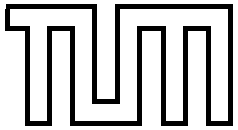


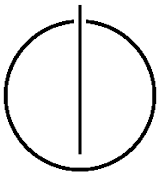
FAKULTÄT FÜR INFORMATIK

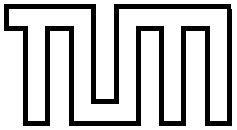
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Translating Qualitative Spatial Scene Descriptions into Their Intended Geometric Representation for Executing Household Tasks on Robots

Gayane Kazhoyan





FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Translating Qualitative Spatial Scene Descriptions into Their Intended Geometric Representation for Executing Household Tasks on Robots

Übersetzung qualitativer räumlicher Szenenbeschreibungen in ihre zugedachte Geometrische Repräsentation um Haushaltsaufgaben auf Robotern auszuführen

Author:	Gayane Kazhoyan
Supervisor:	Prof. Michael Beetz
Advisor:	Dipl. Inf. Lorenz Mösenlechner
Date:	October 15, 2012

Ich versichere, dass ich diese Masterarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

I assure the single handed composition of this master's thesis only supported by declared resources.

München, den 15. Oktober 2012 Gayane Kazhoyan

Acknowledgments

It is a great pleasure to thank everyone who helped me during my studies at TUM and throughout my work on this thesis.

I am truly grateful to my advisor Lorenz Mösenlechner for introducing me to the exciting world of functional programming, for my first practical experience on a real robot, and for proofreading the manuscript. His professionalism as a computer scientist has empowered me to work hard on this project.

This work has greatly benefited from invaluable advices of my supervisor Prof. Michael Beetz who ignited my interest towards AI.

I would like to thank the whole IAS group for a friendly working atmosphere and inspiring conversations. Thanks to all contributors of CRAM for such an impressive system that I had a chance to work on.

Warmest thanks to Ross for being by my side, keeping me happy even in hard days and inspiring my diligence.

My deep gratitude goes to my parents and brothers for supporting and encouraging me all these years.

I am obliged to my friends for their support throughout my Master's studies.

This work would have not been possible without the support of German national agency for academic exchange, and I owe sincere and earnest thankfulness to DAAD for sponsoring my Master's degree in Germany that has opened so many doors and possibilities for me.

Gayane Kazhoyan

Abstract

Action planning for autonomous robots that are supposed to perform various complex actions in real environments, such as human households, is a difficult task. Traditional symbolic task planners can be successfully used to find a course of actions to achieve the desired goal. However, it is hard for them to infer the quantitative parameters of actions, such as put down locations for manipulated objects. Therefore, it is sensible to combine qualitative and quantitative reasoning approaches. One of the systems that does this is Cognitive Robot Abstract Machine (CRAM) and it was used in this work. This thesis describes an approach to translate symbolic spatial descriptions into geometric parameters, which should then be used in low-level robot actions. The novelty of the method is that it utilizes the generative model of CRAM for resolving qualitative spatial descriptions, that is a number of solution candidates are generated based on a certain sampling mechanism, after which they are validated to conform with the execution context. This model is very general and enables working with different types of robot actions described with various symbolically represented geometric parameters. The approach was successfully evaluated in a scenario of autonomous table setting.

Zusammenfassung

Aktionsplanung für autonome Roboter, die verschiedene komplexe Handlungen in einer realen Umgebung, wie einem menschlichen Haushalt, ausführen sollen, ist eine schwierige Aufgabe. Die traditionellen symbolischen Aktionsplaner können erfolgreich eingesetzt werden, um den passenden Plan, der zum gewünschten Ziel führt, zu finden. Innerhalb dieser Aktionsplaner ist es aber schwer, die quantitativen Parameter der Aktionen, wie z.B. die "put down locations" für manipulierende Objekte, auszurechnen. Daher ist es sinnvoll die Ansätze für qualitative und quantitative Argumentation zu kombinieren. Eines der Systeme, das dies leistet, ist Cognitive Robot Abstract Machine (CRAM), das für die Zwecke dieser Masterarbeit verwendet wurde. Die Arbeit beschreibt einen Ansatz zum Übersetzen der symbolischen räumlichen Beschreibungen in die geometrische Parameter, welche innerhalb der Low-Level-Aktionen der Roboter verwendet werden sollen. Die Neuerung des hier vorgestellten Ansatzes ist, dass es die generativen Modelle von CRAM zum Auflösen der qualitativen räumlichen Beschreibungen verwendet werden. Das heißt dass eine Anzahl von Lösungen wird auf der Grundlage eines bestimmten Probenahmemechanismus vorgeschlagen, wonach sie entsprechend dem Ausführungskontext validiert werden. Dieses Modell ist sehr allgemein und ermöglicht das Beschreiben der verschiedenen Arten von Roboteraktionen mit verschiedenen symbolisch dargestellten geometrischen Parametern. Der Ansatz wurde erfolgreich in einem Szenario für das autonome Decken eines Tisches ausgewertet.

Contents

Part I.

Introduction

1. Objective

Robotics research nowadays has come close to creating autonomous robots that could perform manipulation activities to assist humans in household environments, such as pick and place objects, set a table, clean up, perform cooking tasks etc. There is a number of robots that can already perform such tasks, for example, the robotic vacuum cleaners or, as have been shown in a number of demos, robots pairing socks [20] or making pancakes [3]. However, these tasks are either very simple or well specified, including well known working environments. Whereas, being able to perform wider scope of activities in real human households involves carrying out tasks of higher complexity in the unpredictably changing environment. Moreover, the specifications of tasks are usually incomplete and vague. For example, if the robot gets a task to set a table, it needs to have many different types of knowledge to be able to perform it: which low-level actions should be performed to achieve the goal, in which order, which types of objects need to be used and where they can be found, what the desired positions are, what the trajectories for the arms and the robot base are, which grasps can be applied to pick up objects etc. In addition, the robot should also have information about its environment, such as where the table and the other furniture is. It should also have common sense knowledge about its environment and applicable physics laws in it in order to manipulate objects the right way.

There have been many solutions proposed to some of the subproblems listed. This work concentrates on spatial requirements and limitations of task objects and the robot or simply, it tackles the question, where exactly to put objects and po-

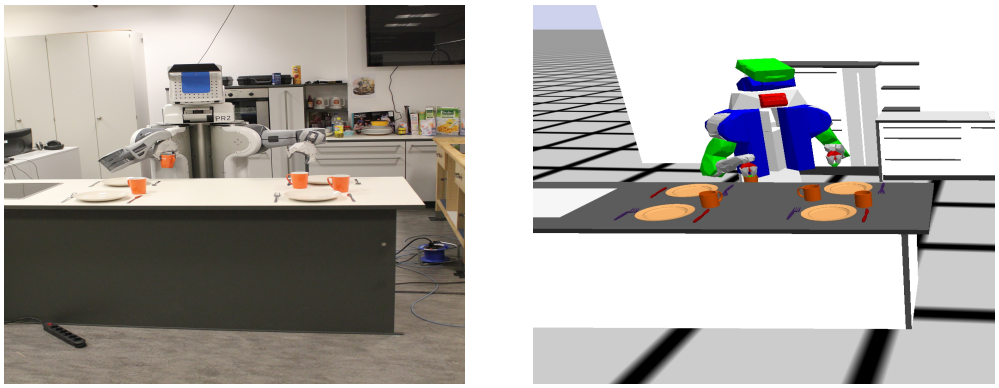


Figure 1.1.: PR2 in real human environment and its geometric representation.

sition the robot in order to achieve desired goals, thereby showing good performance. Knowing *how* to act the right way is as important as knowing exactly what actions to perform and in which order. The traditional approach to planning (i.e. finding the course of action that would lead to the desired goals) is to use symbolic reasoning [17]. Thus, based on the goal state and using a library of low-level plans, the symbolic planner performs logical inference to find the sequence of actions. However, symbolic planning cannot infer how to perform an action (e.g. where exactly to put an object during a put-down task execution), it only says which actions to perform. Therefore, it is also necessary to search for good parameters of robot's actions considering the given environment. They can, of course, be chosen randomly, however, bad parameters can lead to a failure in execution, while a good set of parameters can highly optimize the performance of plan execution. As purely symbolic reasoning systems have limited possibilities for computing quantitative parameters, it is necessary to combine qualitative reasoning with the quantitative one.

It is not that obvious how much influence correct positioning of objects has on performance and quality of execution of manipulation tasks. Humans performing everyday manipulation tasks find the correct parameters of their actions instinc-

tively. For example, they tend not to put easily breakable objects at the edge of a table. Or they try to place certain objects in a way that they do not hinder manipulation of other objects, for example, it is not efficient to put a coffee mug right next to a computer mouse if the mouse is supposed to be used shortly afterwards, or to put a big package in front of the screen that the person is going to look at. Robots, however, have no knowledge about this and for them finding the correct positions involves intense computational efforts.

The reasoning system described in this thesis generates those poses. It takes symbolic abstract specifications of locations and finds solutions that suit the specifications the best under a certain set of constraints. The location specifications are represented as lists of key-value pairs, e.g. `((on table) (for bowl))`. An example set of constraints is `reachable(bowl, robot), visible(bowl, robot)`. The solutions are found based on an accurate geometric representation of the world (as shown on Figure 1.1) and specialized inference methods, such as robot inverse kinematics solver based methods for inferring reachability. The system differentiates between better and worse solutions with respect to the expected performance of the execution of the plan where these solutions will be used. It has a generative model of finding solutions: first a number of solutions is generated, then they are checked against the constraints – the explicit ones and the intrinsic constraints from common sense knowledge and physics laws[1]. If the constraints are satisfied, the generated pose is proposed as a solution. For the previous example with the bowl the solution candidates will be generated based on the relations given in the symbolic description, and then it will be checked if the bowl positioned at the proposed pose is reachable and visible for the robot or not. It will also be checked if the position is stable with respect to physics laws. Using a lightweight simulation mechanism it can also be checked if the solution is good with respect to future actions. For example, it can be checked if the bowl at the chosen put down location

[1]The knowledge about physics laws comes from the Bullet physics engine: `http://bulletphysics.org`

will hinder the robot substantially during the subsequent manipulation tasks.

The main work performed in scope of this thesis was to extend the system described above to enable inferring locations based on spatial relation descriptions, that is it teaches the robot what means "left-of", "behind", "far" etc. It introduces a novel approach to accurately representing geometric scenes by means of qualitative spatial relations that utilizes the advantages of the generative model described above. The system works as following: for each location description a 2D costmap is generated over the whole domain of possible positions, that is if the robot is working in the kitchen, the created costmap associates, based on an objective function, with each coordinate of the kitchen (only in 2D and up to some resolution) a value. The objective function represents, to which extent a certain position is suitable for the specific location description. If the description is `(on table)`, the costmap will assign the maximum values to positions on the table, and minimum values elsewhere. The costmap is then used to sample solutions: the higher the value the coordinate has in the costmap, the more likely it is to be picked as a solution. The system does not rely on the generative model completely: the latter only guides the sampling mechanism by preferring some samples to others. This is important, because it is sometimes difficult to state a correct objective function for a certain location description, and it is hard to find an optimal solution for any function in general, so the system does not search for the optimal solution. Instead, it tries to find a good enough one, which satisfies the required constraints. The constraints are checked in the validation step and only for the sampled solution candidates. Using this approach, combining different descriptions comes down to combining the costmaps, which is done by simple multiplication of the values corresponding to the same position and normalizing the results in the end.

The example context, where this approach is tested, is the scenario of the robot autonomously setting a table. The robot finds the objects it needs for table setting on the kitchen counter and puts them at correct positions on the table. An example description of the task the robot gets is "put a knife to the right of `plate-2` in

table setting context". The robot then searches for an object of type `knife`, finds a position near the `plate-2` based on the description satisfying the constraints (e.g. the knife should not be colliding with the plate or other knives already placed near it) and places the knife at that position. The system has different definitions for each relation depending on the object, for which the pose is generated and the context of the action: a pose for a knife placed near a plate in the table setting context should be different from the position for a bowl placed near a plate in the "clear the table" context.

The robot can either perform those actions in a simulation mode to reason if the pose is good with respect to future actions, or actually execute them in real world based on the results of the reasoning. The reasoning system is a part of CRAM – the Cognitive Robot Abstract Machine [4] – which is a lightweight robot control software written in Common Lisp. It uses knowledge processing, symbolic action planning and specialized parameter inference mechanisms in the robot control plans. It is designed to be flexible and efficient, in the sense that it generates the parameters and courses of actions adapting to its environment and checks if the solutions are efficient based on fast simulation. An emphasis is put on computational efficiency: being lightweight enables CRAM to perform all the reasoning online during the execution of tasks by the robot in real world.

2. Related Work

There are many other systems where symbolic and geometric planning have been combined. For example, in [6] the planner considers both geometric and symbolic constraints, which enables it to not only produce sequences of actions but also their parameters, including the locations of the robots and objects and manipulation and navigation trajectories. Like in CRAM, the reasoning here is also based on an internal 3D representation of the world, consisting of poses of all the objects and current configurations of the robots in the world, which enables to perform reachability reasoning. The belief state, on which the reasoning is performed, is then the combination of symbolic and geometric states. To decrease the dimensionality of the search space a generative model is used, that is, first calculations are performed in certain degrees of freedom (DOF), for instance, the three DOFs of the robot base, and then the generated solutions are validated by taking into account the remaining DOFs.

In contrast to CRAM, this system is highly concentrated on trajectory planning, whereas CRAM has a very general approach and implements not only reachability, but also visibility, stability and spatial reasoning, and can easily be scaled with other concepts due to the costmap mechanism. The representation of the world in CRAM is also more accurate, as it is based on a physics engine.

Another similar approach to planning is presented in [14], where symbolic, geometric and differential constraints based reasoning is used for finding collision-free and dynamically feasible trajectories that satisfy high-level constraints. It, again, combines action planning with sampling based motion planing. The symbolic

planner is used to simplify the search space, which is represented as a second-order dynamical system. This approach uses symbolic action planning to guide the sampling mechanism, so here the motion planner queries the action planner, which makes the planning highly specialized on trajectory generation solely. Whereas in CRAM it is the other way around, so the planning can be applied to very different tasks.

The system in [10] also integrates task and motion planning, in that it uses geometric suggesters, such as simplified grasp or path planners, to infer, which actions and changes in the environment are necessary to achieve the desired goal. While this system gives promising results in the area of high-level planning, the geometric constraints are used here to improve the symbolic reasoning, so the system does not provide any means for inferring parameters of actions.

In CRAM the parameters, as mentioned before, are described symbolically and when a certain action is executed on the robot they are inferred using the reasoning mechanisms. One important way of describing the parameters is by means of spatial relations, and CRAM can infer geometric locations from those abstract descriptions, which none of the planning systems described above is capable of. This is required, for example, to be able to understand plans written as natural language instructions extracted from the Internet.

There has been a vast research done in the field related to spatial relations, however, the most work done up to now has led towards representing existing geometric relations with logical expressions, i.e. given two objects find the qualitative representation of their spatial relation [9], [1], [2]. This information can then be used to perform logical reasoning. The geometric reasoning system for a robot used in CRAM, however, has the inverse problem: having an abstract description of the relation, to find the poses that satisfy this relation. There have been a number of approaches to deal with this issue as well, one of them based on mathematical models using sets theory, as described in [5]. In this approach a set of geometric positions is defined, where a certain constraint is satisfied, and in order to find

a set where more than one constraint is satisfied, all the corresponding sets are combined using a conjunctive, disjunctive or compromise operators.

The representation used in the system described in this thesis is somewhat similar, however, having the requirement to be able to work with all the possible various location specifications (spatial relations, reachability, visibility etc.), it is slightly simpler and more general. In addition, constraint satisfiability is checked in the validation step, not during the generation of initial solutions, which, as it is shown in Subsection 7.2.2, gives a substantial increase in efficiency if the constraints are non-trivial.

3. Structure Overview

The thesis proceeds as following: in the second part an overview of background knowledge necessary to understand the rest of the thesis is presented. In Chapter 4 a brief introduction to Lisp is given together with a demonstration of some of its powerful features, and Chapter 5 explains the working principle of Prolog and its implementation in CRAM.

The third part introduces CRAM itself and examines its mechanisms of resolving symbolic parameter descriptions using the combination of qualitative and quantitative reasoning. Chapter 6 gives a general overview of the system, in Chapter 7 the symbolic descriptions of parameters are explained, Chapter 8 considers the internal representation of the world that CRAM uses for reasoning, and Chapter 9 gives an insight into the generative model of designator resolution.

The fourth part of the thesis explains in detail the mechanism of translating qualitative spatial scene descriptions into their corresponding geometric parameters. Chapter 10 discusses the mechanism implementation, and Chapter 11 presents the results of experiments performed using the mechanism.

In the last part, the final remarks are made. Chapter 12 summarizes the major results of this work, and the outlook for the future work is given in Chapter 13.

Part II.

Background

4. Lisp

Lisp is a family of programming languages, which was inspired by the lambda calculus [7] and implements many of the elements of the functional programming paradigm. The paradigm is called functional because it treats computation as evaluation of mathematical functions. Thus, the functions in the code should return the results of their calculations but not change the state of the program (e.g. change the value of its arguments or that of a global variable) or have an observable interaction with the outside world (e.g. print out data or raise an exception). In addition, the execution of a function cannot depend on the state of the program or the input from I/O devices, i.e. all the necessary variables should be passed as arguments. Such functions are called *pure functions* and are said to have *no side effects*. In this setting, it can be therefore assumed that at any point in the program a function will always return the same result if the argument values are the same. This assumption opens up many possibilities, e.g. it makes it easier to parallelize the execution of the program and perform different optimizations.

There are more than 20 known dialects of Lisp and the two major ones nowadays are Common Lisp and Scheme. CRAM is written in Common Lisp, which is an ANSI standardized dialect with a number of existing implementations, including the open source SBCL (Steel Bank Common Lisp) compiler.

Please note that Common Lisp is not a purely functional language and it does not require all of the functions of the program to be pure.

4.1. Higher-Order Functions

A function is called a *higher-order function* if it can have other functions as an input argument or return value. A standard example is a general sorting function, which takes as input a predicate that compares two elements of the list it wants to sort. For example, in C it is the function `qsort`:

```
void qsort(void *array_to_sort, size_t num_elems,
           size_t elem_size,
           int (compar)(const void *, const void *));
```

where `*compar` is a pointer to the comparison function. Similarly, in Python the function `sorted` takes as input another function `cmp`:

```
sorted(iterable[, cmp[, key[, reverse]]])
```

If now, for example, the order of sorting (descending vs. ascending) is decided by user input, we can write another function, `get_cmp`, which, depending on what the user typed in, returns a pointer to the correct comparison function that we can directly plug into the call to the sort method. As we can see, the `get_cmp` here is also a higher-order function.

The concept of higher-order functions is a part of the functional programming paradigm and all the languages implementing this paradigm support higher-order functions. An interesting example of higher-order functions that are extensively used in functional programming are `map` and `reduce`. `map` is a primitive that takes as input a one-argument function and a sequence and returns another sequence, where each element corresponds to the result of applying the function to the corresponding element of the input sequence:

```
(map 'list #'sqrt '(4 9 25))}
```

returns `(2.0 3.0 5.0)`, where `'list` is the type of the return value (it can also be `'string` etc.).

Reduce takes a two-argument function and a list as input and applies the function first to the first two elements of the list, then to the returned value and the third element, and so on:

```
(reduce #'+ '(2 6 1))}
```

returns `9` ((2 + 6) + 1).

The idea of `map` and `reduce` originated in functional programming languages but many other languages adopted them since then and included into the list of their built-in functions. Google's programming model for processing large data sets called MapReduce[1] was also inspired by those primitives.

4.2. Lambda Functions

Sometimes the function, which is an argument or a return value to the higher-order function, is used only once in the code and nowhere else. For example, let us assume that we need a function to sort a list of strings with respect to their third characters. The Common Lisp primitive `sort`[2] will do the job for us. The only thing we need to provide it with is the predicate that compares two strings of the list in the appropriate for our task way. An example implementation of the predicate could be:

```
(defun compare-by-third-char-p (some-string some-other-string)
  (char< (char some-string 2)
         (char some-other-string 2)))
```

where `char` retrieves a character from a string (please note, that in Common Lisp the indices start from 0) and `char<` is the equivalent of "¡" for characters.

Using this predicate we could then white the sorting function like the following:

```
(defun sort-by-third-character (list)
```

[1] `http://research.google.com/archive/mapreduce.html`
[2] `http://www.lispworks.com/documentation/HyperSpec/Body/f_sort_.htm`

```
  (sort list #'compare-by-third-char-p))
```

and call it like this:

```
(sort-by-third-character
  '("J.McCarthy" "G.Steele" "S.Russell"))
```

which would result in:

```
("J.McCarthy" "S.Russell" "G.Steele")
```

However, the predicate `compare-by-third-char-p` is only used in the `sort-by-third-character` function and the rest of the program would most likely access it only through that function. Therefore, instead of declaring the predicate separately and giving it a name we declare it right on the spot where it is used, utilizing the mechanism of anonymous functions of the functional programming paradigm:

```
(defun sort-by-third-character (list)
  (sort list (lambda (some-string some-other-string)
               (char< (char some-string 2)
                      (char some-other-string 2)))))
```

In Lisp anonymous functions are called *lambda functions*, the name originating from lambda calculus.

A lambda function, just as any other function, can access the local variables from its lexical scope during its calculations, as well as assign new values to them. However, for a lambda function these bindings to the variables persist between different calls. Therefore, it is possible to perform calculations on the same variable using the corresponding lambda function from different parts of the code that have access to it. The lambda functions that contain references to the variables from their lexical scope are called *closures*. They are typically used for implementing callbacks. The advantage of closures over, e.g. function pointers, is that due to the

variable bindings a closure has its own state, which is maintained between different calls.

4.3. Currying

Assume we have a function which takes as parameters two objects and performs collision detection on them. To do that it calculates the axis-aligned bounding boxes of the objects and then checks if the resulting cubes intersect:

```
(defun colliding-p (object-1 object-2)
  (let ((aabb-1 (get-aabb object-1))
        (aabb-2 (get-aabb object-2)))
    (intersect-p aabb-1 aabb-2)))
```

We would like to use this function to check if the object `plate` is in collision with some other object, namely the objects `glass`, `fork`, `spoon` and `bowl`:

```
(map 'list #'colliding-p
     '(plate plate plate plate) '(glass fork spoon bowl)
```

As `colliding-p` is a two-argument function we need to provide `map` with two lists: one with elements for the first argument of `colliding-p` and another one for the second argument values. If in our setting `plate` is colliding only with `fork` the result will be: `(NIL T NIL NIL)`.

Notice that while performing this mapping the bounding box of the plate is calculated all over again. It would be nice if we could fix the first argument of `colliding-p` to `plate` and do all the preprocessing for it before applying the second argument. This is called currying / partial application. Common Lisp does not have a built-in function for this, therefore we use a function from an external library `alexandria`:

```
(map 'list
```

```
(alexandria:curry #'colliding-p 'plate)
'(glass fork spoon bowl))
```

The function `curry` returns another function (a closure), which now accepts only one parameter – `object-2` – and `object-1` is bound to `plate`. In addition to better writing style, currying can also help to avoid calculating the same things many times with the prerequisite that the compiler used for execution is smart enough.

Coming back to the example from the previous section about lambda functions, we can show that it is possible to write our sorting function in two lines using currying:

```
(defun sort-by-third-character (list)
  (sort list #'char< :key (alexandria:rcurry #'elt 2))
```

4.4. Recursions

Extensive usage of *recursions* is another distinctive feature of functional programming, where it is the main way to accomplish iteration, as opposed to loops. Recursion may become relatively inefficient because it requires to perform multiple function calls, i.e. multiple operations on the stack accompanied by its growth. However, compilers for functional programming languages are usually highly optimized for recursions, which makes them perform as fast as looping. One example is the tail recursion optimization. If the recursive call is the last operation in the caller function then the result of the call will also be the result of the caller function. This means that all the local variables of the caller function have already been used for necessary calculations and there is no need to keep them any longer. So, instead of creating a new stack frame for the recursive call, the old space is recycled and overwritten by the new data. This gives a considerable increase in processing time and memory consumption, especially if the recursive function is highly nested. In

addition, with tail recursion optimization it is possible to create infinite recursion loops, for example, for user input or event handling without causing stack overflow.

4.5. Lazy Evaluation

Sometimes in reasoning for robot plans a need arises to process infinite data structures. For example, let us assume that the robot wants to put a plate on a table and the way the put down location is determined is just by taking a random position within the bounds of the table. If the program does not keep track of already generated positions (and most likely it will not as there are very many – as many as there are positions on the table up to a certain resolution) then if we ask for all put down locations for the plate the answer will be an infinite list.

The way the infinite data structures are handled in CRAM is by utilizing the concept of *lazy / delayed evaluation*, which is another typical element of the functional programming paradigm, also coming from lambda calculus. One example of using the lazy evaluation method is that an expression on the right-hand side of a variable assignment is only evaluated when the variable from the left-hand side is first used in the code. That is `a := func(b)` will not initiate a call to `func` until `a` gets used for the first time somewhere else in the code.

Common Lisp does not provide any functionality for lazy evaluation, so CRAM has its own little library for that. The central point of this library is the implementation of lazy lists, which are used to represent big or infinite-sized lists.

In order to create a lazy list it is required to provide a generator function, which will generate the elements of the list, initial values of the variables, on which the function depends, and the update rules of the variables. In general it looks like this:

```
(lazy-list ((variable-1 variable-1-initial-value)
```

```
            ...
            (variable-n variable-n-initial-value))
  (cont generator-function-call update-rule-for-variable-1
                                ...
                                update-rule-for-variable-n))
```

The first time when an element of the list is requested:

- the generator function is called with the initial values of the variables (the first element of the lazy list is the returned value of the generator function),
- the values of the variables are updated according to the given update rules.

On subsequent calls for new elements of the list the process will repeat with the updated values of the variables.

Let us consider an example of how to create lazy lists: the function `random-number-sequence` generates an infinite lazy list of random numbers between the given bounds (`lower-bound` and `upper-bound`):

```
(defun random-number-sequence (lower-bound upper-bound)
  (lazy-list ()
    (cont (+ lower-bound (random (- upper-bound
                                    lower-bound))))))
```

The generator function of this lazy list is:

```
(+ lower-bound (random (- upper-bound lower-bound)))
```

As we can see, all the parameters of the function are constant. Therefore, there are no input variables and there is no need to specify initial values or update rules.

In contrast to the previous example, the following piece of code defines a function `number-sequence` that returns a lazy list that contains subsequent integers from

`lower-bound` to `upper-bound`, for which we need to keep track, which integer is returned each time:

```
(defun number-sequence (lower-bound upper-bound)
  (lazy-list ((i lower-bound))
    (when (< i upper-bound)
      (cont i (+ i 1)))))
```

The update rule here is `(+ i 1)` and the generator function is identity. Please note, that this lazy list is finite and it ends when `i` becomes equal to `upper-bound`.

In addition to means of creating and referencing lazy lists, CRAM provides a number of tools to work with them: a function `force-ll`, which forces the list to expand all its values, mapping and reducing functions specialized on lazy lists etc.

Lazy evaluation is not only used in CRAM for handling infinite data structures, but it also supports the idea of the system being designed to be lightweight: expressions are being evaluated only at the moment when it is really needed.

4.6. Macros

Lisp has a Polish prefix notation, that is all the code expressions have the form of lists (they are called s-expressions). For example, the expression `(2 + 4) / 3` in Lisp will be written as `(/ (+ 2 4) 3)`. This means that all the expressions are identical, which makes parsing the code simple and, therefore, reliable in the sense that it is much harder to make mistakes during parsing in Lisp than in languages with other more complex syntax. In addition, the name Lisp comes from "LISt Processing", which in some sense reflects the fact that the main data type of the language is a list. So, the data and code are identical as well. Therefore, most of the means that are used to manipulate data can also be used to edit and generate code. All of these gives Lisp a very powerful means for metaprogramming due to its extensive macros mechanism, which makes it easy to write new domain-specific

languages embedded into Lisp.

To illustrate the usage of macros in Common Lisp let us consider the example of the primitive `loop`[3]:

```
(loop for i in list
      when (numberp i)
        when (floatp i)
          collect i into float-numbers
      else
        collect i into other-numbers
      else
        when (symbolp i)
          collect i into symbol-list
      else
        do (error "funny value in list ~S, value ~S~%" list i)
      finally (return (values float-numbers other-numbers
                              symbol-list)))
```

If we perform this loop with the `list` equal to

```
(0 3.0 apple 4 5 9.8 orange banana)
```

it will sort the values in the list depending on their type:

```
(3.0 9.8), (0 4 5), (APPLE ORANGE BANANA)
```

4.7. Packages

Common Lisp organizes code modules using packages. Each package has its own namespace such that the programmer could use the same name for different functions from different packages. The developer of a package can decide which sym-

[3] http://www.lispworks.com/documentation/HyperSpec/Body/m_loop.htm

bols (function names, variables etc.) it wants to expose to the users of his package and which ones should not be accessible. In order to use code from a package it should be compiled and loaded. If package A uses functionality defined in package B then B should be loaded before A. The loading process is either done manually or using the Common Lisp building tool called ASDF (Another System Definition Facility) [16]. It automatically compiles and loads in correct order the packages for which dependencies are correctly specified in the build files. Multiple dependency is allowed, as long as there are no loops in the dependency tree. ASDF supports incremental compiling: it compiles only those files, that have been changed since the last compilation.

4.8. SBCL

The Common Lisp compiler used for developing CRAM is SBCL [4]. It translates the Common Lisp code into native machine code, is written in Lisp and is free / open-source[5]. It provides the programmers with declarations to improve the efficiency of the code. This includes specification of optimization policies for the functions (SBCL supports all the four ANSI optimization qualities: `debug`, `safety`, `space` and `speed`), optional data type declarations (Common Lisp has dynamic typing), debugger option specifications etc. It also has a garbage collector. In fact, the concept of automatic garbage collection was first introduced by Lisp.

4.9. Why Lisp?

There is a number of reasons why Common Lisp was chosen as the language of CRAM. They are discussed below.

[4] `http://www.sbcl.org/manual/`
[5] `https://github.com/sbcl/sbcl`

Due to its powerful macros mechanism Lisp is known to be good for implementing other domain-specific languages. This characteristic was extensively used during the implementation of CRAM:

- A new reactive planning language CPL (CRAM Plan Language) was developed for CRAM. It is used for programming cognition-enabled control systems and is based on RPL [11].
- Using Lisp macros it is easier to automatically generate robot plans, as plans are chunks of code and Lisp is good for generating code.
- In addition to the macros mechanism, Lisp is good for working with symbolic data, as `symbol` is one of its built-in types. The combination of the macro and symbol processing mechanisms makes Lisp perfect for writing symbolic reasoning systems. CRAM has a custom Prolog interpreter written in Lisp that can perform both logic reasoning and use specialized inference methods for the cases when logical inference has a too large search space (it is discussed in the next section).

In terms of efficiency Lisp is generally faster than many languages, such as Python or Ruby, but, slower than, for example, C++ or Java [6]. However, CRAM runs on a robot in real-time and the robot usually has enough time to think about its future actions while performing current actions: physical execution of actions is in most cases slower than the computation. In addition, as mentioned before, SBCL has many possibilities to improve code using the optimization declarations. And also, the relatively low computational efficiency is partly compensated by the software development efficiency due to the fact that programmers can implement much functionality in a few lines of code.

Finally, the last but not the least reason for choosing Common Lisp is that it is a very powerful and flexible language and is exciting to program with. In addition

[6] Based on the benchmarks from `http://shootout.alioth.debian.org/`.

to supporting functional programming, it also has the advantages of the object-oriented programming languages, as it supports such concepts, as classes, multiple inheritance, function overloading etc. through Common Lisp Object System (CLOS), which is its built in facility. It also supports hot swapping: it is possible to make changes in the code and recompile it at runtime and the changes will immediately take place. If a CLOS class is changed during execution all the class' instances will adopt to the changes.

5. Prolog

Prolog is a logic programming language. It is mostly declarative but has some imperative programming features as well: it has a purely logical subset, which prohibits side effects, plus a number of imperative features, such as a `cut` operator, which can be used to alter the control flow of the program, and routines for performing input/output operations, accessing the graphics card, using the network etc. It was inspired by first-order logic (though it also has a number of predicates from higher-order logic), so the program is represented in terms of relations. The latter can be divided into two groups: rules and facts. Rules in Prolog are represented as following:

$\texttt{BigGoal :- Goal}_1\texttt{, ..., Goal}_n\texttt{.}$

which can be read as "`BigGoal` is true if $\texttt{Goal}_1$ and ... and $\texttt{Goal}_n$ are true" or, equivalently, "to prove `BigGoal`, prove $\texttt{Goal}_1$ and ... and $\texttt{Goal}_n$". The left-hand side of the rule is called head and the right-hand side - body. All the goals have a form of predicates, which are boolean functions that have zero or more arguments. The number of arguments is called the arity of the predicate, e.g. `color(Object, Color)` is a 2-arity predicate.

The facts are predicates, which are always true, for example:

```
color(sky, blue).
```

which is equivalent to the rule:

```
color(sky, blue) :- true.
```

All the facts and rules known to the program build the knowledge database, to which Prolog states queries in order to prove goals. To prove a goal means to find correct assignments to all the variables in the goal that are unbound. For example, if we have the relations describing some family tree, proving the goal

```
{mother(diana, Child).}
```

with `Child` as an unbound variable and `diana` as constant, would give us a list of Diana's children if such exist, otherwise the goal would be proven to be false. If both arguments of the predicate would be bound, proving it would just return `true` or `false`.

Prolog is usually used for representing knowledge. It maintains a knowledge database that we can query to get specific data. But, in contrast to traditional databases, it can also infer knowledge that is not explicitly given in the knowledge base.

5.1. CRAM Prolog

CRAM has its own custom written full featured Prolog interpreter, implemented in Common Lisp. It is used in CRAM to perform logic-based inference, so the CRAM programs written in Lisp send certain query statements to Prolog, which in its turn returns as a result of its calculations a set of suitable for the statement parameter assignments in a form of objects of Lisp datatypes, or tells if the statement is true or false according to its knowledge. Its working principle is the same as that of traditional Prolog interpreters, but it has additional support of lazy lists to deal with data structures of infinite size that are to be returned to the Lisp code. The syntax of CRAM Prolog is different from the traditional one in that it has the same Polish notation as Lisp.

Let us consider a simple world where we only have a few furniture items, a floor, a number of tableware objects, such as cups or bowls, and a robot, and try to

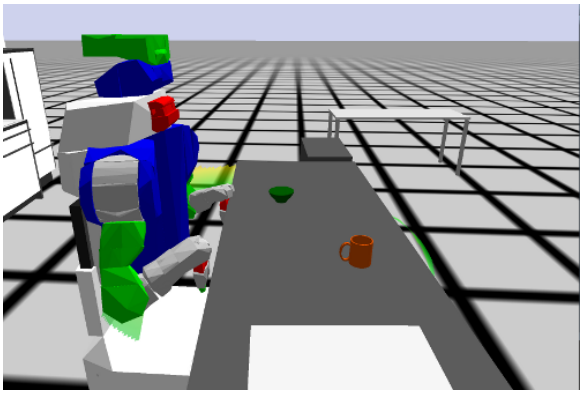


Figure 5.1.: A simple world with a few furniture items, a floor, a mug and a bowl standing on a table and a robot, which is in contact with the table.

use Prolog to perform reasoning on this world. One possible state of such a world, which we will call `visualized-world-state`, is visualized in Figure 5.1. For example, let us take the relation `supported-by`, which specifies if one object is supported by another in a particular state of our world (e.g. on the figure a mug on the table is supported by the table and the robot is supported by the floor). It can be defined as following:

```
(<- (supported-by ?world-state ?top ?bottom)
  (contact ?world-state ?top ?bottom)
  (above ?world-state ?top ?bottom))
```

where the first inner s-expression (the first line) is the head of the rule, and the other two – the body with an implicit logical `and` combining them. Each expression starts with a symbol representing its name followed by its arguments. Symbols starting with a question mark are variables.

If we now state the following goal:

```
(supported-by visualized-world-state ?supported-object table)
```

Prolog should bind the variable `?supported-object` to any object that is in contact with and above the table in `visualized-world-state`, and a list with all

the bindings (assignments) of `?supported-object` should be returned. As can be seen on the Figure 5.1, the orange mug, which we will call `mug-1`, and the green bowl `bowl-1` are standing on the table, so these two will be the valid bindings for `?supported-object`.

CRAM Prolog, just as Prolog in general, uses an approach similar to depth-first search to prove goals. The search tree for `supported-by` for the world state from Figure 5.1 is presented in Figure 5.2. To build the tree, Prolog first puts the goal itself as a root node. Then it expands the goal based on the body of the rule (or rules) it is described with. In our case, the body consist of two predicates, namely, `contact` and `above`, each of which is considered in turn. Prolog tries to find variable bindings for all the predicates with unbound attributes. This process is called unification. In this case, the only unbound variable is `?top` aka `?supported-object`. The other two variables `?world-state` and `?bottom` are bound to `visualized-world-state` and `table` respectively.

The interpreter starts by trying to prove the predicate `contact`. For each possible variable assignment of the predicate it builds a node in the tree, and the parent of these nodes is marked as a choice point. In our case, the predicate `contact` gives three possible variable assignments for `?supported-object`, namely `Robot`, `mug-1` and `bowl-1`, as those are the objects that are in contact with `table` in `visualized-world-state`. Then it proceeds by expanding all the predicates that are described by rules in a similar fashion until it reaches the leaf nodes – the predicates with all the variables bound, i.e. facts. If it is impossible to prove a certain predicate, that is it is impossible to find variable bindings that do not contradict with the bindings from parent nodes of the tree or if the predicate cannot be proven using the knowledge base of facts, it means that it was impossible to find a solution on this particular branch of the tree and the algorithm backtracks – goes up the tree until the closest choice point – and continues the depth-first search. The algorithm can stop as soon as one solution for the goal is found that satisfies all the constraints, i.e. all the variables on the branch of the tree from the

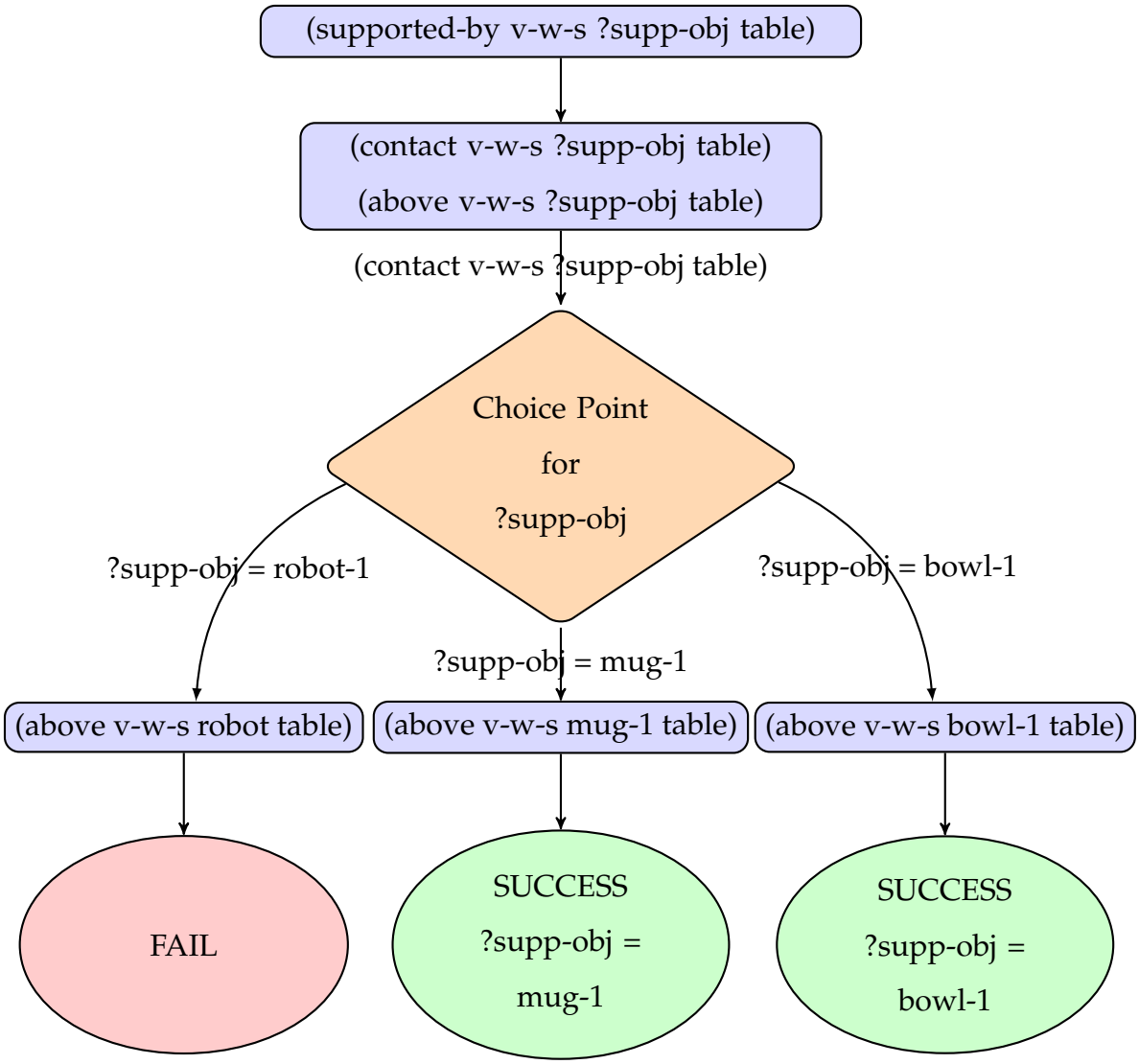


Figure 5.2.: Prolog search tree for a query `(supported-by v-w-s ?supp-obj table)`, where `v-w-s` denotes the `visualized-world-state` and `?supp-obj)` is a shortened version of `?supporting-object`.

goal to the solution are correctly bound, or it can expand the whole search tree and find all the valid solutions, which can in fact be of infinite number. In our example, the left branch of the tree fails, as `Robot` is not `above table`, and the other two branches result in corresponding bindings for the variable `?supported-object`. Thus, our Prolog predicate `supported-by` gives the following result:

```
(((?SUPPORTED–OBJECT . PLATE−1))
 ((?SUPPORTED–OBJECT . PLATE−2)))
```

which is an associative list of valid variable bindings.

This mechanism of searching for valid parameters of predicates that satisfy certain constraints is a powerful tool that is used in CRAM to perform reasoning. Unfortunately, pure logical inference can only perform qualitative reasoning. However, as many of the parameters of robot plans are actually quantitative and symbolic reasoning cannot be used for them, CRAM Prolog has means to combine traditional logic reasoning with quantitative domain-specific inference methods that can, for instance, be based on geometric calculations. It is used in CRAM for finding valid parametrizations of actions, answering queries about the state of the world and the relations of objects in it, finding the correct sequences of actions required to achieve a certain goal etc.

As an example, let us come back to the predicate `supported-by`. In order to prove it Prolog needs to consider two other predicates – `above` and `contact`. One possible implementation of `above` is to keep a list of all the object pairs that satisfy the above relation for every possible state of the world and use Prolog search based on this knowledge. If the world contains many objects and we have many different states of the world to reason about, the search space explodes.

A better approach would be to use a specialized inference method based on geometric reasoning, as the `above` relation is actually a geometrical concept. Thus, the predicate can be defined in the following way:

```
(<− (above ?world−state ?obj−1–name ?obj−2–name)
```

```
(object-geometry ?world-state ?obj-1-name ?obj-1-geometry)
(object-geometry ?world-state ?obj-2-name ?obj-2-geometry)
(lisp-pred above-p ?obj-1-geometry ?obj-2-geometry))
```

where the predicate `object-geometry` relates names of objects to the information about their geometry, including their shape and pose in the specified world state. `lisp-pred` denotes that `above-p` is actually a Lisp function, which returns a boolean value. One possible implementation of `above-p` is to find the axis-aligned bounding boxes of the two objects and compare the bottom of one box with the top of the other one. This solution is more intuitive and efficient for complex worlds than the approach with pure Prolog that was proposed above, which illustrates the advantage of specialized inference methods over logical inference for some domains. The predicate `contact` can be solved in a similar way: we can perform collision detection using, for instance, a game physics engine.

If the goal predicate is described through many rules, i.e. there are multiple rules that have the same goal in their head part, Prolog considers each rule separately and the solutions from all the rules are concatenated. In this respect, the useful feature of CRAM Prolog is that different rules can be declared in different Lisp packages: some packages can have simple solutions and some – more complex ones. To use certain solutions the packages in which they are implemented should simply be loaded into SBCL. The solutions from all the loaded packages are combined with each other. So, if for a certain task a lightweight solution is sufficient, only the packages with simpler implementations should be loaded. For instance, let us assume that there is a Lisp package for performing various cooking tasks with the robot. For any of the tasks, it is always first checked if all the necessary ingredients are at robot's disposal by querying Prolog with the following predicate:

```
(object-at-disposal current-world-state the-ingredient
                    ?ingredient-location)
```

which should find correct bindings for `?ingredient-location` if `the-ingredient` can be found by the robot in `current-world-state` and should fail otherwise. If the predicate fails for any of the ingredients, the cooking cannot be performed. Let us assume that the default implementation is using the camera of the robot to look around and try to detect the necessary object.

If now we want to extend the system to make the robot also open shelves and cupboards and search for the object there, we need to create a Lisp package and implement there a new definition of the predicate `object-at-disposal`, which realizes this approach. If we now load this package together with the other ones, even if the query to `object-at-disposal` will not be able to find any bindings for `?object-position` using the old implementation, Prolog will proceed with the search using the new implementation, which may be successful. No changes are required to be done in the old packages, loading the new package into SBCL is sufficient to start using the new functionality.

Part III.

Cognitive Robot Abstract Machine

6. System Overview

CRAM is a planning and reasoning system written in Common Lisp, which incorporates Prolog based qualitative reasoning with specialized quantitative inference methods. The robot plans are written in CPL – the CRAM Plan Language, which is used for developing plans for mobile manipulation and AI based robot control. It provides means for writing commands that can be executed in a reactive and concurrent way. It represents goals, actions and failure descriptions using Prolog, which enables reasoning about the course of plan execution. The actions are represented as symbolic rules used for updating the world states.

The symbolic planer uses a plan library and Prolog based inference to find the course of actions necessary to achieve a specified goal. The parameters of actions are represented using the so-called *designators*, which are symbolic key-value pairs that are qualitative descriptions of the parameters. They are resolved by querying the reasoning engine. Querying enables the system to perform calculations only when they are actually indeed. So, using the concept of designators we can represent the objects in the world, tasks, etc. in an abstract way, and ground the representations at run-time when a specific designator is needed in order to perform an action.

The designators are symbolic representations of actions, objects or locations, which are compiled into Prolog programs that are executed in order to resolve them. If a designator represents a quantitative concept, specialized reasoning mechanisms are used. For instance, location designators are resolved using an accurate geometric representation of the world. The specialized methods of CRAM

can perform visibility, stability, reasoning, as well as infer locations from spatial relation descriptions.

CRAM uses a generative model to resolve designators, i.e. a number of proposed solutions are generated and then validated to check if they are appropriate in a particular environment state and action execution context.

In order to find if a designator solution is good with respect to future actions of the robot, CRAM can perform a lightweight simulation, called *temporal projection*, record the timeline of events that happened during it, and using behavioural flaw specifications evaluate the quality of the solutions with respect to that flaws. For example, a behavioural flaw can be the distance that the robot should drive during the execution of a task or a number of broken mugs that are expected is a specific designator solution is used to parametrize the plan.

The timelines of actions executed by the robot controlled by CRAM in the real world are stored in a form of a so called *execution trace*, such that the robot could then analyse its past actions and learn from experience [12].

Currently, the most of the functionality of CRAM for using it for real world action execution is implemented for the PR2 robot[1]. However, it has a flexible software architecture and can be easily extended to be used with other robots. For each robot a specific set of functionalities should be implemented. In CRAM those are grouped into so-called *process modules*, which include perception, navigation, manipulation and pan-tilt unit control modules. Altogether they constitute the executive of the robot. In order to use a certain executive it is only needed to load the Lisp package / packages with the corresponding implementation and register the process modules in the system. No changes in the system code is necessary.

CRAM is mainly written in Common Lisp and uses ROS as a middleware for communication between its different modules, and also as a convenient software development tool, e.g. to easily share code between different developers [15]. It is

[1] `http://www.willowgarage.com/pages/pr2/overview`

open-source[2]. As a software toolbox it is designed to satisfy the following requirements [4]:

- It should be lightweight to be able to run on a robot in real-time.
- It should be written in standardized languages to be easily used by other developers in the community.
- It should have a flexible software architecture to make the extension with additional modules easy.

[2] http://www.ros.org/wiki/tum-ros-pkg

7. Designators

7.1. Concept

The whole system is based on the concept of designators, which are symbolic descriptions of building blocks needed during execution of the robot plans in CRAM. This enables to write the plans in an abstract way, e.g. a plan make-me-tea can be described as "put a tea bag in a mug and pour hot water in that mug". When the plan is actually executed all the designators are resolved using robot's environment, such that the mug from the plan would be grounded with an actual mug from robot's environment, which can be done using, for instance, robot's perception. The action "pour" will then also be grounded to a specific motion of the robot's joints.

The designators are represented in plans as abstract symbolic descriptions in a form of a list of designator properties (e.g. for a red cup it would be `((type cup) (color red))`) and are resolved at run-time and only when the program reaches a point in the plan where there is an unresolved designator. If there exist different methods to resolve the same designator, the system combines them, such that if one method fails the solutions from the others would be used.

There are three types of designators in the system at the moment: action, location and object designators. Resolving an action designator gives correct parametrizations of actions to execute. If an action is about manipulation, resolving the designator could give a trajectory of the robot joints that can be then used to execute

the action. For designators describing navigation actions, resolving gives the trajectory for the robot base. Location designators are resolved into 3D poses, such that when executing a plan "put a fork to the left of plate" the robot would know exactly at which position and orientation it should put the fork with respect to the plate. Object designators are usually resolved through robot's perception: it perceives the environment and creates in the belief state an instance of each perceived object, which is then bound to a designator, if the symbolic description corresponds to the properties of the object.

7.2. Implementation

A designator is represented in CRAM as a CLOS class with the following slots: `timestamp`, `description`, `parent`, `successor`, `effective` and `data`. Here, the `description` is the list of properties which give the abstract representation of the designator. The result of resolving should go into the `data` slot of the corresponding designator. Upon resolving the current system time gets assigned to the `timestamp` slot and the `effective` tag gets assigned a `true` value to indicate that this particular designator is grounded. The designators are read-only, so once they become effective, the `data` slot cannot be changed any more. That way the `timestamp` always denotes the time of resolving of the designator. When a new up to date solution for a designator with the same symbolic description is needed, the `(next-solution desig)` function is called, which creates a new designator with the same properties as `desig` has, but the `data` slot of the designator gets assigned a new value as a result of resolution.

During the process of generation of the new designator object, the `parent` slot of the new one gets assigned a pointer to the old designator and the successor of the old one gets assigned a pointer to the new designator. Thereby, a chain of valid grounded representations of the same abstract designator is generated where each of them has a distinct timestamp with the generation time. In order to check if two

effective designators were actually resolved the same way, they can be `equate`-ed. This mechanism makes it possible to account for and keep track of the changes in the environment.

The specific designators – action, location and object – are subclasses of `designator`.

7.2.1. Action Designator

Resolving action designators is done with Prolog, namely the predicate `(action-desig desig ?goal)`. An example of such a predicate implementation is given in Listing 7.1. The resolving function leaves `?goal` in the predicate unbound, so Prolog binds it to the correct parametrization of the action including the type of a manipulation task that `desig` designates. In our example it is the action called "put-down" and the parameters are `?obj`, `?loc`, `:right` and `?obstacles`. In case of a navigation action the goal would probably only contain the position where we want to move the robot, as the goal position is the only parameter needed in order to execute a navigation task.

Each Lisp package which deals with specific action designators is supposed to implement the `action-desig` predicate. Prolog will then unify all the goals coming from the different implementations inside `?goal` in a form of a lazy list, which will then be assigned by the resolving function to the `solutions` slot of the action designator object. The first of these solutions gets assigned to the `data` slot of the designator.

```
(<- (action-desig ?desig (put-down ?obj ?loc :right
                                   ?obstacles))
    (trajectory-desig? ?desig)
    (desig-prop ?desig (to put-down))
    (desig-prop ?desig (obj ?obj))
    (desig-prop ?desig (at ?loc))
```

```
    (obstacles ?desig ?obstacles))
```

Listing 7.1: Implementation of `action-desig` for a `put-down` manipulation task.

What is done with the list of solutions, i.e. in our example how exactly the put-down action is executed and how the parameters are used, is specified in the corresponding parts of the system where the action designators are used, for example, in the plans. The same goes to the implementation of the corresponding `action-desig` predicate that we have in the listing. So the implementation of the concept of designators, of course, only provides the protocol for their usage and no specific implementations.

7.2.2. Location Designator

The way the location designators are resolved is by using the so called location generator functions. Each of these functions gets as input the location designator we want to resolve and gives back a lazy or a normal list of solutions. The solutions will usually be poses in 3D but not necessarily, as location designator is a general concept.

The system maintains a list of all the generators known to it and provides a mechanism to register new ones. When a location designator needs to be resolved, it is passed to all the generators and the results from them are collected together in a lazy list of solutions. Each generator has a priority value assigned to it, so the solutions from generators with smaller values are put at the beginning of the list. The functions that generate infinite solutions should therefore be assigned highest priority values, otherwise they will not give other functions a chance to generate a solution. Of course, a generator does not need to be able to resolve any kind of location designator with any combination of designator properties. Instead, it should be specialized on one certain type of location designators and return `nil` in case of others.

Besides the location generators, the system also provides a way to discard the generated solutions that are not applicable in the particular setting of robot's environment. This validation is done using the list of location validation functions, which get as input the designator and the solution given by the generators, which is then either accepted or rejected. So the generated solutions are taken from the lazy list one by one and passed to the validators. If a solution gets rejected the next one on the list is tried out. There is a maximum number of retries (e.g. 200) after which the system stops looking for more solutions and considers the designator resolution to have failed.

The location generators are supposed to give more general solution candidates for the designator resolution and the validators refine this solutions by rejecting some of them. In order to understand which properties of the environment need to be considered on the generator side, and which ones – during validation – let us look at an example. We would like to resolve the following designator:

```
(designator location ((left-of plate-1)))
```

where `plate-1` is an object from our belief state, the pose of which in the world we know exactly. We would like to use the resolved pose as a put down location for another object, e.g. a fork. The location generator for this designator would then generate an infinite number of poses which all lie to the left of the plate in some particular context. However, not all the locations are good. If there is already one fork lying on the left of the plate we would not like to put the second fork on top of it. Therefore, we use the location validator to reject all the poses which are in collision with other objects. One could argue that the location generator could take the other objects into account while generating the poses, however, the collision check is rather expensive and there are many if not infinite number of solutions that the location generator could give. Therefore, we only perform the check on the validator side for a few generated solutions until we find a suitable one.

The lazy list of solutions from all the generators for a particular designator are

kept in the `data` slot of the designator object.

7.2.3. Object Designator

For the object designators there is no definite mechanism of resolution. They are supposed to be resolved externally, i.e. the `data` slot should be set from outside. The information put into the `data` slot has to be of `object-designator-data` type where the external module sets the pose and the identifier of the object that it considers that the particular designator refers to. The `object-designator-data` class can, of course, be extended with additional slots using the inheritance mechanism. An example of an external module to resolve a designator is the perception of the robot.

7.2.4. Packaging

Whenever support for a new designator property needs to be added to the system, the module which should be responsible for the property should declare it and implement the functionality for resolving designators with such a property. However, different modules of the system, i.e. different Lisp packages, need to be able to use the designator resolving functions defined in other packages. Moreover, it should be possible to combine different implementations of the same designator property. For example, let us assume that we want to resolve the designator

```
(designator location ((left-of plate-1) (for fork)),
```

which means we would like to have a location to put down a fork that is to the left of `plate-1`.

Assume that the package of the system, which is responsible for spatial relations, returns for this designator the solutions as on the Figure 7.1 (left), where the red color depicts locations directly to the left of the plate and the more the position deviates from this direction the closer the color gets to purple on a hue color scale.

This package is only responsible for resolving spatial relations properties and has no distinction between different objects.

Now we load a different package, which has knowledge about correct object placement in the context of table setting. This package would know that if a fork is placed to the left of a plate then only the locations very close to the plate would be suitable and all the locations far from it are wrong in the particular context. Therefore, this package extends the definition of `left-of` from the spatial relations package and adds a distance criterion to it, as shown on Figure 7.1 (center): the red color represents locations near plate. The solution space resulting from combining both definitions is shown in Figure 7.1 (right).

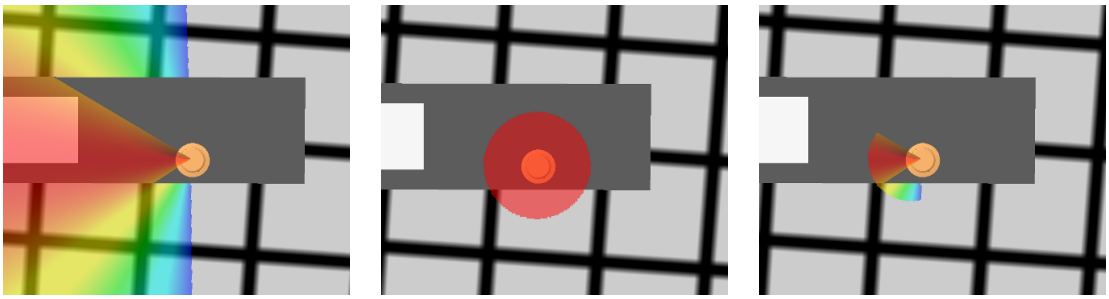


Figure 7.1.: Different implementations of designator property `left-of` and their combination.

In order for the designator resolution mechanism to easily combine different implementations of the same designator property, in our case it was the property `left-of`, both packages discussed above should refer to the same concept. Therefore, all the designator properties are kept in one common package, namely `cram-designator-properties`, and the system provides an easy mechanism to extend this package with new properties.

8. Reasoning World

In order to reason about different properties of the world, think ahead of time and learn from past experience, the robot should have an internal virtual representation of its real environment and some understanding of how the objects in the environment interact with each other. The internal representation of the world can be seen as a picture of the world in robot's head: each change in the real world should be reflected in that internal picture. In order to think ahead of time the robot should be able to imagine different situations, therefore it should be able to have more than one different world representations in mind at the same time: one representing the current state of the environment and others for imagined states.

In CRAM the knowledge about the interaction of objects in the world comes from the Bullet physics engine. It is an open source library written in C++, which, in order to be used in CRAM, was wrapped by the Common Foreign Function Interface for Lisp (CFFI)[1]. Bullet has functionality for performing collision detection on the objects in the world and means for performing rigid body dynamics calculations. It also has a soft body dynamics engine, which is not used in CRAM at the moment. Noteworthy, Bullet is just a physics library, which does not provide any means for visualization. In CRAM all the visualization is based on OpenGL. On Figure 5.1 such a visualization is shown for a certain state of the reasoning world.

Bullet has its own representation of the world to perform dynamics calculations on it. In the Bullet world objects are regarded as rigid bodies that have some forces and torques applied to them. There is a certain gravity force that affects all the

[1] `http://common-lisp.net/project/cffi/`

objects. Some rigid bodies have constraints on them, for instance, if they are connected via joints, which is useful to represent doors, robot links etc.

Each rigid body has a certain name, a mass and a state of motion, which contains information about its pose in the world. In order to perform collision detection, each object has a certain collision shape assigned to it, e.g its convex hull, a box, a cylinder, etc. To represent its dynamic state, each object has information about the overall force and torque applied to it, as well as the linear and angular velocities.

The representation of the world in Bullet comes to specifying a list of all the rigid bodies that are present in it, all the constraints that the objects have between each other and a gravity vector. After performing collision detection, the representation gets supplemented with information about colliding objects and the sets of points where that objects intersect.

Having Bullet gives the robot knowledge about what the dynamics of its environment are. However, one object can consist of multiple semantically connected rigid bodies, as, for instance, a refrigerator has a door and a container part or the robot itself consists of multiple links. So CRAM extends the Bullet world with the concept of an object that can consist of more than one rigid body and has a certain semantic meaning and use.

There are two ways a corresponding representation of a real object from robot's environment can be created in the reasoning world of CRAM. One is the KnowRob knowledge base [19]. It has a semantic map of all the static objects in the environment, as, for example, furniture in the robot kitchen (Figure 8.1 visualizes the semantic map of a kitchen from Figure 1.1 (left)). Each object in the semantic map has a number of properties associated with it: a pose, a shape, whether it can be used to store other objects or not etc. KnowRob does not only provide information about the objects that are already present in the environment, but it also has a database of common sense knowledge, such as, for instance, that cups can be used to contain liquids. The knowledge base represents in some sense the long-term memory of the robot. It has the information about the world that does not

frequently change.

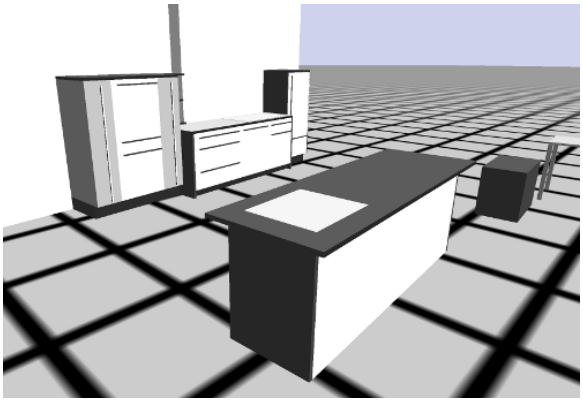


Figure 8.1.: Visualization of a semantic map of a kitchen.

The other source of knowledge about objects that the robot can use is, of course, perception. Using 3D perception the robot can acquire a point cloud of the object, triangulate it into a mesh and add the new object into the reasoning world.

In addition to the point cloud interpolation method, CRAM has the possibility to add new objects to the reasoning world through the ROS package `household_objects_database`, available as a ROS service, which contains 3D models of various common household objects sold in major retailers (such as IKEA). Each object also has a number of valid grasps for the PR2 robot that can be applied to the object. So, as soon as the type of the object is recognized by vision, the information about its shape and the corresponding grasps can be taken from that database.

There is a number of different types of objects in CRAM. There are methods to work with articulated objects as cupboards with doors: to acquire information about the constraints of their joints from the semantic map and to open and close them in the reasoning world. There is a class of household objects to represent objects that the robot can manipulate: grasp, pick up, move etc. Each household object has a certain type and each type has a certain grasp associated with it.

The robot itself is an object of class `robot-object`, which has a hash table of

all the links of the robot, the states of its joints and a list of objects attached to it. If the robot grasps an object, the object gets attached to the robot gripper and whenever the latter changes its position in the reasoning world the attached object moves together with it. The information about the mechanical architecture of the PR2 robot is stored in a URDF (Unified Robot Description Format) file[2], which can be parsed and uploaded to the ROS parameter server to be used in CRAM. The robot can be moved around in the reasoning world: it can change the position of the base and the states of the joints (the joint angles), it can move the pan-tilt unit to point in a certain direction etc.

Currently it is possible to reason about the following aspects of the world using CRAM: reachability, visibility and stability. For example, it is possible to find good put down locations on a table for a cup with milk such that it would be well reachable for a robot while it is cooking, but not make the manipulation of other objects more complicated. Or it can be used to find a position on the table where to put a pot, such that it would not occlude the cup with milk, because then the robot would not be able to track it any more. If there is not much space on the table the robot might want to put the cup on one of the empty plates on the table, then the reasoning engine can find the correct pose for the cup, such that it would stay stable on the plate. The implementation of this is discussed in the next section in scope of the usage of costmaps for resolving location designators.

All the reasoning methods in CRAM have a reasoning world object as input parameter, which makes it possible to work on different world states at the same time. There are also methods to copy a world object, restore a world from a world state object etc.

It is worth mentioning that the Bullet engine is not very precise: the shapes of the objects are rather rough approximations of their real world prototypes and the physics simulation is not completely accurate. However, the prediction can never

[2] `http://www.ros.org/wiki/urdf`

be 100% accurate anyway, because of the imperfections in perception and random factors of the environment. On the other hand, Bullet has high computational efficiency, required for running CRAM in real-time, so it sacrifices accuracy in favour of higher performance.

9. Costmaps

9.1. Motivation

During execution of plans the robot manipulates many different objects and moves itself from one place to another. Finding good poses where to stand or where to put objects is rather important for the efficiency of the plan execution. The resolution of location designators discussed so far is based on location generator and validator functions. Generators take a designator and return a list of solutions. The solutions are sorted based on the priority of the function that generated them. Each element of the generated list is then passed to the validator functions together with the corresponding designator, and the validator accepts or rejects it. If the number of rejected solutions exceeds a certain maximum number the system prompts about designator resolution failure.

All the generators and validators are kept in the corresponding global variables and are all combined to be used during designator resolution. There are macros to register the functions: `(register-location-generator priority function &optional doc)` and `(register-location-validation-function priority function &optional doc)`. When the function `reference` is called for a location designator, the registered generator functions are called one after another using lazy evaluation.

In general, the simplest way to generate solutions is to return all the positions available, e.g. points in the room, in which the robot is standing. The search would

then stop as soon as a valid solution is found. For example, to find a pose where to put a fork, such that it would appear to the left of a plate with respect to some reference point and would not be in collision with other objects, we could consequently put the fork in each point of the room in the reasoning world and check if the coordinates of the two objects have correct correlation and if it is physically possible to put the fork at the given position. If a solution exists, we would eventually find it. However, this method is very costly. Therefore, we use the generative model to guide the search by only generating those solutions, which are already to the left of the plate. In addition, we could say that the positions directly to the left are more preferable than those to the left and behind of the plate. Therefore, it is necessary to not only have a way to assign priorities to the generating functions but also to the solutions that they give.

Another requirement on location generators is that there should be a mechanism to easily combine solutions from different location generators specialized on certain designator properties. For example, the location designator for the location of a glass near a plate in table setting context would be:

```
(designator location ((right-of plate-1) (behind plate-1)
                      (near plate-1) (for glass-1)
                      (on counter-top)))
```

One way to resolve this designator is to write a generator function, which would take all of these properties into account. However, if it was possible to write specialized generators and then combine their solutions the number of necessary generator functions to resolve designators with all the different combinations of properties would be much smaller.

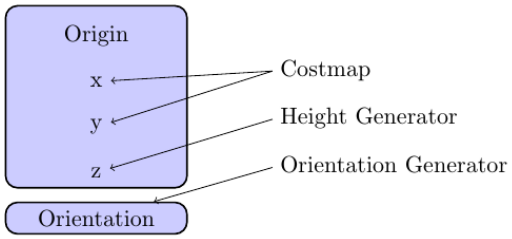


Figure 9.1.: Different 3D pose components and their corresponding generators.

9.2. Implementation

The way all these requirements on generator functions are fulfilled in CRAM is by using location costmaps. The generation of a solution pose is then divided into three stages illustrated on Figure 9.1: first a 2D `(x, y)` position is generated based on the costmap, then the `z` coordinate is added to the 2D position based on height generators, and finally, the correct orientation for the object that should be assigned the resulting pose is found using orientation generators. The different generators are needed to reduce the search space of solutions. Thus, instead of performing the search in six dimensions (three for position and three for orientation), first, a valid solution in two dimensions is found and then the missing components of the pose are generated correspondingly.

The costmap is a 2D grid, where each point `(x, y)` has a cost value associated with it. The grid is defined over the whole area of manipulation of the robot, for example, the area of the room where the robot is standing. A costmap for locations where a robot can stand without colliding with other objects is visualized on Figure 9.2. In most cases the costmap is used as a 2D probability distribution function, from which random samples are taken. The coordinates of the sampled grid cell are then the `(x, y)` coordinates of the solution for the location designator. With random sampling the higher the costmap value is, the higher is the probability that the solution with this value will be chosen. So the costmap generators should

assign higher values to cells whose coordinates are more preferred as solutions. A costmap generator is a function that takes as arguments `x` and `y` coordinates of the costmap and returns a value (from the interval `[0, 1]`) that should be put in the corresponding cell of the grid.

Figure 9.2.: Locations for a robot to stand where it would not collide with other objects – shown in red.

The parameters of the grid are `width` and `height`, which describe the size of the grid in pixels, `origin-x` and `origin-y`, which are the coordinates of the point `0.0` of the grid in the world frame, and `resolution`, which is the size of one grid cell in pixels.

Poses resulting from resolving a location designator should have a 3D position and an orientation. As mentioned before, sampling from the 2D grid gives only the `x` and `y` coordinates. The `z` coordinate is given by a height generator and the rest comes from the orientation generators. A height generator is a function that takes two arguments – `x` and `y` and returns the `z` coordinate for the location designator solution. For example, if the designator has the property `(on counter-top)`, the

corresponding height generator should return a value equal to the `z` coordinate of the surface of the counter top. The orientation generators work similarly: for each point inside the grid they return orientations represented as quaternions. For rotationally symmetric objects they would probably return an identity rotation. However, for example, for a fork that should be put near a plate in a table setting context they should generate correct orientations, such that the fork would be aligned with the table it would be lying on. CRAM has a class called `2d-value-map` containing another 2D grid, which is a generalized version of the costmap grid: it associates with each grid cell an object. This class is used to represent the maps of the height and orientation generators. For the height generators the `2d-value-map` contains `z` values and for orientation generators – quaternions.

All the different generators discussed above are combined to create 3D poses through the class called `location-costmap`. It has the following slots:

- `cost-map`, which is the 2D grid used for generating `x` and `y` coordinates
- `cost-functions`, which is a list of costmap generator functions together with their corresponding names
- `height-generator`, which is a callable object implementing the height-generator functionality
- `orientation-generators`, which is a list of callable orientation generator objects

The generator functions are added to a location costmap object using the corresponding registration methods:

- `(register-cost-function map function name)`
- `(register-height-generator map function)`
- `(register-orientation-generator map function)`

Each cost function from the `cost-functions` list generates its own costmap values. In order to combine the values from different generators, they are multiplied with each other and the resulting costmap is then normalized to have a sum of all its values equal to 1. So, when a location costmap object is created the grid is first empty, then, when the `get-cost-map` function is called for the first time on the object, for each grid cell the corresponding costmap values are calculated and then multiplied with each other. Therefore, if one of the generators assigns 0 to any of the grid cells there is no point to continue going through the values from the rest of the generators. For this reason the generator functions are assigned names, and each name has a corresponding priority value. In order to save computational effort the generators that assign the most 0 values should have the highest priority (please note, that for costmap generators higher value means earlier evaluation, which is the opposite of how the priorities work for the location generator functions). When the costmap grid for a certain location designator is generated for the first time it is visualized using the OpenGL visualization of CRAM.

One location costmap can have only one height generator, therefore, registering a new height generator always overwrites the old one. If a location costmap object does not have any height generator registered the default value is taken for the `z` coordinate of the pose, which is `0.0`.

Orientation generators take a third argument in addition to the `x` and `y` coordinates, which is an orientation returned from the previous generator in the list. For the first one in the list the identity rotation is passed by default. The results from orientation generators are combined by calling each generator in turn and passing the return value of one generator as input argument to the other one. The result from the last generator is then the overall combined orientation. If a costmap object has no orientation generators registered, the default identity rotation is taken.

In order to use the costmap mechanism for resolving location designators it is necessary to define a location generator function based on that mechanism and register it in the system. This function is called `location-costmap-generator`

and has priority value 20. There is also a corresponding validation function called `location-costmap-pose-validator`, which discards all the poses that have a costmap grid value less than a certain threshold (currently it is 20%), as they are considered to be bad solutions. These functions are automatically registered in the system when the `location-costmap` Lisp package is loaded into SBCL.

The function `location-costmap-generator` calls all the generators and combines them in order to get a valid 3D pose. First it generates the 2D costmap grid using the (`merged-desig-costmap ?designator ?costmap`) Prolog predicate. Next, it takes a random sample from the `?costmap`. Then it forgets about the value that the point had in the costmap grid and uses the height generator to find the `z` coordinate for the 2D point. Finally, it generates the orientation and creates the full 3D pose, which is the solution to the location designator. It is, of course, as any other solution, passed to all the validation functions afterwards. If it is rejected by them the cycle repeats with a new random sample from the costmap.

CRAM provides a number of Prolog predicates to work with the location costmaps: (`costmap ?cm`) generates a new costmap objects and binds it to `?cm` if it is not already bound, (`costmap-add-function ?name ?function ?costmap`) adds a costmap generator to the object `?costmap` etc. It also provides a number of costmap generator functions, which are sometimes called cost functions. There is a cost function that evaluates for each coordinate point the value of a Gaussian with certain mean and variance. The range cost function generates a circle with given radius and center with all values equal to 1. The axis-boundary cost function returns 1 for all the coordinates that are on the given side of the given axis. There is a cost function for generating padded costmaps etc. There is also a number of orientation and height generators, such as a function to return orientations facing towards a certain point or to create heights based on the semantic map.

In order to use costmaps for resolving location designators the custom implementation of the Prolog predicate (`desig-costmap ?designator ?costmap`) should be provided to the system, because `merged-desig-costmap`, on which

`location-costmap-generator` is based, just merges all the costmaps that single implementations of `desig-costmap` provide. From software architecture point of view each implementation should be specific to a certain set of designator properties.

There are two implementations of the `desig-costmap` predicate in the `location-costmap` package: one is for the designator property (`to see`), which uses a Gaussian around the target pose to generate the costmap, and the other one is for reachability designators (e.g. those with the property (`to reach`)), which has a similar approach using Gaussian. There is a number of Lisp packages in CRAM that provide more complex implementations of these designator properties. They are based on OpenGL off-screen rendering techniques, inverse reachability calculations, etc [13]. If those packages are loaded into SBCL together with the `location-costmap` package, all the costmaps they provide for specific designator properties are merged together by multiplying the values with each other.

For instance, let us consider the following predicate:

```
(reachable ?world-state ?robot-base-position ?object)
```

which for bound `?world-state` and `?robot-base-position` should give all the objects in the world that the robot can reach from the specified position. A simple solution to this would be to load a package with an implementation that returns all the objects that are not further from `?robot-base-position` than the maximum distance that the robot can reach due to its mechanical design. However, if a higher accuracy is necessary a package with an implementation that uses inverse kinematics calculations can also be loaded. Then both implementations will be combined: first, the distance criterion will be checked for a certain object, and if it satisfied, the inverse kinematics method will be applied.

Another set of costmaps implemented in CRAM is for resolving spatial relations. They are discussed in the next part of the thesis.

Part IV.

Spatial Relations

10. Resolving Mechanism

In scope of this thesis CRAM was extended to enable describing locations based on spatial relations, which enables the use of relations as “left-of” or “far-from” in robot plan specifications. This can be utilized in spatial reasoning that is typically used in robotics, for example, for localization, in human-robot interaction to understand what a human means by these relations or for executing actions specified in natural language: for instance, it can be used for parsing the knowledge that can be found on the Internet about how to perform certain actions written as natural language instructions. Spatial relations expressions are used in such instructions quite often.

The approach was to introduce a number of additional designator properties with their corresponding inference mechanisms. Those properties are: `left-of`, `right-of`, `in-front-of`, `behind`, `near` and `far-from` and, of course, their combinations.

The resolution of spatial relation designator properties is based on the location costmap mechanism. For that the (`desig-costmap ?designator ?costmap`) Prolog predicate was redefined to generate costmaps for specific relations. For some of the costmaps it was necessary to develop new cost functions. The solutions of designators described with spatial relations depend on the context, so a small knowledge base was created to keep the context specific information. In the future this knowledge base is supposed to be merged with KnowRob. In order to validate the solution candidates sampled from the costmaps two validators have been developed: one for performing collision check for an object placed at the gen-

erated pose and the other one – for narrowing down the solutions to conform with the context.

As usual with the costmaps, the pose is generated using three different generators: a generator for a 2D location, height and orientation. The height generators of the costmaps for spatial relations resolution give solutions based on the supporting object of the reference object. For example, let us consider a designator

```
(designator location ((left-of mug-1) (for bowl-1)))
```

which represents a position for a *target object* `bowl-1` to the left of the *reference object* `mug-1`, as shown on Figure 10.1. The height generator will generate a `z` coordinate that corresponds to the position of the surface of the supporting object of `mug-1`, e.g. the table, on which `mug-1` stands.

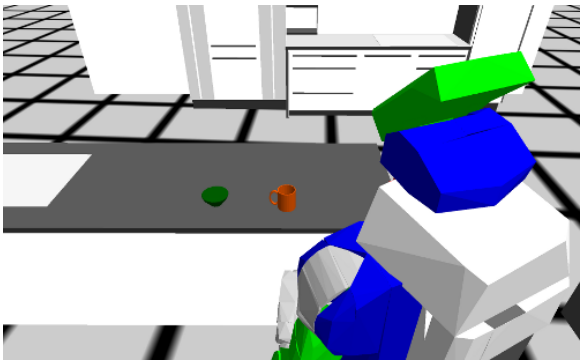


Figure 10.1.: A green bowl `bowl-1` – the target object – is positioned to the left of the orange mug `mug-1` – the reference object.

The orientation of the generated pose also depends on the supporting object of the reference object, and if the latter is not rotationally symmetric, then the corresponding orientation is generated aligned with the supporting object. So, it is important for the designator resolving mechanism to know, for which exactly object the pose is generated. That is why we need to specify the designator property `for`.

Please note, that at the moment the relations are only defined with respect to

poses of reference objects but not for any pose in space, which has no object associated with it.

10.1. Directional Relations

10.1.1. Costmap Generation

The directional relations `left-of`, `right-of`, `in-front-of` and `behind` are resolved based on a cost function, the implementation of which was inspired by the membership functions of fuzzy sets, as, for instance, presented in [8]. It gives higher values to the positions that strictly satisfy the specified spatial relation constraint, and lower values to those that deviate from the strict direction. For example, the costmap for a relation `left-of` is shown on Figure 10.2. The cost function takes an axis, a predicate and an optional threshold as parameters and returns a closure, which for each `(x, y)` gives a cost value. The axis is either `:x` or `:y` (in the supporting object coordinate system), depending on the relation. The predicate is either "bigger" or "smaller" and defines if the corresponding relation goes in the positive or negative direction of the axis. The axis and the predicate depend on the context and the point of view. For example, in the table setting context they are fully determined by the supporting object, more specifically – by the edge to which the reference object is the closest, as shown on Figure 10.3. However, if the context is a human-robot interaction and the human instructs the robot to bring him a certain object and describes it by spatial relations, the robot should know from whose point of view the relations are described – the human's or the robot's.

The edge is decided by the function `get-closest-edge`, which takes the pose of the object (in world coordinates) and the pose and dimensions of the support, calculates the distances from the object pose to the edges of the support and compares them. In context of table setting longer edges of the supporting object are more preferable, because the table is usually set along the longer edges. To ac-

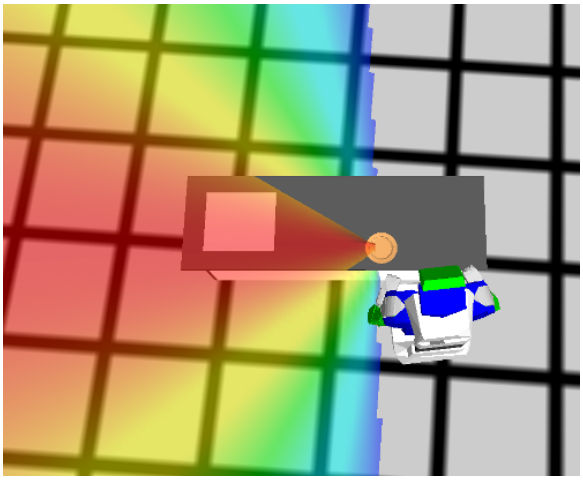


Figure 10.2.: Costmap for the relation (`left-of plate-1`)

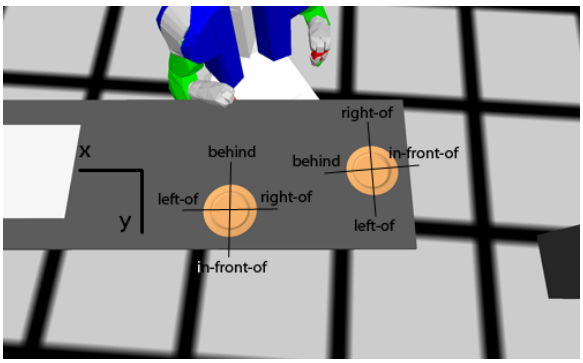


Figure 10.3.: Spatial relation directions for two different plate poses: one near the positive `y` axis edge of the table, and one – near its negative `x` axis.

count for that the distances are multiplied with corresponding coefficients.

The cost function works as following:

- a vector is calculated from the reference object to the target object
- the vector is projected onto the specified axis of the supporting object coordinate system
- the ratio between the lengths of the projection and the vector is calculated

- if (x, y) is positioned on the specified side of the axis the absolute value of the lengths ration is returned, otherwise the cost function returns 0.

As a result, we get, for example, the costmap shown on Figure 10.2 for the relation (left-of plate-1). This gives very high values for the coordinates, which are directly to the left of plate-1, and the more the coordinates deviate from that direction the lower the values get. This means that the coordinates that are only a bit to the left of the cup still have a small chance to be sampled. Depending on the context, this may or may not be allowed. In the latter case the threshold parameter of the cost function may be used to discard all the values, which are below it. However, this parameter can only be used if the corresponding location designator has only one spatial relation property. In the contexts, such as of table setting, where a combination of relations is necessary, this parameter should be used very carefully. For example, the resolution of a designator ((right-of plate) (behind plate)), which is the description of position of a glass in the table setting context, for 0 threshold would look as shown on Figure 10.4 (right). As mentioned before, the merging of costmaps is performed by multiplying the values in a certain grid cell from all the costmaps with each other and then normalizing the resulting costmap to have a sum of all values equal to 1. In case of the designator ((right-of plate) (behind plate)), two values are multiplied with each other for each grid cell – one value comes from the (right-of plate) costmap, and one – from (behind plate).

However, if the threshold is set to 0.9, the multiplication would give a 0 costmap, because there would be no overlap between the costmaps for properties right-of and behind during merging. A better alternative in that case is to discard the wrong solutions in the validation step.

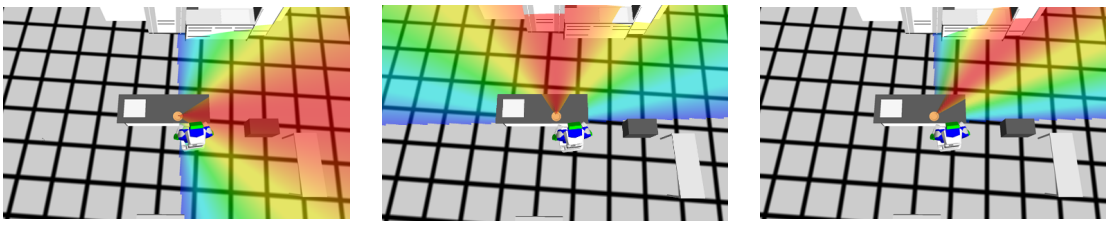


Figure 10.4.: Costmaps for the relations `right-of`, `behind` and their combination: (left) relation `right-of`, (center) relation `behind`, (right) combination.

10.1.2. Height and Orientation Generators

As already mentioned, the height of the solution pose is generated based on the support of the reference object. The corresponding height generator returns values equal to the `z` coordinate of the supporting object surface for any `(x, y)`, which is inside the boundaries of the support.

For the target objects that need to be correctly oriented, a Prolog predicate `orientation-matters` should be defined. For instance, in the table setting context those are the cutlery items. However, in a context of clearing the table, when the robot collects all the cutlery from the table on a tray, the orientation is not that crucial anymore. For rotationally symmetric objects the orientation, obviously, does not matter. The orientation of those objects, for which `orientation-matters` is defined, is calculated based on the supporting object. Depending on the context, there can be two ways to generate the orientation: either based on the reference object – then the orientation parallel to the edge of the support closest to the reference object will be taken, or based on the target object – then the closest edge will be calculated with respect to the target pose. The object is oriented in such a way, that the angle around the `z` axis would be 0 between the correct direction along the edge for the corresponding relation and the `y` axis of the object mesh, as shown on Figure 10.5.

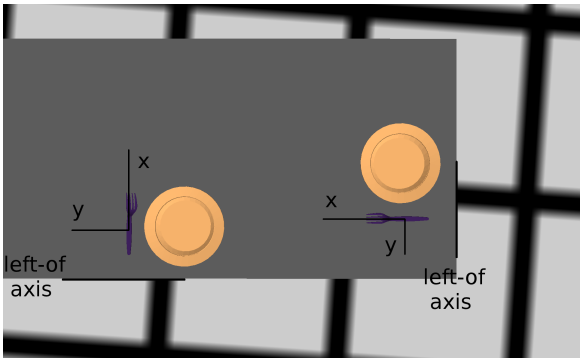


Figure 10.5.: The forks are positioned to the left of plates, as the angle between their `y` axes and the `left-of` axes of the table have 0 angle.

As a result, the pose shown on Figure 10.6 is generated upon resolving the location designator with properties (`(left-of plate-1) (for fork-1)`):

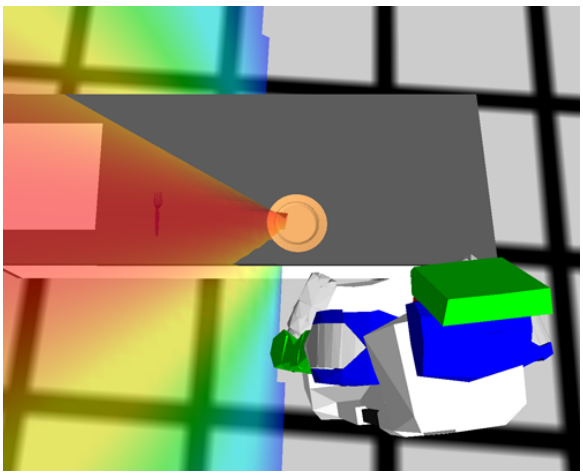


Figure 10.6.: A fork was assigned a pose generated from resolving the designator (`(left-of plate-1)(for fork-1)`).

10.1.3. Validation

Per default, the resolution of directional relations gives a costmap with spread angle of $180°$. In some contexts this angle is too wide, for example, when placing

a fork to the left of a plate the allowed angle is very small – around $20°$. A validation function was developed to reject all the solutions of a location designator below a certain threshold. The difference between this method and the threshold in the generator cost function is that the validation function looks at the values of the merged costmap, whereas the cost function can only process values for a single costmap. The corresponding threshold for each object type is defined in the knowledge base.

Collision avoidance is also performed by a validation function. It places the target object at the generated pose in the copy of the reasoning world and checks if the object is in collision with any other household object using the Bullet physics engine. It checks only the household objects because collisions with objects from the semantic map, such as furniture, are allowed, as the object will at least be in collision with its support.

10.2. Distance Relations

In the example above, a pose for a fork to the left of a plate was generated. As it can be seen, in the context of table setting the pose is incorrect: the fork should be placed very close to the plate. Additional designator properties – `near` and `far-from` – that describe distances can be used to fix that.

The costmap for the property `near` is generated by merging the following three costmaps:

- a costmap generated by a 2D Gaussian function with the mean positioned at the pose of the reference object and standard deviation taken from the knowledge base, as shown on Figure 10.7 a).
- an inverted range costmap, which has 1 assigned to all the points that are further than a certain minimal distance from the reference object pose, and 0 – elsewhere, as shown on Figure 10.7 b).

- a range costmap, which has ones at all the points that have less than a certain maximum distance from the object pose, as shown on Figure 10.7 c).

The resulting costmap is shown on the Figure 10.7 d).

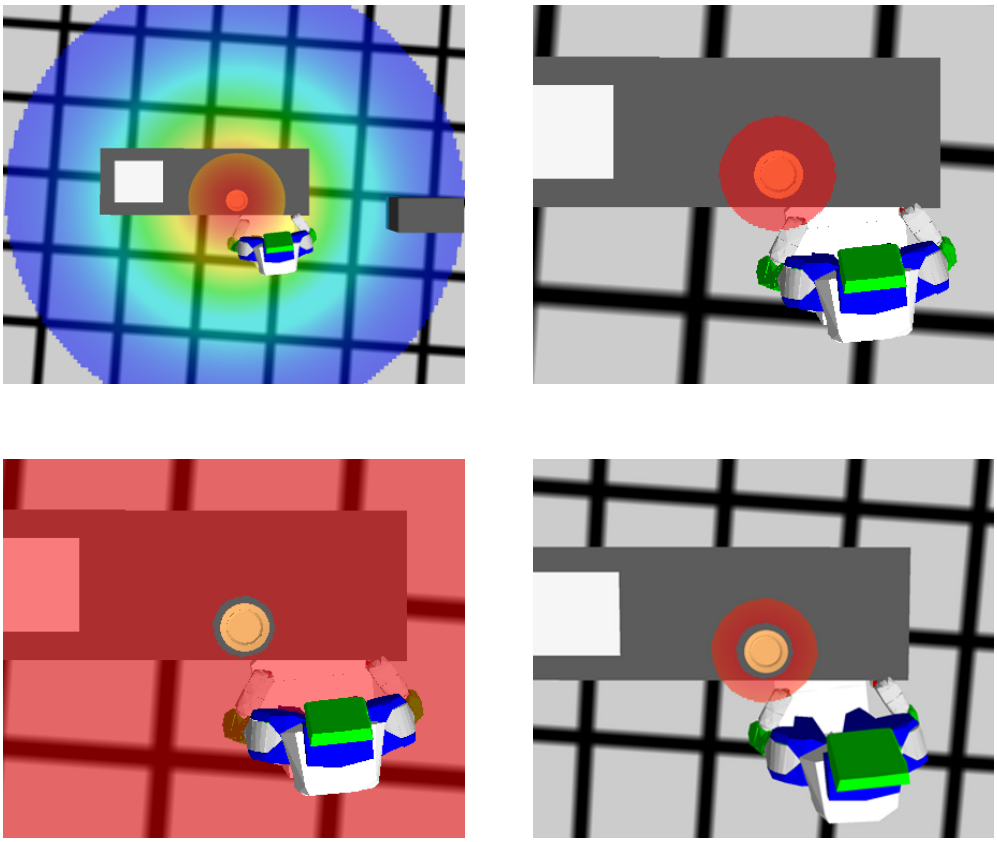


Figure 10.7.: Different costmaps defined for the `near` designator property and the overall costmap resulting from their combination: (top left) Gaussian cost function based costmap, (top right) range costmap for a certain maximal distance, (bottom left) inverted range costmap for a certain minimal distance, (bottom right) combination of the three

The minimal distance from the reference object pose depends on the size of the object and a certain padding, specified in the knowledge base for different objects and contexts. The maximum distance is defined based on the dimensions of the

reference and target objects and their corresponding paddings.

The relation `far-from` is resolved in a similar way, except that it has no gaussian cost function among its generators, as all the solutions in a certain range are equally good. In our system the relation `far-from` means that the reference and target objects should have a distance enough for another object with the size of the bigger of the two to be put in between, as shown on Figure 10.8.

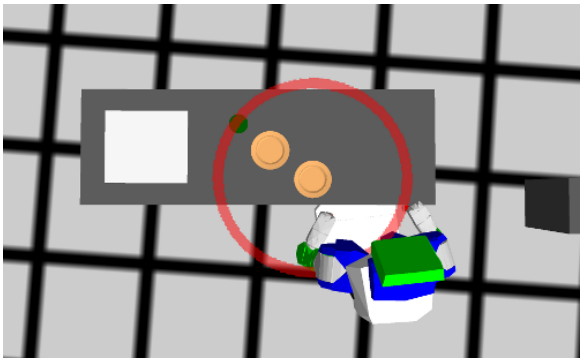


Figure 10.8.: Costmap for the `far-from` designator property.

10.3. Table Setting

The table setting is performed by first calculating the poses for the plates on the table, and then, putting the other objects, namely, the knives, forks and cups, relative to the plates. The cost function, which generates the poses for the plates, takes as input the number of objects, dimensions and the pose of the table, minimal and maximal distances that the objects can have between each other, the padding sizes that the objects should have from the table edges etc. and generates a costmap, which tries to distribute the objects on the table under the given minimal distance and padding constraints. If there is not enough space for all the objects it tries to fit as many as possible. The costmaps for three and six objects are shown on Figure 10.9.

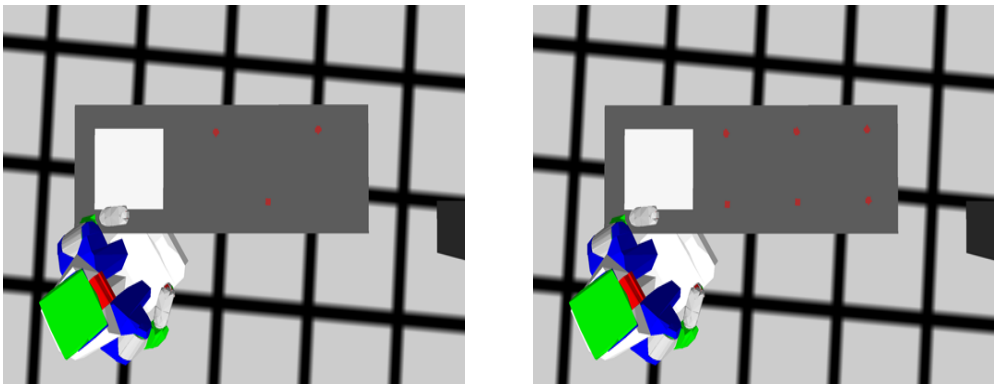


Figure 10.9.: Costmaps for plate locations on a counter top in a table setting scenario: (left) costmap for three objects, (right) costmap for six or more objects.

During the table setting the necessary objects are initially placed on the kitchen counter, as shown on Figure 10.10.

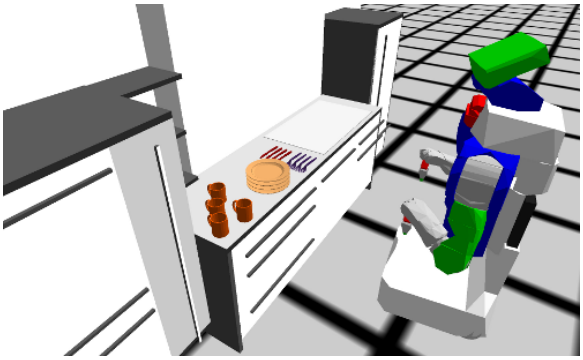


Figure 10.10.: Initial poses of objects to be used in table setting.

The execution of the scenario progresses as following: the robot searches for a plate on the kitchen counter and, as soon as one is found, it grasps it and puts it on the corresponding spot on the table, avoiding collisions (Figure 10.11 (top left)). An example designator for a location of a plate looks as following:

```
(designator location ((on counter-top) (name kitchen-island)
```

```
                    (context table-setting) (object-count 4)
                    (for plate-1)))
```

Then the robot performs the same actions, as for the plate, with a fork, a knife and a cup, using locations generated with the designators defined with respect to the found plate. The position for a cup is found by sampling from the costmap shown on Figure 10.11 (top center) and generated by the following designator:

```
(designator location ((right-of plate-1) (behind plate-1)
                      (near plate-1) (for cup-1)
                      (on counter-top)))
```

The designators for a fork and a knife generate costmaps shown on Figures 10.11 (top right), (bottom left) and are written as following:

```
(designator location ((left-of plate-2) (near plate-2)
                      (for fork-2) (on counter-top)))
(designator location ((right-of plate-3) (near plate-3)
                      (for knife-3) (on counter-top)))
```

The result of the table setting is shown on the Figure 10.11 (bottom right).

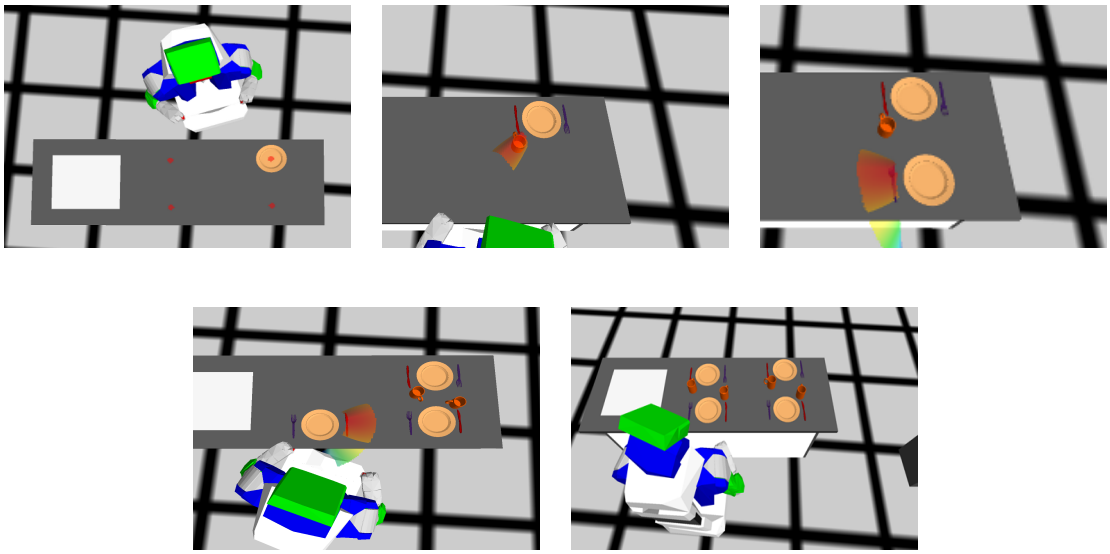


Figure 10.11.: Different stages of execution of a table setting scenario for four tableware sets: (top left) costmap for a plate position, (top center) costmap for a put-down location of a mug, (top right) costmap for a fork, (bottom left) costmap for a knife and (bottom right) final result.

11. Experimental Results

One run of execution of the whole scenario in projection mode with four table sets took 16:46 minutes[1] (depending on the random factors it may take slightly more or less time), from which 3:41 takes to generate the costmaps and sample valid poses for spatial relations. Sampling from a costmap took in different cases from 1 to 42 seconds. The reason for higher sampling times is that in table setting context the validity thresholds for costmap values are chosen very high in order to achieve accurate object placement. Therefore, it takes much time to find a valid value using the default random sampling. This type of sampling is efficient in some applications, such as for sampling poses on a table that do not bring to collisions, but is not the best method to use for spatial relations.

To improve that a different sampling mechanism was developed: the costmap is first sorted according to the values in its cells, then the sampling proceeds by returning coordinates of the cells with the highest values. The performance improved to 1 to 8 seconds for all the sampling during the scenario execution.

Using the new priority sampling mechanism makes some tasks simpler, such as putting down three forks and knives near a plate, as shown on Figure 11.1: all the positions (starting from the closest to the plate) are tried out in turn and rejected because of collision with other knives and forks, until a free spot is found. For random sampling this would not be possible, as the near relation should then be defined over a very narrow range of poses that are very close to the plate, because

[1] On a Dell Precision M4400 computer with a 2.53GHz processor, 3.9 GB RAM and a 512 MB NVidia graphics card.

any pose in the range, including the furthest ones, can be sampled, and therefore used as a put down location for the cutlery item. As a result, the cutlery can appear much further from the plate than is appropriate in the specific context. Whereas with priority sampling, further from the plate poses are sampled only if the closer ones are already taken.

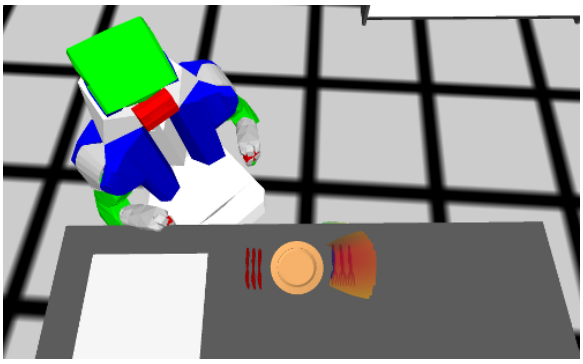


Figure 11.1.: Three cutlery sets are positioned near a plate using the priority sampling mechanism.

In certain contexts, some sampling mechanisms are better than others. Thus, a mechanism for selecting the sampling algorithm based on context needs to be implemented, which is considered future work.

The costmaps for spatial relations were tested in a setup where many objects were already placed on a table to prove their usefulness in real world settings. The initial setup is shown on Figure 11.2 (left).

In most cases where a human could theoretically achieve good results without moving the objects on the table, it was also possible with the system, as shown on Figure 11.2 (right). Please note, that this result would be much more complicated to achieve using heuristics or hard-coded locations, because they would have to use specific failure handling methods, e.g. if a collision is detected, try a new location 5 cm to the left. Whereas costmaps are very general and can be used for any kind of constraint based location description, as was demonstrated in this thesis: be it a

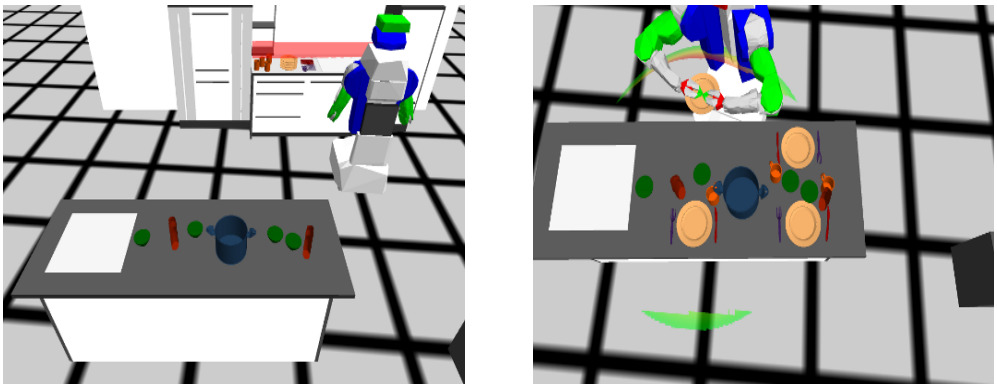


Figure 11.2.: Table setting in a cluttered environment: (left) a number of objects that do not belong to a dining set are positioned on a table, (right) the robot searches for put-down locations for the dining set items that do not cause collisions with other objects but still satisfy the context-specific constraints

reachability, visibility, stability, spatial relation or some other constraint.

Part V.

Conclusion and Future Work

12. Conclusion

In this Master's Thesis an approach was presented for resolving qualitative spatial scene descriptions into geometrical poses based on a generative model, which can be utilized in spatial reasoning, communication between the human and the robot or executing plans described in natural language.

The mechanism was integrated into the Cognitive Robot Abstract Machine (CRAM), which is a framework for robot action planning, reasoning and control. It is written in Common Lisp, uses the Robot Operating System (ROS) as a middleware and is open-source. Plans in CRAM are written in extended RPL called CRAM Plan Language (CPL) and are parametrized by designators, which are abstract symbolic descriptions of locations, objects, actions and other parameters that can be used for specifying plans.

Designators are resolved on demand when the plan needs to use a certain designator in the execution of a low-level action. Resolution works by translating the designator into a Prolog program, which uses both qualitative and quantitative reasoning in order to find a solution. The qualitative reasoning is done by first-order logic inference used in Prolog and quantitative reasoning is based on specialized methods, be it inverse kinematics solver inferring reachability or OpenGL offscreen renderer for visibility reasoning, etc. The quantitative reasoning is based on an accurate 3D representation of the environment of the robot.

The resolution mechanism for location designators is based on a generative model: initial solution candidates are first generated using lazy evaluation and are accepted only when they pass the validation step. For complex location designators

the concept of costmaps is used. Those are 2D grids defined over a certain domain, usually the area of the room, in which the robot is positioned, and have a value in range of 0 to 1 associated with each grid cell. A location designator can have a costmap associated with it, in which case the value in the grid cell determines, to which degree the location of the cell is appropriate as a solution for the given designator. These values are then used to sample solutions from the costmap during the generation step of designator resolution.

Spatial relations are represented in CRAM using location designators and are resolved using the costmap mechanism, which is a novel approach for resolving symbolic spatial relations. The relations defined at the moment are `left-of`, `right-of`, `in-front-of`, `behind`, `near`, `far-from` and their combinations. A number of cost functions was developed for generating the costmaps for the spatial relations. In addition, a new sampling mechanism was developed for the spatial relation costmaps, which gave substantial increase in efficiency of finding valid solutions. The validity of solutions is decided by two validation functions: one for checking if the generated pose for the object would cause it to collide with other objects, and the other – to check if the costmap value at the sampled position is good enough for a specific context.

A number of experiments was performed using the lightweight simulation framework of CRAM to check the applicability of the costmaps. All the experiments gave promising results:

- They were used to set a table using the objects, such as plates, cups, forks and knives.
- They were tested for putting more than one cutlery item on one side of a plate in the table setting context.
- The table setting scenario was run on a table full of other objects to see if the costmaps are flexible enough to still give valid solutions avoiding collisions and conforming with the context specific constraints.

13. Future Work

Despite the promising results demonstrated in this work, there is still much to be done to improve the approach and make it more robust and flexible. One of such improvements could be to implement a flexible mechanism for switching between the sampling algorithms based on the designator that is resolved, as currently the switching is done manually. In addition, a more flexible system could be implemented to work with different execution contexts and choosing correct designator resolution methods based on those contexts. Some of the context-based parameters are currently hard-coded for specific objects, so to improve that, machine learning could be used for finding correct context based parameters that cannot be inferred by reasoning.

The code should be further optimized and tested. It would be interesting to see the table setting scenario executed on a real robot. If more accuracy in the resolving mechanism will be necessary, it can be improved, for instance, by taking into consideration the bounding boxes of reference objects instead of just using their origins as reference points for the relations.

Ultimately, the system for translating spatial scene descriptions into geometric poses should be integrated with the natural language processing framework [18] to enable the robots to execute complex actions explained in detail in the Internet, also by means of spatial relations.

List of Figures

Bibliography

[1] Qualitative representation of positional information. *Artificial Intelligence*, 95(2):317 – 356, 1997.

[2] Marco Aiello, Ian Pratt-Hartmann, and Johan van Benthem, editors. *Handbook of Spatial Logics*. Springer, 2007.

[3] M. Beetz, U. Klank, I. Kresse, A. Maldonado, L. Mösenlechner, D. Pangercic, T. Rühr, and M. Tenorth. Robotic roommates making pancakes. In *Humanoid Robots (Humanoids), 2011 11th IEEE-RAS International Conference on*, pages 529 –536, oct. 2011.

[4] M. Beetz, L. Mösenlechner, and M. Tenorth. Cram – a cognitive robot abstract machine for everyday manipulation in human environments. In *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, pages 1012 –1017, oct. 2010.

[5] Isabelle Bloch. Spatial reasoning under imprecision using fuzzy set theory, formal logics and mathematical morphology. *Int. J. Approx. Reasoning*, 41(2):77–95, February 2006.

[6] Stéphane Cambon, Rachid Alami, and Fabien Gravot. A hybrid approach to intricate motion, manipulation and task planning. *The International Journal of Robotics Research*, 28(1):104–126, 2009.

[7] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, April 1936.

[8] Soumitra Dutta. Qualitative spatial reasoning: A semi-quantitative approach using fuzzy logic. In Alejandro Buchmann, Oliver Günther, Terence Smith, and Yuan-Fang Wang, editors, *Design and Implementation of Large Spatial Databases*, volume 409 of *Lecture Notes in Computer Science*, pages 345–364. Springer Berlin / Heidelberg, 1990.

[9] Erland Jungert. Symbolic spatial reasoning on object shapes for qualitative matching. In Andrew Frank and Irene Campari, editors, *Spatial Information Theory A Theoretical Basis for GIS*, volume 716 of *Lecture Notes in Computer Science*, pages 444–462. Springer Berlin / Heidelberg, 1993.

[10] L.P. Kaelbling and T. Lozano-Perez. Hierarchical task and motion planning in the now. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 1470 –1477, may 2011.

[11] Drew Mcdermott. A reactive plan language. Technical Report YALEU/DCS/RR-864, Yale University, 1991.

[12] L. Mösenlechner, N. Demmel, and M. Beetz. Becoming action-aware through reasoning about logged plan execution traces. In *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, pages 2231 –2236, oct. 2010.

[13] Lorenz Mösenlechner and Michael Beetz. Parameterizing Actions to have the Appropriate Effects. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, San Francisco, CA, USA, September 25–30 2011.

[14] E. Plaku and G.D. Hager. Sampling-based motion and symbolic action planning with geometric and differential constraints. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 5002 –5008, may 2010.

[15] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

[16] François-René Rideau and Robert Goldman. Evolving asdf: More cooperation, less coordination. In *Proceedings of the International Lisp Conference 2010*, 2010.

[17] S.J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*, chapter 10: Classical Planning. Prentice Hall Series in Artificial Intelligence. Prentice Hall, 2010.

[18] M. Tenorth, D. Nyga, and M. Beetz. Understanding and executing instructions for everyday manipulation tasks from the world wide web. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 1486 – 1491, may 2010.

[19] Moritz Tenorth and Michael Beetz. KnowRob – Knowledge Processing for Autonomous Personal Robots. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4261–4266, 2009.

[20] Ping Chuan Wang, Stephen Miller, Mario Fritz, Trevor Darrell, and Pieter Abbeel. Perception for the manipulation of socks. In *International Conference on Intelligent Robots and Systems (IROS)*, to appear 2011.