their properties, including joint limits and rotation axis. One of the most important aspects is that object instances are grounded in an underlying knowledge base of concepts, specified in OWL. For instance, to find a location to store the milk, the system can infer that milk is perishable and perishable goods should be stored in a refrigerator.

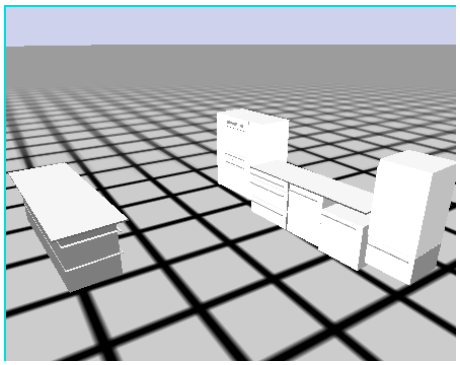
To understand the information that needs to be provided by the semantic map, let us consider a few examples for location designators that are commonly used in CRAM high-level plans. The following list gives an overview of the corresponding designator properties and which features of the semantic map need to be used.

- (on <owl type>): The location needs to be on objects of the corresponding owl type. This can include tables and counters. To generate the corresponding density maps, the respective OWL instance needs to provide a position, width and height.

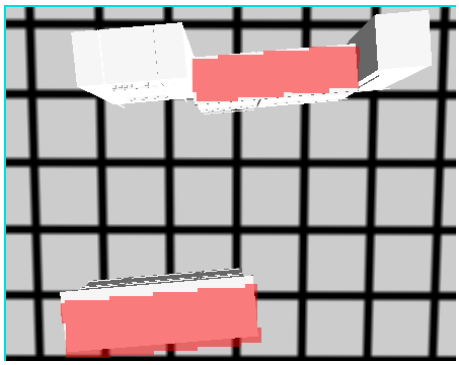
- (in <owl type>): Restrict locations to areas that correspond to objects of the specified type. The density maps are similar to the on relation but a different heuristics for generating a height value (i.e. the z-coordinate) must be used.

- (name <instance name>): Restrict the generated density map to a specific instance in the semantic map, identified by its name.

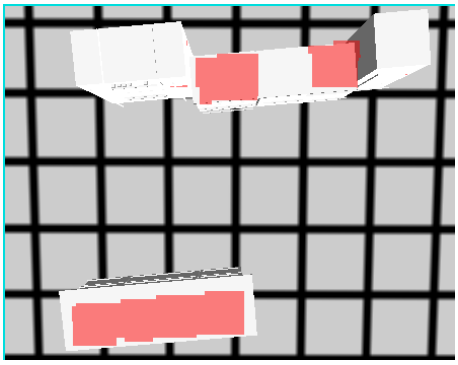
Currently, the shapes of all objects of interest provided by the semantic map are rectangular which simplifies density map generation. To implement the corresponding density maps, the system needs to set the values of all density map cells that belong to the projection of the corresponding semantic map objects on the x-y-plane to numeric values greater than zero. More specifically, to resolve a location that is only constrained by the *on* or *in* property, the system first queries the semantic map for the set of all matching objects. Then it iterates over all objects and sets the value of each



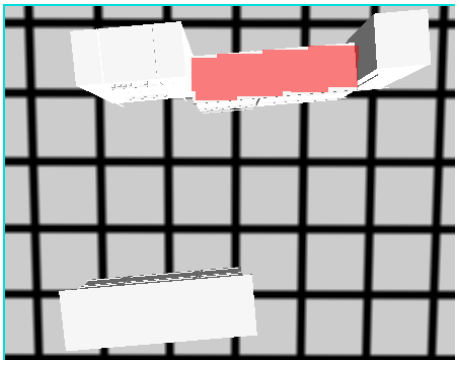
(a) The semantic map of the TUM kitchen lab.



(b) Density Map of the property (on counter—top)



(c) Density Map of the property (in drawer)



(d) Density Map of the properties (counter—top drawer) (name "Counter205")

Figure 4.5: *Different density maps generated from Knowrob's semantic map.*

density map entry that is inside the rectangle defined by the object's position and its size along x and y. For designators that contain a *name* property the density map is generated from only one object instance, but the underlying mechanism is the same. Figure 4.5 shows a few examples for different density maps and the corresponding designators.

4.1.1.3 Density Maps for visibility

Density Maps generated from existing maps such as an occupancy grid or the semantic map only reuse existing information. Visibilty is more complex because, in order to be accurate enough, the corresponding density map needs to take into account the position of the sensors relative to the robot, including its height, but also occlusions caused by other objects. Density Maps for visibility need to have non-zero values in cells from which a specific object or coordinate is visible. To solve this problem, CRAM uses OpenGL's depth maps. To generate a density map indicating poses from which a specific object, e.g a cup on the table, can be seen, the system places OpenGL's camera at the location of that cup. Then, four images are rendered with the camera pointing in four directions (0°, 90°, 180° and 270°). Rendering only includes objects that might be occluding the object of interest, i.e. the object itself and the robot are not rendered. The resulting depth maps are then translated into a density map. In other words, instead of checking visibility for all possible camera positions, the system solves the inverse problem, i.e. which areas around the object are occluded by clutter.

Figure 4.6 shows a visualization of the resulting depth maps where the different shades of gray indicate different distances from the object for which a visibility density map should be computed.

Based on the depth maps, a density map is generated. Since objects are visible only within a certain range, say two or three meters, the density map does not need to be extremely big. With a resolution of 5 centimeters per cell and a size of the density map of 5x5 meters, the overall size of the density map is 100x100 cells. The density map is positioned with the object for which visibility should be computed in the center. For the sake of simplicity, we define the coordinates of the object of interest to be $(0, 0)$. The top left corner of the density map has the coordinates $(-2.5, -2.5)$ and

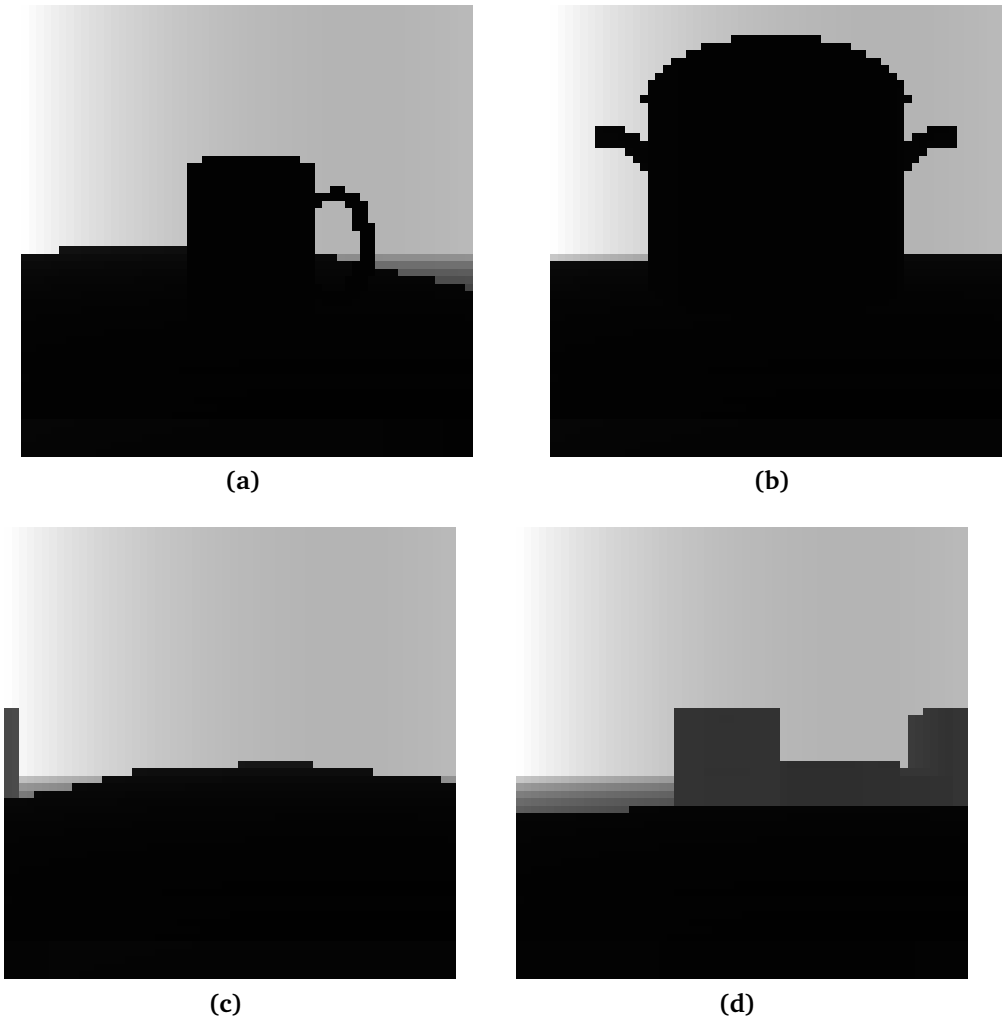


Figure 4.6: *OpenGL depth maps generated by rendering the scene from the location of one of the cups in four directions (0 degrees, 90 degrees, 180 degrees and 270 degrees).*

the bottom right corner has the coordinates $(2.5, 2.5)$. To compute the density of a cell, the system performs the following steps:

1. Render the scene in all four directions with a camera opening angle of 90 degrees. The resolution of the depth maps is the size of the density map, e.g. if it is 100x100 cells, the depth images will have the same size.

2. For each cell, compute which of the four rendered depth images to use.

3. Compute the indices of pixels to take into account. Robots such as the PR2 can move their torso up by about 30cm. That means a range of pixels has to be taken into account.

4. Compute the cell values by counting all cells with a depth value greater than the density map cell's distance from the center of the density map divided by all pixels in the depth map that are taken into account.

The density map generated from the depth images shown in Figure 4.6 can be seen in Figure 4.7.

Although the generation based on depth maps is relatively simple, the most tricky part is the computation of the row and the indices of the pixels in the depth map to take into account. Given the x- and y-coordinates of a cell in the density map, the corresponding depth map is selected by checking the angle between the cell and the origin (i.e. the center of the density map). If it is between $-45°$ and $45°$, the first depth map is used, if it is between $45°$ and $135°$, the second density map is used, etc. Since the density map size and the size of the depth images are identical, each cell in the density map can be computed from a subset of a single column in the depth map. The column is computed as follows:

$$c = \frac{x}{y} * \frac{\text{size}_x}{2}$$

The variable c specifies the column index, i.e. the x-coordinate of pixels in the depth map to take into account for computing a specific cell. Please note that x and y are not the coordinates of the density map cell but the corresponding coordinates mapped into the respective depth map. For instance, if we want to compute the column index

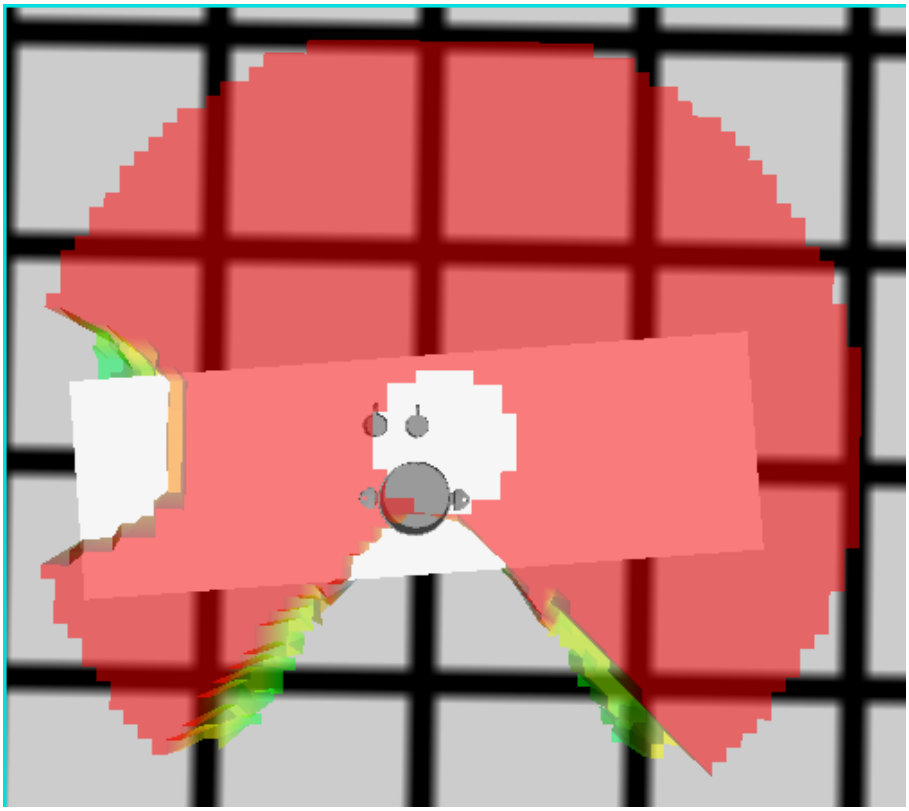


Figure 4.7: *Density Map generated from the depth maps shown in Figure 4.6.*

in depth map 1 (angle range 45° to 135°), the density map cells at (x, y) need to be mapped to $(y, \text{size}_x - x)$ when computing the column index.

Which pixels of a given row to take into account for computation of a density map value depends on the possible positions of the camera. If the camera is always at a fixed height on the robot, only one single value in the depth map has to be taken into account. However, if the camera can be moved up and down as it is the case for the PR2, several values might be included in the computation, depending on the distance of the density map cell from the origin (the closer it is the more pixels need to be considered). For density map cells that are most distant from the origin of the density map, one pixel in the depth map will correspond to the resolution of the density map since the size of the depth maps and the density map are set to be equal. If a density map pixel is closer to the origin, the number of depth map entries to take into account will change respectively. For instance, if the camera can be moved up and down between a height of h_{min} and h_{max}, the object is at a height of h_{obj} and the distance of the density map cell from the center of the density map is d, the minimum and maximum column index to use for calculating the density map value is computed as follows:

$$
\begin{aligned}
r_{min} &= \arctan\left(\frac{h_{min} - h_{obj}}{d}\right) * \text{size}_y \\
r_{max} &= \arctan\left(\frac{h_{max} - h_{obj}}{d}\right) * \text{size}_y
\end{aligned}
$$

Algorithm 2 shows the algorithm for computing visibility density maps. The resulting density map has its center in the center of the object (or location) for which the visibility density map should be generated.

Visibility density maps are currently only generated for the following designator properties:

- ((to see) (obj <object designator>)
- ((to see) (location <location designator>)

Algorithm 2 The algorithm for generating a visibility density map based on previously rendered depth maps in all 4 directions around the object of interest.

$Ds \leftarrow \{$*Rendered depth maps in all four directions*$\}$
$C \leftarrow$ *empty density map with size* $\left(size_x, size_y\right)$
for y from $-\frac{1}{2}size_y$ to $\frac{1}{2}size_y$ by *resolution* **do**
  for x from $-\frac{1}{2}size_x$ to $\frac{1}{2}size_x$ by *resolution* **do**
    $D \leftarrow SelectDepthMap(Ds, x, y)$
    $d \leftarrow \|(x, y)\|$
    $c \leftarrow ComputeColumn(x, y)$
    $r_{min} \leftarrow ComputeRow(d, h_{min} - h_{obj})$
    $r_{max} \leftarrow ComputeRow(d, h_{max} - h_{obj})$
    $v \leftarrow 0$
    for r from r_{min} to r_{max} **do**
      if $D(c, r) > d$ **then**
        $v \leftarrow v + 1$
      end if
    end for
    $C(x, y) \leftarrow v / (r_{max} - r_{min})$
  end for
end for
return C

It can be seen that the generated visibility density map as shown in Figure 4.7 has values of zero in its center around the object. The reason is that in this area, the minimum and maximum indices to be taken into account fall out of the generated depth images. Although this could be fixed easily by rendering a fifth depth image facing upwards, it is considered to not be a serious problem since the robot cannot be standing too close to the object anyway. The reason is that possible locations of the robot's base are limited by other factors such as the size of the robot's base and the table the object of interest is standing on.

4.1.1.4 Density Maps for reachability

Besides visibility, reachability is a second important aspect when generating locations for the robot to stand since most of the robot's actions are related to either picking up or putting down objects. Finding a location from which an object is reachable at least requires some sort of inverse kinematics computation to check if an arm configuration

for reaching the object exists at all. Classic approaches for finding such poses are the computation of reachability inside a limited area around the object of interest by subdividing it into cells and checking if there is a solution for inverse kinematics or the use of precalculated inverse reachability maps, for instance as shown in [Diankov, 2010].

Essentially, the problem that needs to be solved is to find a location for the robot's base given a pose in world coordinates the robot's arm should be moved to. If the pose is well-defined, i.e. a full six dimensional pose in space, the simplest solution is to compute inverse kinematics solutions for a complete area around the pose to reach until a solution for inverse kinematics could be found. This approach works fairly well if many valid solutions can be found but might become computationally relatively expensive if no solution is possible since all cells on a grid around the pose to reach has to be tried. However, reachability can be precalculated.

CRAM uses simplified reachability maps because, in order to satisfy the *reachable* predicate, the system does not do any grasp planning but uses only four predefined grasps. This restricts the number of required orientations in the reachability map and allows to generate smaller maps. More specifically, in the case of one top grasp, one front grasp and two side grasps, the reachability map is generated only for orientations around the Z axis for the side grasps and the top grasps. This is a too strong simplification for replacing inverse kinematics computations completely by reachability maps but reduces the space requirements drastically. Since the system is supposed to generate pose candidates that are later verified, this restriction is not a problem in our special case.

A density map for reachability should have values greater than zero for all density map cells from which a certain pose, point or object is reachable. To generate such a map, it requires a reachability map, i.e. a three-dimensional map where each cell stores with which orientations it is reachable by the robot, with the root link of the arm's kinematic chain in the map origin. From this map it is relatively simple to generate an inverse reachability map, i.e. a three-dimensional map that stores at which locations the root link of the kinematic chain can be placed to be able to reach the origin.

Generation of the reachability map. If we wanted to use reachability maps only for resolving the reachable predicate, generating it with only the four grasp orientations

would be sufficient. However, we want to generate an inverse reachability map for which we need more orientations. Thus, we compute the map for 16 orientations, 8 orientations for side grasps and 8 orientations for top grasps.

Algorithm 3 The algorithm for computing a reachability map.

```
Os ← {Orientations of 8 side grasps and 8 top grasps}
R ← Empty reachability map with size (size_x, size_y, size_z)
for z from −½size_z to ½size_z by resolution do
  for y from −½size_y to ½size_y by resolution do
    for x from −½size_x to ½size_x by resolution do
      for o in Os do
        if FindIKSolution((x, y, z), o4) then
          R((x, y, z), o) ← true
        else
          R((x, y, z), o) ← false
        end if
      end for
    end for
  end for
end for
return R
```

Given the set of orientation of the grasps and a (robot specific) size of the three dimensional map[1], the map is generated by simply iterating over all cells. For each cell, an inverse kinematics solver is called for each of the possible grasp orientations. If a solution can be found, the system marks the position of the cell with the corresponding orientation as reachable. Figure 4.8 shows the reachability maps for the PR2 for its both arms.

The algorithm for computing the reachability map for the later generation of an inverse reachability map can be seen in Algorithm 3.

Generation of the inverse reachability map. In order to generate a density map for reachability, the system needs to compute an *inverse reachability map*. As mentioned already, instead of marking which cells are reachable by the robot, the inverse reachability map marks from which cells the origin can be reached. To generate the inverse map, the system iterates over all cells of a three-dimensional grid around the origin. For each cell, it computes if the origin is reachable. The corresponding check

[1] For the PR2 $2m \times 3m \times 1.25m$.

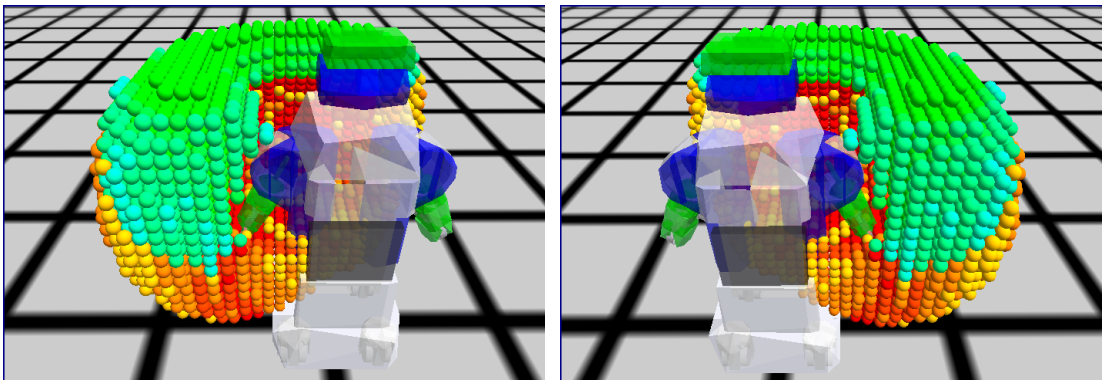


Figure 4.8: *Reachability maps for the PR2's two arms.*

uses the previously computed reachability map. Since the robot can be rotated, the check for the origin's reachability from a specific pose requires several look ups in the reachability map. Since our simplified map contains only a restricted set of rotations, to check if the origin is reachable from a specific pose, we need to perform one check per orientation. The index vector in the reachability map for reaching the origin from a pose P is the inverse of P. Thus, we can mark a point p in the inverse reachability map as valid if there exists (at least) one orientation o for which the corresponding entry in the reachability map indexed by the inverse of the pose constructed from point p and orientation o.

Algorithm 4 shows the algorithm for computing the inverse reachability map.

Figure 4.9 shows the inverse reachability map for reaching the origin with the PR2's left arm. As can be seen, the origin can be better reached from locations above the origin which is consistent with the generated reachability maps.

Generation of the reachability density map. The construction of a density map is, in the simplest case where a complete pose to be reached is specified, just copying the inverse reachability map and normalizing it. However, often, the system does not have knowledge about the actual object to grasp yet or does not have any information about the grasp to use. Therefore, the density map is generated based on how many orientations at a specific point are reachable. More specifically, the value of a density map cell is computed by counting how many orientations at the goal point are reachable if the robot would be standing at a specific density map cell and finally normalizing the value, i.e. dividing it by the overall number of possible orientations. Algorithm 5

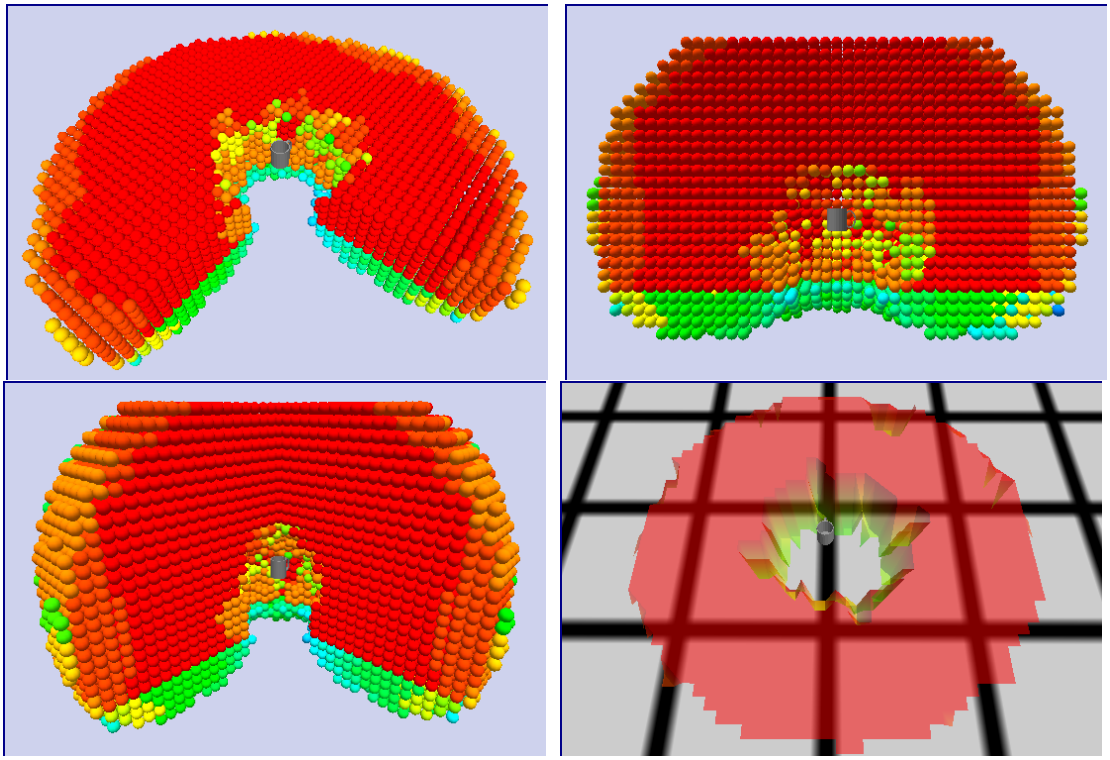


Figure 4.9: *Different visualizations of inverse reachability maps for the PR2's left arm. The colored spheres indicate locations for placing the root of the manipulator's kinematic chain from which the origin can be reached. A red color indicates that more orientations exist from which the origin can be reached, blue indicates less orientations.*

Algorithm 4 The algorithm for computing an inverse reachability map from a reachability map as generated by Algorithm 3.

$R \leftarrow$ *Reachability map*
$size \leftarrow \max\left(size_x, size_y\right)$
$I \leftarrow$ *Empty inverse reachability map with size* $(size, size, size_z)$
$Os \leftarrow$ *Orientations in* R
for z from $-\frac{1}{2}size_z$ to $\frac{1}{2}size_z$ by *resolution* **do**
  for y from $-\frac{1}{2}size_y$ to $\frac{1}{2}size_y$ by *resolution* **do**
    for x from $-\frac{1}{2}size_x$ to $\frac{1}{2}size_x$ by *resolution* **do**
      for o' in Os **do**
        if $\exists o \in Os\, R\left((-x,-y,-z), o^{-1}o'\right) =$ **true then**
          $I\left((x,y,z), o'\right) \leftarrow$ **true**
        else
          $I\left((x,y,z), o'\right) \leftarrow$ **false**
        end if
      end for
    end for
  end for
end for
return I

shows the algorithm for generating the reachability density map and the resulting reachability map for reaching a cup on the counter top is shown in Figure 4.10.

In the current implementation, reachability density maps are used for designator resolution if one of the following property combinations is present in a location designator:

- ((to reach) (obj <object–designator>)∗)
- ((to reach) (location <object–designator>)∗)
- ((to execute) (action <manipulation action designator>)∗)

All versions allow for the specification of multiple objects, locations or action designators for reaching multiple destination poses. The resulting pose will be a pose from which all objects or poses are reachable or from which all actions can be executed. In case of a location to execute a specific action, several key trajectory points might be considered for the generation of the density map. For instance, for opening a drawer, these key points are the start and the end pose of the gripper to open the drawer.

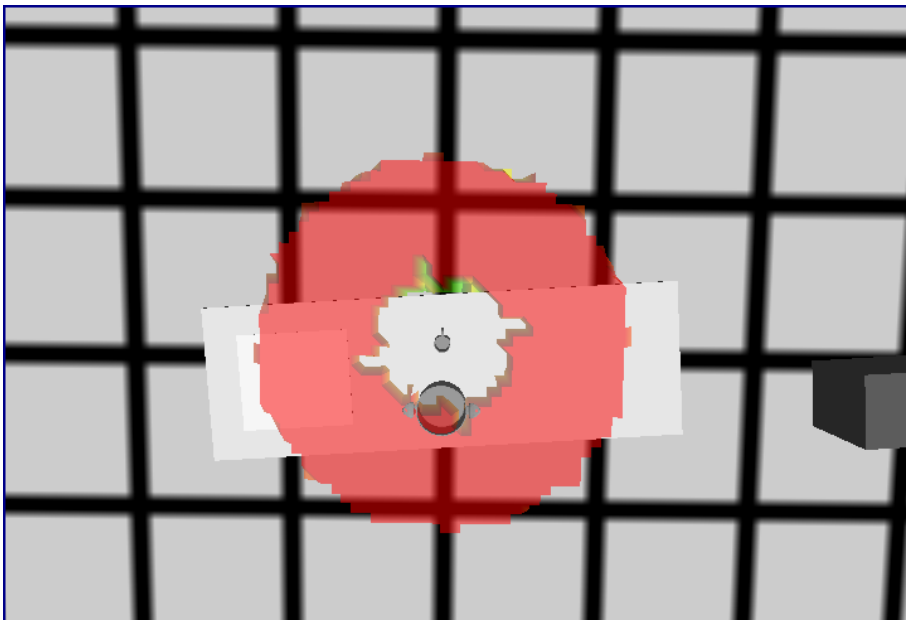


Figure 4.10: *Generated reachability density map for reaching the cup on the counter top.*

Algorithm 5 The algorithm for generating a reachability density map as shown in Figure 4.10. The algorithm uses an inverse reachability map generated with Algorithm 4.

$I \leftarrow$ *Inverse reachability map*
$C \leftarrow$ *Empty density map*
$z \leftarrow z_{obj} - z_{cm}$
$Os \leftarrow$ *Valid goal orientations*
for y from $-\frac{1}{2}size_y$ to $\frac{1}{2}size_y$ by *resolution* **do**
  for x from $-\frac{1}{2}size_x$ to $\frac{1}{2}size_x$ by *resolution* **do**
    for o in Os **do**
      if $I((x,y,z),o) =$ **true then**
        $C(x,y) \leftarrow C(x,z)+1$
      end if
    end for
    $C(x,y) \leftarrow C(x,y)/length(Os)$
  end for
end for
return C

For picking up an object, only one key point is defined, the end point of the reaching trajectory. The system uses the predicate trajectory–point with the following signature to query all required trajectory points of a specific action:

```
(trajectory-point ?action-designator ?pose ?arm)
```

The predicate holds for each pose for a specific arm given an action designator. The system then collects all solutions for a specific action designator and combines the resulting reachability maps to find locations from which all points are reachable.

4.1.1.5 Spatial relations

In particular when using information intended to be understood by humans, for instance when using the world wide web for plan generation, locations are often specified using spatial relations such as *"left of"*, *"right of"*, *"near"* or *"behind"*. The mechanism for generating locations based on density maps is flexible and powerful enough to allow for the resolution of such spatial relations. The basic idea is to implement context specific density maps based on the spatial relation to be resolved and combine them.

Spatial relations are expressed as designator properties. In its current state, CRAM supports the following set of relation specifiers:

- (left–of <obj>)
- (right–of <obj>)
- (in–front–of <obj>)
- (behind <obj>)
- (near <obj>)
- (far–from <obj>)

As can be seen, all designator properties related to spatial relations require a reference object as parameter. These designator properties can be combined to define, for instance, relations such as *"between"* (right of one object and left of another object) or *"right of and behind"*. For instance, a location designator for the location of a mug in a table setting scenario could be defined as follows:

```
((at place-1) (for mug) (right-of plate) (behind plate)
 (context table-setting))
```

A location for a knife in the same table setting scenario can be specified as follows respectively:

```
((at place-1) (for knife) (right-of plate)
 (context table-setting))
```

The result of designator resolution strongly depends on context. For instance, when setting a table, the viewpoint is defined by the table and the seating location at the table while a command such as "grasp the mug right of the plate" is either relative to the location of the human giving the command or the robot itself.

Density Maps for directional spatial relations. Directional spatial relations supported by CRAM are, as mentioned before, *"left of"*, *"right of"*, *"in front of"* and *"behind"*. To be resolved correctly, all relations require a reference object and a target object. In the context of table setting, one way for inferring the reference coordinate system for directional spatial relations is to use the supporting object (most often a counter or a table). More specifically, humans tend to use well-defined locations for placing the objects involved in table setting and spatial relations are relative to the location where a human would sit at the table. A good heuristic for defining the reference coordinate system is to use the closest edge of the supporting object. Figure 4.11

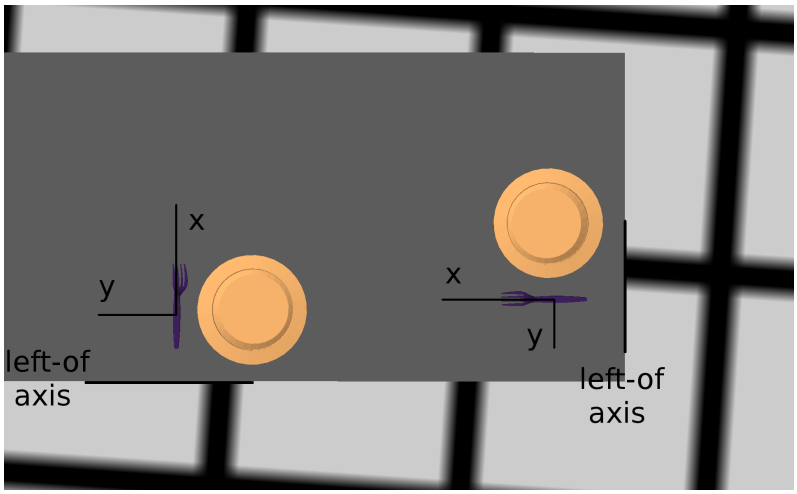


Figure 4.11: *Different coordinate systems inferred based on the closed edge on the supporting object.*

shows two examples for different reference coordinate systems inferred based on the supporting object.

In the reference coordinate system, we can define four density maps, one for each directional relation. The density maps are inspired by membership functions of fuzzy sets as presented in [Dutta, 1989]. The basic idea is to construct a density map that contains high values for locations close to the exact spatial relation. For instance, for *"left-of"*, all locations that exactly satisfy the relation are along a vector parallel to the y-axis originating in the center of the reference object. Additionally, the distance from the reference object is taken into account. The further away a location is from the reference object, the bigger is the allowed deviation from the reference vector. Values that are on the wrong side of the reference object are set to zero in the density map. A density map value for a point o in the density map is computed as follows.

- Transform the point into the reference coordinate system with the reference object in its origin and the coordinate axis aligned as shown in 4.11.
- Compute the ratio of either the x or y component of the transformed point to its length and take the absolute value. For relations left of or right of, the y component is taken, for behind and in front of, the x axis is considered.
- Set all values on the wrong side of the reference object to zero. For relations left of and behind, the y or x component of the vector must be greater than zero to

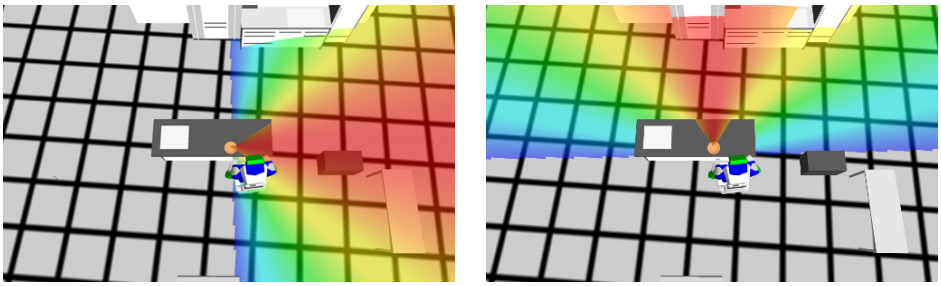


Figure 4.12: *Density Maps for resolving the spatial relations "right-of" and "behind".*

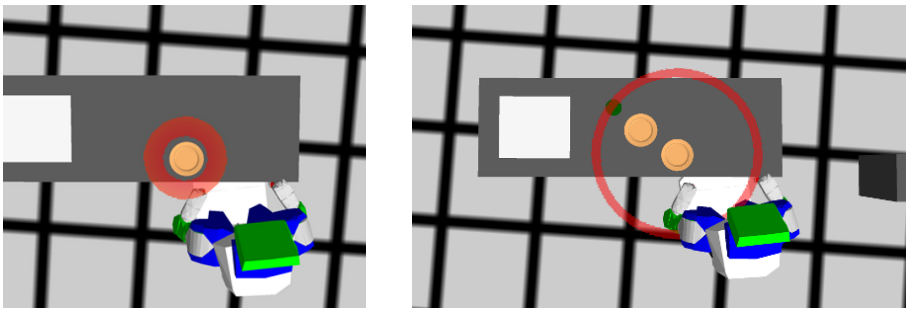
be on the right side and for right of and in front of they must be smaller than zero respectively.

Algorithm 6 shows the corresponding computational steps.

Algorithm 6 The algorithm for generating density maps for the directional spatial relations *"left-of"*, *"right-of"*, *"behind"* and *"in-front-of"*.

$T \leftarrow$ *Transform into the reference coordinate system*
$p' \leftarrow Tp$
if Relation $=$ *"left-of"* **then**
  if $p'_y > 0$ **then**
    return $p'_y / |p'|$
  end if
else if Relation $=$ *"right-of"* **then**
  if $p'_y < 0$ **then**
    return $-p'_y / |p'|$
  end if
else if Relation $=$ *"behind"* **then**
  if $p'_x > 0$ **then**
    return $p'_x / |p'|$
  end if
else if Relation $=$ *"in-front-of"* **then**
  if $p'_x < 0$ **then**
    return $-p'_x / |p'|$
  end if
end if
return 0

The above steps yield values between 0 and 1 with values directly along the reference axis set to 1. Figure 4.12 shows to examples of such density maps.



(a) (b)

Figure 4.13: *Density Maps for resolving the "near" and the "far" relations.*

Density Maps for distance relations. CRAM supports two distance relations, *far* and *near* which are both based on relatively simple heuristics. To find locations that are near a reference object, a threshold for the maximal allowed distance is inferred from the context and the objects involved. The density map for the *"near"* relation is then assembled from the following two density maps:

- A density map with values of 1 for all locations closer than the threshold to the reference object.
- A Gaussian with its center on the reference object to bias the sampling algorithm in order to prefer locations closer to the reference object.

Additionally, locations directly on the reference object are cut out, i.e. set to zero since these locations would cause a collision between the reference object and the object to be placed. Figure 4.13a shows a density map representing the *"near"* relation.

The *"far"* relation is resolved in a similar way. Locations are considered to be far from a reference object if they are further away than a specific threshold but closer than another threshold. Figure 4.13b shows an example for a density map to generate poses far from one of the plates.

4.2 Generation of Height and Orientation Values

Density Maps provide a two-dimensional grid with information about the validity of the corresponding point as a designator solution. However, designator solutions are

poses, i.e. they consist of a three dimensional point and an orientation. The information that is missing from pure density maps, i.e. the Z coordinate of a point and the orientation of the solution are generated using separate mechanisms that are either based on heuristics, on sampling or a combination of both.

Generation of height values. In our scenario, locations for the robot's base are generally on the floor, i.e. they need to have a Z value of zero. Locations for objects on the other hand are normally on supporting planes such as on the bottom board of a drawer or a counter top. When stacking objects, the Z coordinate is normally determined by the upper boundary of the supporting object's bounding box. If multiple solutions are possible, for instance if searching for locations in drawers without specifying the drawer instance to use, the height value is randomly chosen from the list of possible height values. To summarize, in the current implementation, height values, i.e. the Z coordinates of the generated poses, are set to zero for all poses for the robot, i.e. for poses *to see*, *to reach* or *to execute* and generated from the upper boundary of the bounding box of the supporting object for poses *on* or *in* other objects, including semantic map objects.

Generation of orientation values. Generation of orientations is not only based on the location sample generated from density maps and the height value generator but also on the context and the objects involved.

For orientations to see or to reach, the system generates a sequence of orientations and lets the verification step of location designator resolution select an appropriate orientation. One reason for this approach is that when combining multiple reachability density maps, for instance when searching for a location from which several objects can be reached, the best orientation for the robot's base is different from the optimal orientation when grasping a single object. A second reason is that if the object is standing relatively far away from the boundary of its supporting table, it might be necessary to turn the robot's base to move the robot's arm closer to the object. An optimal solution in this case might require relatively expensive planning and exact geometric knowledge about the shape of the supporting table which might not be available if only 3D point cloud data can be used.

To generate multiple orientation candidates, the system first computes the angle between the X axis originating in the point generated by sampling from the density map

and the vector from that point to the object of interest. Based on this reference orientation, a fixed number of orientations are generated that range from the reference angle minus a certain value to the reference angle plus a certain value. The current system generates five different orientations from $-25°$ to $25°$ around the reference angle.

The generation of object orientations not only depends on the objects to be placed but also on context. We distinguish between rotationally symmetric (or almost symmetric) objects such as plates, glasses but also mugs in most cases. For these objects, orientations do not really matter and thus the generated orientation is set to identity. Please note that in cases of objects for which orientation does not matter, the robot's put-down routines are free to ignore the orientation of the put down location completely. That means that in most cases related to these objects, the orientation is ignored by most robot components anyway. For other objects such as silverware, orientation does matter in many cases but is highly context specific. For instance, in the context of clean up tasks where the silverware is put in the correct drawer, the orientation is defined by the orientation of the drawer. In case of table setting tasks, the orientation is determined by aligning the object with the supporting entity, too. However, since different place settings for different seating locations are required, the orientation of the place setting is used. If the context indicates that the object just needs to be put down to a temporary location and picked up later again, the orientation does not matter. Thus, in these cases, a random orientation is chosen.

4.3 The Density Map Generator API and Implementation Details

CRAM provides a library for generating density maps and for integrating them into location designator resolution. Location designators provide the functionality to register generator and validation functions. The CRAM density map library registers a generator function with the following functionality:

- Use the CRAM Prolog interpreter and the predicate desig−density−map to generate an instance of the class location−density−map given the designator to be resolved.

- Select a sampling algorithm, e.g. random sampling by interpreting the density map as a probability density function or by taking entries with the greatest values first.
- Generate a lazy list of poses by drawing samples from the density map using the sampling algorithm and using the height and orientation generators.

On the lowest level, the library provides a class for representing density maps and the required generators and for computing the actual density map. On a higher level, the user is supplied with a Prolog API that allows for integrating density maps with the resolution mechanism of designators. This section will describe the two APIs and show how they are implemented and how they can be used.

4.3.1 The Internal Representation of Density Maps and Generators

Class 13 gives an overview of the slots of the class location−density−map which is the central data structure for representing density maps and for generating samples. As can be seen, the class inherits from density−map−metadata which provides slots for storing the density map's width, height, its origin i.e. the coordinates of its upper left corner, and its resolution i.e. the size of each density map cell.

The density map API provides a few methods for registering generator functions, for computing the density map, for accessing it and for generating samples. The following list gives an overview of the most important API functions:

register−cost−function . Registers a density map generator function under a given name. A generator function can either be a function object, e.g. a lambda function or a reference to a function obtained by the #' reader macro, or an instance of the class density−map−generator. This class can be inherited to allow the user to apply different density map generation strategies. For instance, in case a normal function object is passed to register−cost−function , it is wrapped into an instance of function−density−map−generator which is a class that inherits from density−map−generator. The function must accept exactly two parameters, the x and y coordinate of a density map cell and return a positive floating point number representing the density value of the cell. Calling a

Class 13 The class location−density−map. It contains all data required for generating three-dimensional poses based on the density map and for combining multiple density maps.

class LOCATION-DENSITY-MAP (**superclasses:** density-map-metadata)

slot DENSITY-MAP: Two-dimensional matrix for storing the generated density map.

slot DENSITY-MAP-GENERATORS: A sequence of generators. They are called sequentially and their output is combined to compute the overall density map.

slot HEIGHT-GENERATORS: A sequence of generator functions that compute the Z coordinate given a two-dimensional point generated by sampling from the density map.

slot ORIENTATION-GENERATORS: A sequence of orientation generators that return the orientation of the sampled pose given the three dimensional point generated by sampling and the height generator.

function for each cell can be rather expensive. Therefore, the system provides a second generator class, map−density−map−generator. Instead of calling a function once per cell, the generator function gets two parameters, the density map metadata with its size, origin and resolution, and a two dimensional matrix with the size of the density map. The generator function must then return a new matrix that is the input matrix plus the newly generated density values. The name can be an arbitrary Common Lisp object, e.g. a symbol, a string or an instance of a class. It is used to detect and remove duplicates, i.e. it must be specific and unique. For each name, the method density−map−generator−name−>score must be defined that returns a numeric value. These values are used to order the generator functions before executing them, greater scores result in earlier execution. This allows to utilize the fact that if a density map cell is already zero, no generator functions have to be executed anymore since zero values always stay zero.

register−height−generator . Registers a height generator in a density map instance. If multiple generators were registered, they are executed in the order of their definition. Each height generator receives an X and Y coordinate and returns a sequence of numbers representing different possibilities for Z coordinates of a specific point. All returned lists are concatenated and for computing the Z value of a specific point, one of the possible Z values is chosen randomly.

register–orientation–generator . Registers an orientation generator. Orientation generators are handled slightly differently to height generators. Each generator function gets the point for which the orientation should be generated and the result of the previous orientation generator. For the first orientation generator this parameter is set to NIL. The generator can either extend the list of possible orientations returned by the previously executed generator or ignore it and return a completely new list. One sample for each returned orientation is generated, i.e. instead of randomly choosing one orientation, different poses for all orientations are generated. This allows the verification step of designator resolution to select the correct orientation if the orientation generator cannot compute it yet.

get–density–map. This method generates (if not generated yet) and returns the density map. It iterates over all generator functions and calls them. Finally, it normalizes the density map by computing the sum of all elements and dividing each density map cell by that sum. That way, the overall sum of all density map cells is one, i.e. the density map is a valid probability density function.

gen–density–map–sample–point. Generates a single point by first calling a sampling function that uses the density map to generate an X and Y coordinate for the point. Then it calls the height generator with the two values and randomly chooses the Z coordinate from its result.

density–map–samples. Returns a (lazy) list of density map samples, i.e. complete poses, containing all possible samples in random order. For each lazy list entry, the method first generates a point using the method gen–density map–sample–point and then calls the orientation generator to get full poses.

4.3.2 Prolog API for Integration in Designator Resolution

The API for defining density maps and using them for designator resolution is completely based on predicates in CRAM's Prolog interpreter. To use density maps for generating location designator solutions, the user has to provide a version of the predicate desig–density–map that, if it holds, binds a variable to a new density map. The predicate has the following signature:

```
(desig-density-map ?designator ?density-map)
```

The variable ?designator is always bound to the designator to be resolved while the variable ?density-map is initially unbound and the binding has to be established by the predicate if the corresponding implementation can generate a density map for the designator.

To create a new density map or to verify that a variable is bound to a density map, the system provides the predicate density-map with the following signature:

```
(density-map ?density-map)
```

To add density map generators, the system provides the predicate ?density-map-add-function with the following signature:

```
(density-map-add-function
    ?generator-name <generator-expression> ?density-map)
```

The variable ?generator-name must be a valid name as required by the method register-cost-function, i.e. the method density-map-generator-name->score must be defined for it. The expression <generator-expression> must be a valid Lisp S-Expression that is evaluated in Lisp. It must return a valid density function, i.e. either a function object or an instance of density-map-generator which the predicate registers in the density map bound to the variable ?density-map.

For height and orientation generators, the predicates density-map-add-height-generator and density-map-add-orientation-generator are defined. They have a similar signature as density-map-add-function:

```
(density-map-add-height-generator <generator-expression>
    ?density-map)
(density-map-add-orientation-generator <generator-expression>
    ?density-map)
```

Please note that no name has to be specified for height and orientation generators since their evaluation order does not need to be customized.

Listing 4.1: *Definition of the predicate* desig-density-map *that matches designators* to see *an object and restricts the generated poses to locations closer than two meters to the object.*

```
1  (<- (desig-density-map ?designator ?density-map)
2    (desig-prop ?designator (to see))
3    (desig-prop ?designator (obj ?object))
4    (density-map ?density-map)
5    (pose ?object ?object-pose)
6    (density-map-add-function
7      2m-range (make-range-cost-function ?object-pose 2.0)
8      ?density-map)
9    (density-map-add-orientation-generator
10     (make-angle-to-point-generator ?object-pose)
11     ?density-map))
```

Listing 4.1 shows the definition of a density map that only includes locations that are closer than two meters to an object for location designators to see the object. First, the predicate checks if the designator is a designator to see a specific object and binds the object designator to a variable. Then it asserts that the variable ?density-map must be a density map which might create a new instance of type location-density-map. Then, the pose of the object is bound to the variable ?object-pose and the generator function is added. The Lisp function make-range-cost-function is a function that returns a callable object for setting only values that are closer than a certain threshold to a point or pose. It is a utility function also provided by the location density map library.

Finally, an orientation generator is added that generates an orientation for the robot's base to directly face the object.

4.3.3 Integration in Designator Resolution

As mentioned already, the location density map library integrates with the location designator resolution mechanism. This integration is based on a generator function that first uses Prolog to get a density map and then uses it to generate samples that are solution candidates for the location designator. More specifically, the generator function first executes the Prolog interpreter to get a density map. The Prolog call is as follows:

```
1 (prolog `(merged-desig-density-map ,designator ?density-map))
```

The predicate merged-desig-density-map first queries all solutions for the desig-density-map predicate and then merges all resulting density maps into one. The reason for this being necessary is that several versions of the predicate desig-density-map might match the designator, i.e. several solutions are generated that need to be merged. Merging density maps basically concatenates the lists of all generator functions and the lists of all orientation and height generators.

Finally, the location designator generator function returns the lazy list of density map samples returned by the density-map-samples method.

In addition to the generator function, the system also registers a validation function in order to allow the user to verify if a certain pose would be a valid solution. The implementation just checks if the density map value that corresponds to a specific point is greater than zero which means that the probability for the point to be drawn as a density map sample is also greater than zero.

4.4 Related Work

Visibility constraints are used by different planning systems such as Move3D [Siméon et al., 2001] and OpenRAVE [Diankov and Kuffner, 2008]. These systems are optimized for motion planning and visibility constraints can only be verified given a specific camera pose. The generation of probability distributions that can for instance be used by RRT planners can thus become computationally expensive since a lot of poses have to be verified for visibility. In contrast, CRAM solves the inverse problem: it generates all locations that are visible if the camera was placed at the location of the object of interest. Thus, a much lower number of rays has to be generated. Our approach is further optimized by utilizing OpenGL on modern graphics hardware.

Inverse reachability maps are also used by planning systems to find locations from which a specific grasp can be executed [Zacharias et al., 2007, Diankov, 2010]. The process of generating a reachability map is computationally expensive but can be performed offline. The reachability map generated and used by CRAM is very similar to OpenRAVE although CRAM's map is a lot simpler since only a low number of grasps is used to approximate reachability.

Spatial relations are particularly important for interpreting natural language specifications of actions. Resolving spatial relations to find a specified object is a well studied field [Jungert, 1993, Clementini et al., 1997, Aiello et al., 2007]. However, for using spatial relations to parameterize plans, the inverse problem has to be solved, i.e. instead of finding a specific object left of another object, a location for the object has to be found to place it left of the other object. One approach to solve this problem is presented in [Bloch, 2006] where the authors use fuzzy set theory. However, in contrast to CRAM, the specific context of an action and knowledge about the objects involved is not taken into account.

4.5 Discussion

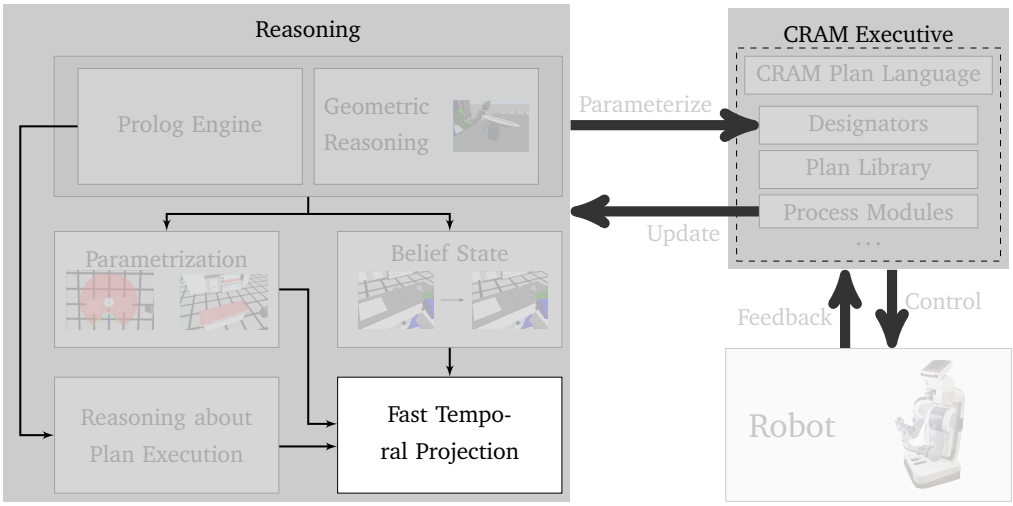
The work presented in this section shows an integrated framework for combining different algorithms to generate locations for the robot's base given a number of constraints in order to execute specific actions such as grasping objects or just finding them in the robot's sensors. Although most algorithms for resolving symbolic plan

parameters in CRAM are related to existing work, CRAM additionally integrates symbolic reasoning and thus allows to incorporate additional knowledge in the process of finding valid solutions. The use of density maps to generate solution candidates allows to easily combine algorithms of completely different nature into a single framework and the use of a verification step after sampling solution candidates allows for the integration of decision mechanisms that cannot generate solution candidates but only decide on the validity of a solution.

Drawbacks of this approach are that while it is highly general, it does not lead to provably optimal solutions in most cases. The generation of visibility density maps does not take into account the shape of the object of interest and thus is not completely accurate for obstacles that are close to the object and for complex shapes that are not cyllindrical, e.g. a knife. Reachability is only considered for a low number of grasps which means that actually valid locations might be rejected. However, experiments on in simulation and on TUM-Rosie and TUM-James prove the applicability of our approach.

Chapter 5

Temporal Projection of Plans



In order to reliably perform actions in dynamic environments, for instance setting a table, the success of individual plan steps depends on the order in which actions are executed and the parameters of previously executed actions. For instance, a previous put down action can cause later pick up actions to fail because the put down location of one object can cause it to occlude the object to be picked up or an object is blocking pick up or put down trajectories of later actions. For instance, when setting a table, the plate must be placed first because placing the silverware first would cause collisions between the robot's grippers and the silverware when putting down the plate.

Often, symbolic planners are used to find a partial ordering of actions. However, most symbolic planners are too coarse to be able to deal with the geometric constraints present in domains such as a human household. Motion planning on the other hand

is fine grained enough to deal with geometric problems, it is computationally very expensive though.

In contrast, CRAM plans are control programs that are carefully hand crafted, however they are not less flexible than plans generated by symbolic planners. In fact, by integrating a geometric reasoning and plan parametrization system as introduced in Chapter 3 and Chapter 4, the system is able to robustly execute single pick-and-place actions. To handle interferences between different pick-and-place actions, CRAM provides a powerful and fast temporal projection mechanism that is based on the world state representation and reasoning system that was introduced in Chapter 3. Temporal projection is integrated through the resolution of location designators which takes into account future actions to be executed. The implementation of projection is based on repeatedly executing the top level plan that is currently executed on the robot in projection mode and recording and storing time lines for each execution episode. Projection is an optional module that runs completely in parallel to plan execution on the robot, normally using a thread pool.

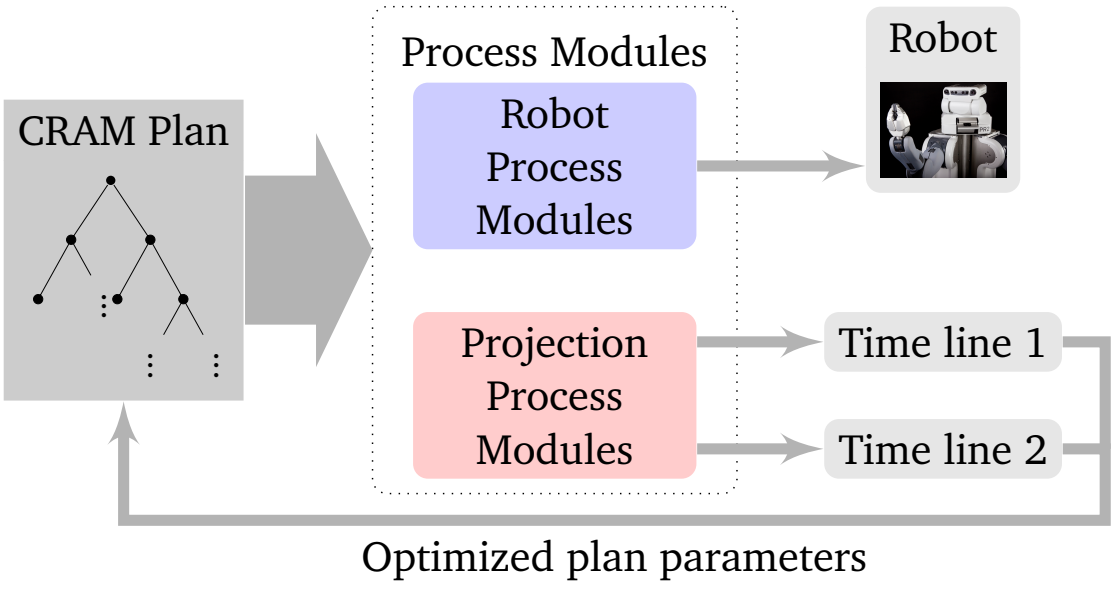


Figure 5.1: *System overview of plan projection for designator resolution.*

In order to use these projected time lines for choosing a solution for location designators, the CRAM projection module hooks into the with-designators macro. Given a set of time lines, ordered according to an objective function based on the number

and severity of flaws on the time line, the system uses the current task path to get the location designator used in the best time line and re-uses it in the plan that is executed on the robot. Figure 5.1 gives an overview of the integration of projection in the designator resolution mechanism.

The work presented in this Chapter has been published previously by the author in [Mösenlechner and Beetz, 2013], in [Mösenlechner and Beetz, 2011] and in [Mösenlechner and Beetz, 2009].

5.1 Projection of Plans and Generation of Timelines

To project a plan, it is executed in a projection environment. The most important difference compared to executing it on an actual robot is that a different set of process modules is used. Additionally, the execution context represented by dynamically bound variables that keep data such as the current belief state or the current time line is replicated in the projection environment.

In order to implement a working projection mechanism, the user needs to define a projection environment. It consists of the process modules used in a plan, usually one for manipulation, one for navigation, one for perception and one for moving the robot's sensors, i.e. its pan tilt unit, and a set of global variables holding the state of plan execution.

5.1.1 The Projection Environment

To create a new projection environment, CRAM provides the macro `define-projection-environment`. It is defined as follows:

```
(define-projection-environment name
    &key
      special-variable-initializers
      process-module-definitions
      startup
      shutdown)
```

The system supports multiple projection environments that can be referenced by a name, hence the name parameter. Additionally, the macro accepts four parameters with the following meaning:

- :special-variable-initializers A list with a similar form as Common Lisp's let, i.e. a list of pairs of the form (name value). The names must be the names of special (i.e. global) variables and the values specify how these values are initialized. For instance, the world database object used by the physics-based reasoning engine of Chapter 3 is copied here.
- :process-module-definitions A list of process module names to be started before executing plans in this projection environment. In the common case, it is a list of four symbols naming the four process modules used in CRAM's plan library.
- :startup A Lisp form that is executed after the special variables are initialized but before the actual plan is executed. Additional setup steps can be performed here.
- :shutdown A Lisp form that is executed right after plan execution has finished. All special variable bindings of the projection environment are still valid in the context of the shutdown form.

In order to execute code within a projection environment, CRAM provides the macro with-projection-environment with the following signature:

```
(with-projection-environment name &body body)
```

When evaluated, the macro first establishes the projection environment matching name by binding the special variables according to the initializers. Then, the defined process modules are started, the startup form is executed and the body forms are executed in the context of the projection environment. Finally, the shutdown form is evaluated. The result of the form contains the result of the body form and the bindings of the special variables as they were when the body finished executing. This result is then stored for later evaluation by the projection subsystem since it contains the timeline recorded during projection, the execution trace as well as the actual result of projection.

5.1.2 Process Modules for Projection

Process modules for projection can be implemented in many different ways, for instance on a purely symbolic level similar to STRIPS-like planners or an accurate physics simulation such as Gazebo. This implementation of course influences the performance of projection but also its accuracy. In order to be able to project in parallel to actual plan execution, projection needs to be significantly faster than execution on the robot though. Hence, complete physics simulation including the dynamics and control loops of the robot is not applicable in many cases. On the other hand, a purely symbolic projection approach as presented in the work of McDermott [Hanks and McDermott, 1987] and Beetz [Beetz, 2000] abstracts away from the geometric properties that severely influence the success of plans in a domestic environment. As a compromise, the CRAM system provides an implementation of projection process modules for the PR2 that is based on the physics-based reasoning system introduced in Chapter 3. This allows to accurately reason about occlusions, reachability, stability and blocking objects while ignoring dynamic aspects of plan execution such as the behavior of controllers or trajectories the robot follows. In other words, the currently implemented process modules for projection do not abstract away from aspects important for decision making in a high-level plan while ignoring lower level aspects.

As mentioned already, all process modules use the world database introduced in Chapter 3 for representing the current state of projection. Basically, all process modules just change the robot's state by making assertions in the world state. Instead of simulating

a trajectory, the robot or its link poses make discrete transitions. Hence, this approach assumes trajectory execution and navigation to work relatively robust.

The navigation process module. The navigation process module receives an action designator of the form:

```
((type navigation) (goal <loc>))
```

The goal <loc> is a location designator for the robot base. The implementation of the process module resolves this location designator to attain a three dimensional pose. Then, this pose is used to assert the pose of the robot object in the world database before emitting a robot–state–changed event.

The PTU process module. The purpose of the pan-tilt process module is to move the robot's head and point its sensors to a specific pose. The following example shows a corresponding action designator to parametrize it:

```
((type trajectory) (to see) (location <loc>))
```

The location must be a valid location designator and the process module is supposed to move the pan and tilt joints of the robot's head to point the sensor frames on its solution. To compute the joint angles, the correct solution would be to use an inverse kinematics solver that uses the main axis of the sensor as, for instance, shown in [Dianko v, 2010]. However, for the sake of simplicity, the current implementation of the PTU process module assumes that the sensor axis are sufficiently close to the main axis of the pan-tilt unit and the opening angle of the sensors is wide enough. Given the object location P in the coordinate frame of the pan tilt unit, the current angle

of the pan joint ϕ and the current angle of the tilt joint θ, the new angles for these joints ϕ' and θ' can be computed as follows:

$$\begin{aligned} \phi' &= \phi + \arctan\left(P_y,\ P_x\right) \\ \theta' &= \theta + \arctan\left(-P_z,\ P_x^2 + P_y^2\right) \end{aligned}$$

After computing the joint angles, the process modules just asserts them in the current world database and emits a robot—state—changed event.

The perception process module. The projection of the behavior of perception requires to approximate the behavior of the actual perception algorithms used on the robot. The current implementation is rather rudimentary and assumes perception with no false positives or negatives. The only constraint that is taken into account is visibility. Perception process modules receive action designators of the following form:

```
((to perceive) (obj <object-designator>))
```

The properties used in the object designators bound to the obj depend on the actual perception routines that are available. However, they need to at least provide the property type for specifying the class of the object that is to be detected. The following example shows an object designator that can be used in perception projection:

```
((type mug) (at <location-designator>))
```

Although the location designator is not required it can be used to restrict the search space of the object. If not present, KNOWROB is queried for the most probable storage locations for the corresponding object type.

Listing 5.1: *Prolog code used in the perception projection process module. Given an input object designator, in infers all matching instances in the world database.*

```
1 (<- (object-perception ?designator ?object)
2   (desig-prop ?designator (type ?type))
3   (world ?world)
4   (robot ?robot)
5   (object-type ?world ?object ?type)
6   (visible ?world ?robot ?object)
7   (-> (desig-prop ?designator (at ?location))
8       (location ?world ?object ?location)
9       (true)))
```

Projection of perception uses the type property together with the Prolog reasoning engine to infer all known object instances in the world database that match the corresponding type. Then, the visible predicate is used to infer all visible objects. Finally, if a location designator was specified using the at property, the system verifies that the visible object is at the correct search location by calling the location designator's validation function with the pose of the object. Listing 5.1 shows the corresponding Prolog code. The predicate object-perception holds for each visible object that matches the input object designator. The process module then generates a new object designator for each solution and returns a list of the newly created designators.

The manipulation process module. Projection of manipulation is the most critical part since pick-and-place actions mostly fail because of failed grasp actions. The process module needs to be able to handle actions such as pick-up, lift, carry and put-down. Similar to the navigation and PTU process modules, the manipulation process module asserts joint states for specific key points of the action and emits robot-state-changed events. The process module re-uses the same key-points that are used for the generation of reachability density map as explained in Section 4.1.1.4. To find all points to reach in order to execute a given action designator, the process module uses the predicate trajectory-point to. It asserts the joint states of an inverse kinematics solution to reach each of the positions and one robot-state-changed event is

generated per trajectory point. Besides robot–state–changed, the process modules generates object–attached events for grasping actions, object–detached events when objects are put down and the grippers are opened and object–articulation–event instances for opening and closing actions.

5.2 Handling of Time in Projection

In its simplest form, classical planning does not require a notion of time. Ordering is achieved by pre- and post-conditions. On the other hand, CRAM plans are just normal robot control programs with semantic annotations grounded in an underlying first-order representation. These plans can be highly concurrent and trigger actions on the robot asynchronously. In order to correctly project the execution of such parallel actions, the process modules used for projection need to explicitly advance a time line and block the resources controlled by the process module for a specific amount of time. The duration of actions is either a fixed constant that approximates the duration of the action on the robot (e.g. for the perception process module or the PTU process module) or based on heuristics. For instance, the duration of the navigation process module is approximated by the distance to travel. Although not implemented yet, these durations could also be learned, for instance based on data extracted from previously recorded execution traces.

We define a projection clock as a data structure that implements the following two methods:

- clock–time Return the current time of the clock.
- clock–wait Sleep for a specific amount of time and advance the clock.

The trivial implementation of a clock would simply use the current system clock and Common Lisp's standard sleep function. However, this would cause projections to take as long as the actual execution of plans which would render it almost useless. Linear scaling of time is one solution. For instance, the projection clock can be just sped up by a specific (constant) factor. However, choosing the constant can be rather hard. On the one hand, the projection clock must not be sped up by too much. The reason is that in order to be able to project concurrent control programs, sleeps must be long

enough to allow for task switches and scheduling. On the other hand, the speed up of the projection clock must be sufficiently high in order to allow for a high number of projection runs in parallel to actual plan execution.

CRAM's projection mechanism uses a non-linear clock as a compromise between maximal projection performance and support for concurrent actions. The basic idea is that the clock is incremented at a specific constant rate (per default 20Hz in the current implementation), while the time increment, i.e. the advancement on the timeline, varies depending on the action. More specifically, each call to the clock–wait method will block at least 20ms. First, clock–wait enqueues the duration in an internal sorted expiration time queue with the shortest duration first. Then, it sleeps for 20ms. Then, the internal clock (as returned by the clock–time method) is incremented by the shortest duration and clock–wait checks if its duration as specified in its parameter is now expired. If not, it sleeps for another 20ms until the duration expired.

For instance, let us assume that two actions, a pick-up action with the right arm and a movement with the pan-tilt-unit should be projected. Both actions run in parallel and the pick-up action should take 30 seconds while the pan-tilt action should take 10 seconds. Listing 5.2 shows the corresponding CRAM high-level code.

Listing 5.2: *Example code for projecting two parallel actions, one movement of the pan-tilt unit and one movement with the arm to grasp an object.*

```
1 (with-designators
2     ((grasp-action (action '((to grasp) (obj cup))))
3      (ptu-action (action '((to see) (location pose)))))
4   (par
5     (perform grasp-action)
6     (perform ptu-action)))
```

As can be seen, two process modules are involved, the pan-tilt process module and the manipulation process module. When executing, both process modules make assertions on the joint states of the robot. When the above plan is executed, the order in which the two parallel perform tasks are started is undefined and determined by various factors including the operating system scheduler. When started, each pro-

cess module will first infer the duration of the executed action. Then it will call the method clock–wait to sleep for the duration of the action and then perform the necessary assertions in the world database and generate the respective events. As shown in Figure 5.2, both process modules enter a sleeping state more or less at the same time. After 20ms, the PTU process module will exit the sleep state first since its action only takes 10 seconds and is shorter than the duration of the manipulation action. Then, the timeline is advanced and a new robot–state–changed event is asserted. In the meantime, the manipulation action's clock–wait method will terminate and re-enter its sleep state since its duration has not expired yet. After another 20ms, it will finally advance the timeline and assert a second robot–state–changed event at 30 seconds from start.

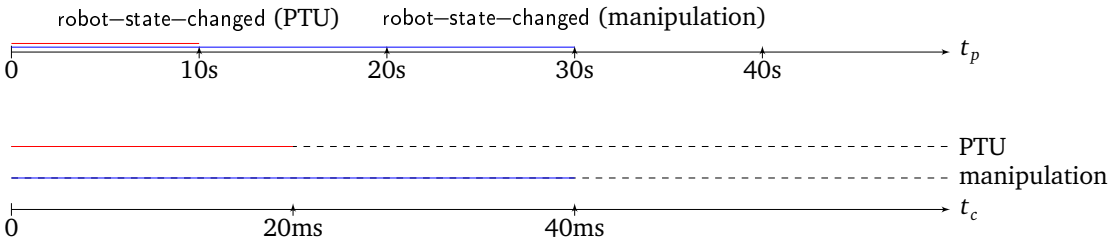


Figure 5.2: *Projection of two parallel actions in two different process modules and the generated projection timeline.*

The implementation of this projection clock assumes that most of the time required to execute a plan is spent waiting for process modules to finish an action and not in the high-level plan. In fact, it is important to assume that the execution of the high-level plan never blocks for longer than 50ms. This assumption is mostly true, however some reasoning tasks such as the inference of locations might require more time. Hence, the projection clock provides the functionality to enable and disable the clock completely which is used in designator resolution, for instance.

5.3 Definition and Matching of Behavior Flaws

Execution of plans in a projection environment generates the same data structures as executing the plan on the actual robot, in particular the time line with events and the corresponding (projected) world databases at the time the events occurred and the

recorded task tree. Based on these data structures, specific predefined flaws can be matched which allows to infer specific unwanted conditions in plan execution with different severities.

To define and match behavior flaws, CRAM's Prolog based reasoning engine is used. More specifically, a flaw is defined with the predicate behavior–flaw with the following signature:

```
(behavior-flaw ?trace ?severity ?name ?data)
```

The matching mechanism always binds the variable ?trace to the execution trace that contains the task tree and time line to be matched. For each flaw that can be found in the execution trace, the behavior–flaw predicate will hold. The following Prolog code shows how matching all behavior flaws on a given execution trace is performed:

```
1 (<- (flaws ?trace ?flaws)
2   (setof (?name ?severity ?data)
3          (behavior-flaw ?trace ?severity ?name ?data)
4          ?flaws))
```

As can be seen, the code just computes the (unsorted) set of all behavior flaws that match on an execution trace and binds them to a variable. The system then uses the severity value (a numeric value) for computing an overall score of the execution trace which is used for finding the best execution scenario. Four main flaws with different severity are currently defined, plan errors which are flaws that correspond to run-time errors, occlusions of objects (both, objects involved in a plan and other objects), blocking objects and finally the distance the robot navigated during the execution of a plan.

Plan errors. Flaws matching plan errors are most severe since they represent runtime errors which might cause the execution of the top-level plan to fail. Many plan failures

are handled locally in the plan library and cause the top-level plan to fail only if they occur repeatedly, which normally happens if they cannot be fixed. Although failures that are handled locally might not cause the overall plan to fail, they still influence the performance of the overall plan since local fixes and re-execution of a plan require additional actions. Therefore, we define two classes of plan error flaws:

1. Critical plan failures that caused the top-level plan to fail in projection. These failures have a very high and constant severity value since they prevent the top-level plan to succeed.

2. Plan failures that are handled. The severity value is computed based on the number of failures that occurred and their type.

Listing 5.3: *The definition of the behavior flaw for top-level plan failures.*

```
1 (<- (behavior-flaw ?trace 1000 top-level-failure ?error)
2   (top-level ?trace ?task)
3   (task-error ?task ?error))
```

Listing 5.3 shows the definition of the behavior flaw for top-level errors and Listing 5.4 shows the definition of the flaw for (handled) plan errors with the severity value computed from the number of failures.

Listing 5.4: *The definition of the flaw for computing the severity value from the number of (handled) plan errors.*

```
1 (<- (behavior-flaw ?trace ?severity handled-plan-failures ?
      errors)
2   (setof ?error (task-error ?trace ?_ ?error) ?errors)
3   (lisp-fun severity-from-errors ?errors ?severity))
```

The definition of this flaw uses a utility function implemented in Lisp for computing the actual severity value from the list of errors that occurred during plan execution.

Since the functor setof is used, each failure object is counted only once which is important since one failure object can cause several tasks to fail when it propagates up the call stack.

Occlusions. Occlusions might indirectly cause plan errors since visibility of objects is a precondition for grasping them. These cases are relatively rare though because the generation of poses for the robot's base to see objects as explained in Section 4.1.1.3 only fails if the environment is not completely known (this case cannot be projected though) or if the scene is extremely cluttered. Occlusions are still considered as flaws because if, for instance, the robot needs to navigate to the other end of the table in order to perceive an object because it is occluded otherwise, the performance of the plan is worse than without navigation. The occlusion flaw is mainly used for finding put-down locations which do not negatively influence subsequent pick-and-place actions.

Listing 5.5: *The definition of the behavior flaw for matching occlusions caused by put-down actions.*

```
1  (<- (behavior-flaw ?trace ?severity occlusion-flaw
2           (?object ?occluded-objects))
3    (task-goal ?trace ?put-down-task
4      (achieve (object-paced-at ?object ?_)))
5    (task-end ?put-down-task ?end-time)
6    (execution-trace-timeline ?trace ?timeline)
7    (robot ?robot)
8    (holds ?timeline
9      (occluding-objects ?_ ?robot ?object ?occluded-objects)
10     (at ?end-time))
11   (lisp-fun severity-from-occlusions ?occluded-objects ?severity
       ))
```

The definition of the "occlusion" flaw is based on the occluding-objects predicate which is evaluated in the recorded world instance after an object has been put down by a corresponding action. The definition of the respective flaw is shown in Listing 5.5.

The severity value is computed from the number of occluded objects before and after the put-down action.

Blocking objects. In particular for put-down locations which are highly under-specified, e.g. locations on a counter top for temporarily putting down an objects, flaws based on the *blocking* relation as defined in Section 3.1.3 can become important. This is in particular the case for plans involving many objects where some areas are highly cluttered. The severity value of the flaw related to blocking objects is computed from the number of objects that are blocked by a specific object after it has been put down. The corresponding flaw definition is shown in Listing 5.6.

Listing 5.6: *The definition of the behavior flaw for finding locations for objects which are causing other objects to be blocked.*

```
(<- (behavior-flaw ?trace ?severity blocked-objects-flaw
        (?object ?blocked-objects))
  (task-goal ?trace ?put-down-task
             (achieve (object-paced-at ?object ?_)))
  (task-end ?put-down-task ?end-time)
  (execution-trace-timeline ?trace ?timeline)
  (robot ?robot)
  (holds ?timeline
         (setof ?blocked-object
                (blocking ?_ ?robot ?blocked-object ?object)
                ?blocked-objects)
         (at ?end-time))
  (lisp-fun severity-from-blocked-objects ?blocked-objects ?
     severity))
```

Navigation distances. Although not critical, the distance the robot drives over the course of actions in a plan has a great impact on the overall plan performance. By defining a severity value computed from this distance, plan execution optimizes locations for least distance, i.e. for minimizing the time spent during navigation. Ad-

ditionally, this allows for executing pick-up and put-down actions for two objects in parallel given they are both reachable from the same location. The definition of the flaw is straight forward. The inference engine searches for all navigation sub-plans and queries the corresponding start and end time. Since projection does not simulate the complete trajectories the robot follows while executing actions but is based on discrete transitions in the state space, the navigated distance is approximated by using the euclidean distance. Although this approach is not accurate in all possible cases, it still provides a good approximation in cases where the robot would follow a linear trajectory anyway. Listing 5.7 shows the corresponding flaw definition.

Listing 5.7: *The definition of the navigation distance behavior flaw that is based on a linear approximation of the robot's navigation trajectory.*

```
(<- (behavior-flaw ?trace ?severity navigation-distance-flaw
        ?full-distance)
  (execution-trace-timeline ?trace ?timeline)
  (setof ?distance
         (and
           (task-goal ?trace ?location-task (at-location ?_))
           (task-start ?location-task ?start-time)
           (task-end ?location-task ?end-time)
           (holds ?timeline (loc Robot ?start-loc)
                  (at ?start-time))
           (holds ?timeline (loc Robot ?end-loc) (at ?end-time))
           (lisp-fun distance ?start-loc ?end-loc ?distance))
         ?distances)
  (lisp-fun severity-from-distances ?distances ?fill-distance))
```

The lisp function severity–from–distances just sums up the distances from a list and scales the resulting value to generate a severity value smaller than the severity values of all other flaws since the navigation distance flaw is not critical.

5.4 Location Designator Optimization Using Projection

The simplest while still one of the most useful applications of projection is its application in location designator resolution. Instead of just using a static scene (i.e. the currently known world state), integrating projection allows for generating locations by taking into account the future course of actions. The basic idea is to run projections in parallel to actual plan execution and record a set of execution traces. Since projection is relatively fast (in the order of a few seconds for a complete pick-and-place plan), most plan steps in actual plan execution have access to at least one projected timeline. When a location designator in a plan needs to be resolved, the system first generates a severity value for a projected episode by querying the set of all flaws and then summing up the severity values. The episode with the smallest severity value is then used by designator resolution.

Each designator is uniquely identified by its location in the plan since every with−designator form has a unique path and designator names at a specific path are unique as well. These paths are the same in all projections of the same plan so to resolve a specific location designator at a specific path, the system can just use the corresponding designator in the best projection.

Plans are written to be universal. One of the most important properties of these plans is that nothing is executed if a goal has been achieved already. This is in particular interesting for projection since this allows the projection mechanism to always start with the top-level plan and all actions that have been executed already will not be projected anymore. In other words, although the top-level plan is projected, only future actions actually cause updates on the projection time line. However, there are still cases where all projections have to be dropped completely. This is the case for all events that assert new, incompatible, states in the world database. For instance, if new, formerly unknown objects are detected by perception, all previously generated projections must be considered invalid.

5.5 Related Work

Related publications in the area of planning using geometry and physics simulation include [Zickler and Veloso, 2009] where the authors integrate a physics engine to

calculate state transitions in planning. In [Dornhege et al., 2009], the authors combine symbolic and geometric planning by calculating predicate values for a high level planner using a geometric planner. However, the authors in both publications do not integrate high-level concepts such as reachability or visibility. Planning under uncertainty, integrating a geometric representation of the world including free and occluded areas has been shown in [Kaelbling and Lozano-Perez, 2012]. In [Michel et al., 2007], visibility simulation is integrated into a motion planner and in [Marin et al., 2008] the authors show similar visibility calculation as presented in this paper in the context of human-robot interaction and perspective taking.

While all these publications show the implementation of reasoning in geometric domains, none of them provides means for *generating* parameters such as destination poses for objects or poses for the robot to stand based on static but also temporal constraints.

Temporal projection is a well studied field in formal logic. Given a sequence of actions, temporal projection tries to infer the state of the world after executing these actions. In [Hanks, 1990] and [McDermott, 1997], the authors deal with the problem of uncertain knowledge about the initial state of the world. McDermott introduces the generation of time lines, similar to the work presented in this article. However, purely symbolic approaches are often too abstract to represent geometric properties of the world, for instance occlusions. While simulation based projection as presented in [Kunze et al., 2011] and the authors' previous work in [Mösenlechner and Beetz, 2009] provide similar functionality for geometric reasoning and reasoning about actions as presented in this paper, they suffer from high computational complexity and extremely long run times.

5.6 Discussion

In this chapter, we presented a novel projection mechanism for high-level robot plans to execute pick-and-place actions in a human household. In contrast to previous approaches for deriving the outcome of a plan that were either using a simulation environment or that were implemented completely on a symbolic level, the work presented here combines both approaches. While actions are represented as discrete transitions in the world state, they are modeled to have a duration and concurrency is

modeled by using the CRAM Plan Language and a special delay mechanism. Physical effects, e.g. stability, reachability and visibility are integrated by using CRAM reasoning extensions that use a geometric representation of the robot and its environment. Projection as presented here is used to optimize the locations for the robot to stand while performing specific actions. However, projection could be used in a complete transformational planning system as presented, for instance, in [Müller, 2008] to replace the computationally very expensive simulation environment.

Chapter 6

Conclusion

As more and more research institutes own a robotic platform such as the PR2 or Baxter[1] that are capable of dexterous manipulation tasks, the integration of sophisticated reasoning components, task execution and lower level planning and control components for navigation and manipulation becomes more and more important. In this thesis, we presented the Cognitive Robot Abstract Machine, a toolbox for implementing cognitive behavior on mobile robots.

We presented the CRAM Plan Language, a domain specific programming language particularly designed to solve the problems roboticists are facing when solving complex tasks such as household chores. These include the highly concurrent nature of such control programs since different sensors have to be monitored in parallel, new tasks arrive and constraints change during program execution. Reasoning is an integral part of decision making and to achieve optimal results, not only symbolic knowledge but also the environment's geometry has to be taken into account.

In fact, the CRAM Plan Language provides first class language features for monitoring, parallel task execution, error handling and recovery. In addition, CRAM plans can be annotated with logical expressions describing the semantics of the annotated plan part. This is important to make inferences about the intention and the effects of a plan.

Besides the CRAM Plan Language, a second fundamental component of the CRAM framework is its Prolog interpreter that is designed for integrating it into programs. This is achieved by saving the complete computational context of a query for the later generation of further solutions. The Prolog interpreter integrates seamlessly with the

[1] www.rethingrobotics.com

CRAM Plan Language and has access to all its data structures which allows for the implementation of predicates that reason about the properties of programs and the robot's belief state.

In addition a clean and concise yet flexible interface to the actual robot's hardware is important. To maximize code reuse, components which are robot specific need to be separated from code that is independent of the robot's hardware. For instance, the basic actions that need to be performed to pick up an object are the same for a huge variety of different robot platforms. However, the way they interact with the hardware through middlewares differs among hardware platforms. Robot specific parameters such as the grasps to use in order to pick up an object or the location to stand while performing an action differs between robots as well. CRAM thus provides two further libraries, Designators and Process Modules, which complement the CRAM Plan Language with a layer of abstraction to separate robot specific code from generic code. Designators provide a way to describe constraints on parameters such as grasps, locations but also to describe objects on a symbolic level and define an interface to provide mechanisms to resolve these symbolic parameters and generate suitable input data for the robot's low level components. Process Modules on the other hand are hardware specific components responsible for performing the actions described by Designators on a robot. The Process Module library defines an interface for executing actions on the robot and enables the user to substitute one process module with another to run the same plan on different robots.

The CRAM framework has been used on two mobile platforms (TUM-Rosie and TUM-James) for executing various pick-and-place actions. This lead to the development a plan library with plans for navigation, perception and pick-and-place actions.

Most high-level planners only allow to plan on a symbolic level to keep planning problems feasible. However, most problems that occur when executing actions in complex environments are caused by geometric aspects, for instance occlusions, objects blocking grasp trajectories to reach for other objects or the robot being placed suboptimally for executing an action. Thus, geometry and reasoning about the geometric properties of the environment is a fundamental capability required to successfully perform actions for instance in a human household. We presented an extension to CRAM's Prolog interpreter that allows to reason about visibility, reachability and physical aspects such as stability. We presented a novel approach of integrating geometric reasoning

and reasoning about physics in a Prolog reasoning engine. To evaluate predicates such as *visible* or *stable*, the system uses a geometric database representing the robot's environment to avoid problems occurring with classic symbolic reasoning. For instance, it is hard to deal with occlusions in purely symbolic reasoning systems. In fact, our approach elegantly circumvents the frame problem and makes reasoning about object identities and the anchoring problem easier since object identities can be resolved on a geometric level.

Besides the integration of geometric aspects, we also presented CRAM's support for generating execution traces that allow to fully reconstruct the course of actions and the robot's belief state at any point in time during plan execution. The execution trace is accessed through Prolog predicates which allows for automated plan debugging by detecting flaws in the robot's behavior. A second application for the execution trace is to use it to generate input data for learning algorithms.

Choosing a good location for executing a specific task is a crucial but complex task and has a high impact on the overall performance of a plan. Many constraints including geometric constraints, symbolic constraints and temporal constraints have to be taken into account when generating locations for the robot to stand in order to perform a specific action or locations for placing objects. We presented different algorithms, including the computation of locations from which an object is visible and the use of inverse reachability maps which are integrated in the CRAM Designators library.

The locations chosen in earlier actions influence the set possible locations in later actions, i.e. there exist implicit temporal constraints between the parameters of subsequent actions. For instance, objects placed on the table might occlude other objects which are needed later in a plan. Our proposed solution to avoid problems caused by poorly chosen locations that hinder subsequent actions as shown in this thesis is to apply a lightweight temporal projection mechanism that combines execution traces, the geometric world database and reasoning about visibility, stability and reachability to make accurate predictions of the outcome of a plan and to infer flaws in the chosen locations used in that plan.

CRAM is designed with extensibility and rapid prototyping in mind. It provides a rich toolbox of libraries and already implemented algorithms while providing interfaces to extend the existing functionality. It consists of many different small libraries and the

user can chose which functionality to use by just loading these specific libraries. For instance, if the user wants to just use geometric reasoning, there is no requirement for using the CRAM Plan Language. If the user needs a mechanism for finding locations from which objects can be detected, it is enough to load the corresponding library.

The lack of a full featured high level action planner is one limitation of the current CRAM system. We believe however that this is not a major limitation because high level plans for household chores tend to break down to the sequencing of a low number of actions such as *pick up*, *put down*, *navigation* and the interaction with articulated objects such as drawers and doors. KNOWROB already contains the functionality to generate CRAM plans that sequence these basic actions according to action recipes, for instance imported from the World Wide Web.

Although CRAM already provides a rich set of features, some limitations have to be tackled in the future:

Asynchronous Process Modules. Although plans in the CRAM Plan Library have been rewritten to support asynchronous process modules and a prove of concept implementation of such process modules has been created, there is still no implementation of asynchronous actions on an actual robot. Evidently, the advantages of asynchronous process modules are that subsequent actions, for instance reaching for an object, grasping it, lifting it and moving it to a carry position, can be combined to one smooth action. Unfortunately, it is hard to implement such smooth behavior with existing motion planning frameworks. One notable exception is the iTaSC framework [Smits et al., 2009].

Interaction with Knowrob. Some CRAM components already integrate specific components of KNOWROB, for instance its semantic maps. CRAM would benefit from an even closer integration, for instance by making use of a semantic robot description in the geometric reasoning framework to infer robot specific parameters such as the names of the camera links, grippers and kinematic chains. Feedback about currently executed actions has to be provided to KNOWROB in the future as well. For instance, when opening a drawer, KNOWROB's semantic map has to be updated to ensure consistency of KNOWROB's knowledge base with CRAM's belief state.

Plan transformations. A full featured transformational planning system that integrates temporal projection, execution traces and the application of transformation rules would allow for optimizing plans at run time or while the robot is idle. In [Müller, 2008], it has been shown that the performance of commonly executed plans can be improved significantly by applying transformational planning. In fact, the CRAM Plan Language has been designed to support plan transformations even while a plan is executed.

List of Prior Publications

The work presented in this document is partly based on prior publications. Sections of this work that drew upon content from prior publications cited the respective publications where appropriate. A complete list of publications that were (co-)authored during my research as a doctoral candidate is provided below.

Journal Articles

Freek Stulp, Andreas Fedrizzi, Lorenz Mösenlechner, and Michael Beetz. Learning and Reasoning with Action-Related Places for Robust Mobile Manipulation. *Journal of Artificial Intelligence Research (JAIR)*, 43:1–42, 2012.

Michael Beetz, Dominik Jain, Lorenz Mösenlechner, Moritz Tenorth, Lars Kunze, Nico Blodow, and Dejan Pangercic. Cognition-Enabled Autonomous Robot Control for the Realization of Home Chore Task Intelligence. *Proceedings of the IEEE, Special Issue on Quality of Life Technology*, 100(8):2454–2471, 2012.

Michael Beetz, Dominik Jain, Lorenz Mösenlechner, and Moritz Tenorth. Towards Performing Everyday Manipulation Activities. *Robotics and Autonomous Systems*, 58(9):1085–1095, 2010.

Conference and Workshop Papers

Lorenz Mösenlechner and Michael Beetz. Fast Temporal Projection Using Accurate Physics-Based Geometric Reasoning. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2013. Accepted for publication.

Michael Beetz, Ulrich Klank, Ingo Kresse, Alexis Maldonado, Lorenz Mösenlechner, Dejan Pangercic, Thomas Rühr, and Moritz Tenorth. Robotic Roommates Making Pancakes. In *11th IEEE-RAS International Conference on Humanoid Robots*, 2011.

Lorenz Mösenlechner and Michael Beetz. Parameterizing Actions to have the Appropriate Effects. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2011.

Michael Beetz, Lorenz Mösenlechner, Moritz Tenorth, and Thomas Rühr. CRAM – a Cognitive Robot Abstract Machine. In *5th International Conference on Cognitive Systems (CogSys 2012)*, 2012.

Ulrich Klank, Lorenz Mösenlechner, Alexis Maldonado, and Michael Beetz. Robots that Validate Learned Perceptual Models. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2012.

Séverin Lemaignan, Raquel Ros, Lorenz Mösenlechner, Rachid Alami, and Michael Beetz. ORO, a knowledge management module for cognitive architectures in robotics. In *Proceedings of the 2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3548–3553, 2010.

Lorenz Mösenlechner, Nikolaus Demmel, and Michael Beetz. Becoming Action-aware through Reasoning about Logged Plan Execution Traces. In *IEEE/RSJ International Conference on Intelligent RObots and Systems.*, pages 2231–2236, 2010.

Michael Beetz, Lorenz Mösenlechner, and Moritz Tenorth. CRAM – A Cognitive Robot Abstract Machine for Everyday Manipulation in Human Environments. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1012–1017, 2010.

Alexandra Kirsch, Thibault Kruse, and Lorenz Mösenlechner. An Integrated Planning and Learning Framework for Human-Robot Interaction. In *4th Workshop on Planning and Plan Execution for Real-World Systems (held in conjuction with ICAPS 09)*, 2009.

Lorenz Mösenlechner and Michael Beetz. Using Physics- and Sensor-based Simulation for High-fidelity Temporal Projection of Realistic Robot Behavior. In *19th International Conference on Automated Planning and Scheduling (ICAPS'09).*, 2009.

Dominik Jain, Lorenz Mösenlechner, and Michael Beetz. Equipping Robot Control Programs with First-Order Probabilistic Reasoning Capabilities. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 3626–3631, 2009.

Andreas Fedrizzi, Lorenz Moesenlechner, Freek Stulp, and Michael Beetz. Transformational Planning for Mobile Manipulation based on Action-related Places. In *Proceedings of the International Conference on Advanced Robotics (ICAR).*, pages 1–8, 2009.

Michael Beetz, Freek Stulp, Bernd Radig, Jan Bandouch, Nico Blodow, Mihai Dolha, Andreas Fedrizzi, Dominik Jain, Uli Klank, Ingo Kresse, Alexis Maldonado, Zoltan Marton, Lorenz Mösenlechner, Federico Ruiz, Radu Bogdan Rusu, and Moritz Tenorth. The Assistive Kitchen – A Demonstration Scenario for Cognitive Technical Systems. In *IEEE 17th International Symposium on Robot and Human Interactive Communication (RO-MAN), Muenchen, Germany*, pages 1–8, 2008. Invited paper.

Bibliography

[Aiello et al., 2007] Aiello, M., Pratt-Hartmann, I., and van Benthem, J., editors (2007). *Handbook of Spatial Logics*. Springer.

[Anderson, 1993] Anderson, J. R. (1993). *Rules of the Mind*. Lawrence Erlbaum Associates Inc, New Jersey. Gibts in Golm unter 'CP 4000 AND'.

[Arnold et al., 2004] Arnold, M., Fink, S., Grove, D., Hind, M., and Sweeney, P. F. (2004). Architecture and Policy for Adaptive Optimization in Virtual Machines. Technical Report 23429, IBM Research.

[Baillie, 2005] Baillie, J.-C. (2005). Urbi: towards a universal robotic low-level programming language. In *IROS*, pages 820–825.

[Bechhofer et al., 2004] Bechhofer, S., van Harmelen, F., Hendler, J., Horrocks, I., McGuinness, D., Patel-Schneider, P., and Stein, L. (2004). OWL Web Ontology Language Reference. W3C Recommendation.

[Beetz, 2000] Beetz, M. (2000). *Concurrent Reactive Plans: Anticipating and Forestalling Execution Failures*, volume LNAI 1772 of *Lecture Notes in Artificial Intelligence*. Springer Publishers.

[Beetz et al., 2011] Beetz, M., Klank, U., Kresse, I., Maldonado, A., Mösenlechner, L., Pangercic, D., Rühr, T., and Tenorth, M. (2011). Robotic Roommates Making Pancakes. In *11th IEEE-RAS International Conference on Humanoid Robots*, Bled, Slovenia.

[Beetz et al., 2010] Beetz, M., Stulp, F., Esden-Tempski, P., Fedrizzi, A., Klank, U., Kresse, I., Maldonado, A., and Ruiz, F. (2010). Generality and legibility in mobile manipulation. *Autonomous Robots Journal (Special Issue on Mobile Manipulation)*,

28(1):21–44.

[Berenson et al., 2007] Berenson, D., Diankov, R., Nishiwaki, K., Kagami, S., and Kuffner, J. (2007). Grasp planning in complex scenes. In *IEEE-RAS International Conference on Humanoid Robots*.

[Bloch, 2006] Bloch, I. (2006). Spatial reasoning under imprecision using fuzzy set theory, formal logics and mathematical morphology. *International Journal of Approximate Reasoning*, 41(2):77 – 95. <ce:title>Advances in Fuzzy Sets and Rough Sets</ce:title>.

[Blodow et al., 2010] Blodow, N., Jain, D., Marton, Z.-C., and Beetz, M. (2010). Perception and Probabilistic Anchoring for Dynamic World State Logging. In *10th IEEE-RAS International Conference on Humanoid Robots*, pages 160–166, Nashville, TN, USA.

[Bohren and Cousins, 2010] Bohren, J. and Cousins, S. (2010). The SMACH High-Level Executive. *IEEE Robotics and Automation Magazine*, 17:18–20.

[Bohren et al., 2011] Bohren, J., Rusu, R. B., Jones, E. G., Marder-Eppstein, E., Pantofaru, C., Wise, M., Mosenlechner, L., Meeussen, W., and Holzer, S. (2011). Towards autonomous robotic butlers: Lessons learned with the pr2. In *ICRA*, Shanghai, China.

[Bonasso et al., 1997] Bonasso, P., Firby, J., Gat, E., Kortenkamp, D., Miller, D., and Slack, M. (1997). Experiences with an Architecture for Intelligent, Reactive Agents. *Journal of Experimental and Theoretical Artificial Intelligence*, 9(1).

[Clementini et al., 1997] Clementini, E., Felice, P. D., and Hernández, D. (1997). Qualitative representation of positional information. *Artificial Intelligence*, 95(2):317 – 356.

[Coyne and Sproat, 2001] Coyne, B. and Sproat, R. (2001). Wordseye: an automatic text-to-scene conversion system. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '01, pages 487–496, New York, NY, USA. ACM.

[Diankov, 2010] Diankov, R. (2010). *Automated Construction of Robotic Manipulation Programs*. PhD thesis, Carnegie Mellon University, Robotics Institute.

[Diankov and Kuffner, 2008] Diankov, R. and Kuffner, J. (2008). Openrave: A planning architecture for autonomous robotics. Technical Report CMU-RI-TR-08-34, Robotics Institute, Pittsburgh, PA.

[Dijkstra, 1965] Dijkstra, E. W. (1965). Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569–.

[Doherty et al., 1998] Doherty, P., Gustafsson, J., Karlsson, L., and Kvarnstrom, J. (1998). Temporal action logics (tal): Language specification and tutorial.

[Dornhege et al., 2009] Dornhege, C., Gissler, M., Teschner, M., and Nebel, B. (2009). Integrating symbolic and geometric planning for mobile manipulation. In *IEEE International Workshop on Safety, Security and Rescue Robotics (SSRR)*.

[Dutta, 1989] Dutta, S. (1989). Qualitative spatial reasoning: A semi-quantitative approach using fuzzy logic. In *SSD*, pages 345–364.

[Firby, 1987] Firby, J. (1987). An Investigation into Reactive Planning in Complex Domains. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 202–206, Seattle, WA.

[Giacomo et al., 2000] Giacomo, G. D., Lespérance, Y., and Levesque, H. J. (2000). Congolog, a concurrent programming language based on the situation calculus.

[Grisetti et al., 2007] Grisetti, G., Stachniss, C., and Burgard, W. (2007). Improved techniques for grid mapping with rao-blackwellized particle filters. *IEEE Transactions on Robotics*, 23:2007.

[Hammond et al., 1995] Hammond, K. J., Converse, T. M., and Grass, J. W. (1995). The stabilization of environments. *Artificial Intelligence*, 72(1-2):305–327.

[Hanks, 1990] Hanks, S. (1990). Practical temporal projection. In *Proc. of AAAI-90*, pages 158–163.

[Hanks and McDermott, 1987] Hanks, S. and McDermott, D. (1987). Nonmonotonic logic and temporal projection. *Artificial intelligence*, 33(3):379–412.

[Hoare, 1974] Hoare, C. A. R. (1974). Monitors: an operating system structuring concept. *Commun. ACM*, 17(10):549–557.

[Hsiao and Lozano-Pérez, 2006] Hsiao, K. and Lozano-Pérez, T. (2006). Imitation learning of whole-body grasps. In *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*.

[Johnston and Williams, 2008] Johnston, B. and Williams, M. (2008). Comirit: Commonsense Reasoning by Integrating Simulation and Logic. In *Artificial General Intelligence 2008: Proceedings of the First AGI Conference*, page 200. IOS Press.

[Jungert, 1993] Jungert, E. (1993). Symbolic spatial reasoning on object shapes for qualitative matching. In *Spatial Information Theory A Theoretical Basis for GIS*, volume 716 of *Lecture Notes in Computer Science*, pages 444–462. Springer Berlin Heidelberg.

[Kaelbling and Lozano-Perez, 2012] Kaelbling, L. P. and Lozano-Perez, T. (2012). Unifying perception, estimation and action for mobile manipulation via belief space planning. In *IEEE Conference on Robotics and Automation (ICRA)*.

[Kiczales and Rivieres, 1991] Kiczales, G. and Rivieres, J. D. (1991). *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA.

[Klank, 2012] Klank, U. (2012). *Everyday Perception for Mobile Manipulation in Human Environments*. PhD thesis, Technische Universität München.

[Kunze et al., 2011] Kunze, L., Dolha, M. E., Guzman, E., and Beetz, M. (2011). Simulation-based temporal projection of everyday robot object manipulation. In Yolum, Tumer, Stone, and Sonenberg, editors, *Proc. of the 10th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2011)*, Taipei, Taiwan. IFAAMAS.

[Langley and Choi, 2006] Langley, P. and Choi, D. (2006). A unified cognitive architecture for physical agents. In *PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE*, volume 21, page 1469. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999.

[Latombe, 1991] Latombe, J.-C. (1991). *Robot Motion Planning*. Kluwer Academic Publishers, Boston, MA.

[Levesque et al., 1997] Levesque, H., Reiter, R., Lesp?rance, Y., Lin, F., and Scherl, R. (1997). Golog: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31:59–84.

[Maitin-Shepard et al., 2010] Maitin-Shepard, J., Cusumano-Towner, M., Lei, J., and Abbeel, P. (2010). Cloth Grasp Point Detection based on Multiple-View Geometric Cues with Application to Robotic Towel Folding. In *International Conference on Robotics and Automation (ICRA)*.

[Marin et al., 2008] Marin, L., Sisbot, E. A., and Alami, R. (2008). Geometric tools for perspective taking for human-robot interaction. In *Mexican International Conference on Artificial Intelligence (MICAI 2008)*.

[Matsui and Inaba, 1991] Matsui, T. and Inaba, M. (1991). Euslisp: an object-based implementation of lisp. *J. Inf. Process.*, 13(3):327–338.

[McCarthy, 1960] McCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195.

[McCarthy and Hayes, 1969] McCarthy, J. and Hayes, P. J. (1969). Some philosophical problems from the standpoint of artificial intelligence. In Meltzer, B. and Michie, D., editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press.

[McDermott, 1993] McDermott, D. (1993). A reactive plan language. Technical report, Yale University, Computer Science Dept.

[McDermott, 1997] McDermott, D. (1997). An algorithm for probabilistic, totally-ordered temporal projection. In Stock, O., editor, *Spatial and Temporal Reasoning*. Kluwer Academic Publishers, Dordrecht.

[McGann et al., 2008] McGann, C., Py, F., Rajan, K., Thomas, H., Henthorn, R., and McEwen, R. (2008). A deliberative architecture for auv control. In *International Conference on Robotics and Automation (ICRA)*, pages 1049–1054.

[Meeussen et al., 2010] Meeussen, W., Wise, M., Glaser, S., Chitta, S., McGann, C., Mihelich, P., Marder-Eppstein, E., Muja, M., Eruhimov, V., Foote, T., Hsu, J., Rusu, R. B., Marthi, B., Bradski, G., Konolige, K., Gerkey, B. P., and Berger, E. (2010).

Autonomous door opening and plugging in with a personal robot. In *International Conference on Robotics and Automation (ICRA)*.

[Michel et al., 2007] Michel, P., Scheurer, C., Kuffner, J., Vahrenkamp, N., and Dillmann, R. (2007). Planning for robust execution of humanoid motions using future perceptive capability. In *Proceedings of the IEEE/RSJ IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'07)*, pages 3223–3228.

[Milnes et al., 1992] Milnes, B. G., Pelton, G., Doorenbos, R., Laird, M. H., Rosenbloom, P., and Newell, A. (1992). A specification of the soar cognitive architecture in z. Technical report, Pittsburgh, PA, USA.

[Mösenlechner and Beetz, 2009] Mösenlechner, L. and Beetz, M. (2009). Using physics- and sensor-based simulation for high-fidelity temporal projection of realistic robot behavior. In *19th International Conference on Automated Planning and Scheduling (ICAPS'09)*.

[Mösenlechner and Beetz, 2011] Mösenlechner, L. and Beetz, M. (2011). Parameterizing Actions to have the Appropriate Effects. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, San Francisco, CA, USA.

[Mösenlechner and Beetz, 2013] Mösenlechner, L. and Beetz, M. (2013). Fast temporal projection using accurate physics-based geometric reasoning. In *IEEE International Conference on Robotics and Automation (ICRA)*, Karlsruhe, Germany. Accepted for publication.

[Mösenlechner et al., 2010] Mösenlechner, L., Demmel, N., and Beetz, M. (2010). Becoming Action-aware through Reasoning about Logged Plan Execution Traces. In *IEEE/RSJ International Conference on Intelligent RObots and Systems.*, pages 2231–2236, Taipei, Taiwan.

[Müller, 2008] Müller, A. (2008). *Transformational Planning for Autonomous Household Robots using Libraries of Robust and Flexible Plans*. PhD thesis, Technische Universität München.

[Norvig, 1992] Norvig, P. (1992). *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. Morgan Kaufmann, San Mateo, CA.

[Pangercic et al., 2012] Pangercic, D., Tenorth, M., Pitzer, B., and Beetz, M. (2012). Semantic object maps for robotic housework - representation, acquisition and use. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Vilamoura, Portugal.

[Quigley et al., 2009] Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Berger, E., Wheeler, R., and Ng, A. (2009). ROS: an open-source Robot Operating System. In *In IEEE International Conference on Robotics and Automation (ICRA)*, Kobe, Japan.

[Reiser et al., 2009] Reiser, U., Connette, C., Fischer, J., Kubacki, J., Bubeck, A., Weisshardt, F., Jacobs, T., Parlitz, C., Hagele, M., and Verl, A. (2009). Care-o-bot 3 - creating a product vision for service robot applications by integrating design and technology. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1992 –1998.

[Ruehl et al., 2010] Ruehl, S., Xue, Z., Kerscher, T., and Dillmann, R. (2010). Towards automatic manipulation action planning for service robots. In Dillmann, R., Beyerer, J., Hanebeck, U., and Schultz, T., editors, *KI 2010: Advances in Artificial Intelligence*, volume 6359 of *Lecture Notes in Computer Science*, pages 366–373. Springer Berlin / Heidelberg.

[Saxena et al., 2008] Saxena, A., Driemeyer, J., and Ng, A. (2008). Robotic grasping of novel objects using vision. *The International Journal of Robotics Research*, 27(2):157.

[Schoppers, 1987] Schoppers, M. J. (1987). Universal plans for reactive robots in unpredictable environments. In McDermott, J., editor, *Proceedings of the Tenth International Joint Conference on Artificial Intelligence (IJCAI-87)*, pages 1039–1046, Milan, Italy. Morgan Kaufmann publishers Inc.: San Mateo, CA, USA.

[Schuster et al., 2012] Schuster, M., Jain, D., Tenorth, M., and Beetz, M. (2012). Learning Organizational Principles in Human Environments. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 3867–3874, St. Paul, MN, USA.

[Siméon et al., 2001] Siméon, T., Laumond, J.-P., and Lamiraux, F. (2001). Move3D: a generic platform for path planning. In *4th International Symposium on Assembly and Task Planning (ISATP 01)*, Fukuoka, Japan.

[Simmons et al., 2002] Simmons, R., Goldberg, D., Goode, A., Montemerlo, M., Roy, N., Sellner, B., Urmson, C., Bugajska, M., Coblenz, M., Macmahon, M., Perzanowski, D., Horswill, I., Zubek, R., Kortenkamp, D., Wolfe, B., Milam, T., and Maxwell, B. (2002). Grace: An autonomous robot for the aaai robot challenge.

[Smits et al., 2009] Smits, R., Laet, T. D., Claes, K., Bruyninckx, H., and Schutter, J. D. (2009). iTASC: A Tool for Multi-Sensor Integration in Robot Manipulation. In Hahn, H. K. H. and Lee, S., editors, *Multisensor Fusion and Integration for Intelligent Systems*, volume 35 of *Lecture Notes in Electrical Engineering*, pages 235–254. Springer.

[Sterling and Shapiro, 1994] Sterling, L. and Shapiro, E. (1994). *The Art of Prolog: Advanced Programming Techniques*. MIT Press.

[Sussman and Jr., 1975] Sussman, G. J. and Jr., G. L. S. (1975). Scheme: An interpreter for extended lambda calculus. In *MEMO 349, MIT AI LAB*.

[Tenorth and Beetz, 2012] Tenorth, M. and Beetz, M. (2012). Exchange of action-related information among autonomous robots. In *12th International Conference on Intelligent Autonomous Systems*.

[Tenorth and Beetz, 2013] Tenorth, M. and Beetz, M. (2013). KnowRob – A Knowledge Processing Infrastructure for Cognition-enabled Robots. Part 1: The KnowRob System. *International Journal of Robotics Research (IJRR)*. Accepted for publication.

[Tenorth et al., 2010a] Tenorth, M., Kunze, L., Jain, D., and Beetz, M. (2010a). KNOWROB-MAP – Knowledge-Linked Semantic Object Maps. In *10th IEEE-RAS International Conference on Humanoid Robots*, pages 430–435, Nashville, TN, USA.

[Tenorth et al., 2010b] Tenorth, M., Nyga, D., and Beetz, M. (2010b). Understanding and Executing Instructions for Everyday Manipulation Tasks from the World Wide Web. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 1486–1491, Anchorage, AK, USA.

[Thielscher, 1998] Thielscher, M. (1998). Introduction to the fluent calculus. *Electron. Trans. Artif. Intell.*, 2:179–192.

[Vernon et al., 2007] Vernon, D., Metta, G., and Sandini, G. (2007). A survey of artificial cognitive systems: Implications for the autonomous development of mental capabilities in computational agents. *IEEE Transactions on Evolutionary Computation*, 11(2):151–180.

[Waibel et al., 2011] Waibel, M., Beetz, M., D'Andrea, R., Janssen, R., Tenorth, M., Civera, J., Elfring, J., Gálvez-López, D., Häussermann, K., Montiel, J., Perzylo, A., Schießle, B., Zweigle, O., and van de Molengraft, R. (2011). RoboEarth - A World Wide Web for Robots. *Robotics & Automation Magazine*, 18(2):69–82.

[Weitnauer et al., 2010] Weitnauer, E., Haschke, R., and Ritter, H. (2010). Evaluating a physics engine as an ingredient for physical reasoning. In *Proceedings of the Second international conference on Simulation, modeling, and programming for autonomous robots*, SIMPAR'10, pages 144–155, Berlin, Heidelberg. Springer-Verlag.

[Weld, 1994] Weld, D. (1994). An introduction to least commitment planning. *AI Magazine*, 15(4):27–61.

[Wielemaker et al., 2012] Wielemaker, J., Schrijvers, T., Triska, M., and Lager, T. (2012). SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96.

[Wyrobek et al., 2008] Wyrobek, K. A., Berger, E. H., der Loos, H. F. M. V., and Salisbury, J. K. (2008). Towards a personal robotics development platform: Rationale and design of an intrinsically safe personal robot. In *International Conference on Robotics and Automation (ICRA)*, pages 2165–2170. IEEE.

[Zacharias et al., 2007] Zacharias, F., Borst, C., and Hirzinger, G. (2007). Capturing robot workspace structure: representing robot capabilities. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3229–3236.

[Zickler and Veloso, 2009] Zickler, S. and Veloso, M. (2009). Efficient physics-based planning: sampling search via non-deterministic tactics and skills. In *AAMAS '09: Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems*, pages 27–33, Richland, SC. IFAAMAS.