UNIVERSITY OF BREMEN

BACHELOR THESIS

PyCRAM – Python-based concurrent reactive programming language for autonomous mobile manipulation

PyCRAM – Python-basierte nebenläufige reaktive Programmiersprache für autonome mobile Manipulation

Authors:
Andy AUGSTEN
Dustin AUGSTEN

Supervisors:
Prof. Michael BEETZ PhD
Dr. Daniel GROSSE

Advisor:
Gayane KAZHOYAN

A thesis submitted in fulfillment of the requirements for the degree of Bachelor of Science

in the

Institute for Artificial Intelligence
Center for Computing and Communication Technologies (TZI)

February 11, 2019

Declaration of Authorship

I, Andy AUGSTEN, declare that this thesis titled, "PyCRAM – Python-based concurrent reactive programming language for autonomous mobile manipulation" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

Declaration of Authorship

UNIVERSITY OF BREMEN

Abstract

Faculty of Computer Science
Center for Computing and Communication Technologies (TZI)

Bachelor of Science

PyCRAM – Python-based concurrent reactive programming language for autonomous mobile manipulation

by
Andy AUGSTEN
Dustin AUGSTEN

Mobile manipulation robots that work in a domestic environment such as a houshold have made solving tasks like cleaning up possible. Researches have already given solutions for navigation and manipulation but the complexity of control programs to solve such tasks rises. Thus we need high level executives to be capable of doing a general and flexible planning and have to give it the possibility to abstract from the robot's hardware. In our thesis we describe PyCRAM which is a framework to accomplish exactly this using the Python 3 programming language.

Three main modules of PyCRAM are the Plan Language, Designators and Process Modules. The Plan Language is a domain specific language on top of Python which provides an easy error handling concept with the ability of re-execution. It supports the user in developing highly concurrent control programs to monitor different sensors in parallel without having to set up complex threading structures and without having to care about synchronization. To make control programs as general and flexible as possible, designators – as concept of symbolic plan parametrization – can be used to describe motions but also objects and locations. Process modules allow to abstract from the robot's hardware by separating the hardware communication from the action plan. This, for example, allows to create a plan to pick up an item and use it with both, a robot with only one arm and a robot with two arms, whereby the process modules implemented for each of the two robots decide how to execute it.

UNIVERSITY OF BREMEN

Zusammenfassung

Faculty of Computer Science
Center for Computing and Communication Technologies (TZI)

Bachelor of Science

PyCRAM – Python-based concurrent reactive programming language for autonomous mobile manipulation

by
Andy AUGSTEN
Dustin AUGSTEN

Mobile Roboter, die in einer häuslichen Umgebung wie einem Haushalt arbeiten, ermöglichen das Lösen von Aufgaben wie Aufräumen. Forschungen haben bereits Lösungen für Navigation und Manipulation bereitgestellt, aber die Komplexität von Kontrollprogrammen zur Lösung solcher Aufgaben steigt. Deshalb brauchen wir High-Level-Anwendungen die in der Lage sind eine allgemeine und flexible Planung durchzuführen, und müssen diesen die Möglichkeit geben von der Hardware eines Roboters zu abstrahieren. In unserer These stellen wir PyCRAM vor, ein Framework mit welchem genau dies in der Programmiersprache Python 3 möglich ist.

Drei Hauptmodule von PyCRAM sind die Plan Language, Designatoren und Prozessmodule. Die Plan Language ist eine domänenspezifische Sprache, die auf Python aufbaut und ein einfaches Fehlerbehandlungskonzept mit der Möglichkeit der erneuten Ausführung bietet. Sie unterstützt den Anwender bei der Entwicklung von stark nebenläufigen Kontrollprogrammen, die zeitgleich verschiedene Sensoren überwachen, ohne dass dieser sich dabei mit komplexen Thread-Strukturen oder Synchronisation auseinandersetzen muss. Um Kontrollprogramme so allgemein und flexibel wie möglich zu gestallten, können Designatoren – als Konzept der symbolischen Planparametrisierung – zur Beschreibung von Bewegungen sowie Objekten und Standorte genutzt werden. Prozessmodule ermöglichen die Abstraktion von der Hardware des Roboters, indem die Hardwarekommunikation vom Ausführungsplan getrennt wird. Auf diese Weise kann man beispielsweise einen Plan erstellen, um einen Gegenstand aufzunehmen, und ihn sowohl mit einem Roboter mit nur einem Arm als auch mit einem Roboter mit zwei Armen verwenden. Dabei entscheiden die Prozessmodule, die für jeden der beiden Roboter implementiert wurden, wie dieser ausgeführt wird.

Acknowledgements

We want to thank our advisor Gayane Kazhoyan for teaching us in how to use CRAM, for sharing her ideas with us, for all the discussions we had with her, for all the feedback she gave us and for motivating us all the time during our work on this thesis. Thanks to Gayane Kazhoyan and to Arthur Niedzwiecki for teaching us the prorgamming language Lisp.

We also want to thank our supervisors Prof. Michael Beetz PhD and Dr. Daniel Grosse for their great vision and motivation. And, of course, also Lorenz Mösenlechner for his amazing work on CRAM which our work is based on.

Thanks to our family as well as to all our friends and colleagues who always helped us during our studies and supported us. Especially we want to thank our parents Herbert & Marlies Augsten, our brothers Enrico & Daniel Augsten and our girl friends Jasmin Grotheer and Irem Yavaşoğlu, for their lovely motivation. Thanks to our friend Janik Brüggemann who took part in many courses with us during our studies. Thanks to Martin Manuszewski, Arthur Stricker, Albert Denhof, Eduard Stalbaum, Mustafa Hashimi and Zeeshan Iqbal for being the best friends of our childhood. Thanks to our great friends Ilyas Ryari, Andreas Paschigorow, Philip Płodzień, Furkan Doğtaş and Sabesan Sivananthan for all the time they have spent with us. Thanks to our friend Hendrik Engelhardt for all the projects we have worked on together and thus making us to better software engineers.

Contents

List of Figures

Chapter 1

Introduction

1.1 Problem Description

In the state of the art robotics there is a lack of high-level programming languages for autonomous robots, with which the programmer could easily represent event-guided behavior, failure handling and recovery mechanisms. An example of a task that would benefit from such a language could be setting a table for dinner. Although such a task seems simple for a human being, it is but the opposite for a robot and requires large constructs of code. There are many factors that need to be considered like is the location of the dishes and the table known to the robot or does it have to search for it, which dish or table to pick if there are several, what to do when there suddenly appears an obstacle, i.e. a pet walking in front of the robot, how to pick the items and how to place them to not create collisions and many more some of which need parallel execution or retry constructs. In order to not collide with an obstacle for example the robot needs to monitor its sensors while doing actions at the same time like moving. Or in case the robot fails to pick up an item as seen on Figure 1.1 (p. 2) because it was not found at the assumed location an retry construct is needed to retry the pick up action with adjusted values. Another important factor that must not be underestimated is the rapid development and production of software and hardware these days and the wide variety of different devices that go with it. While one user might be using robot *A* another user might want to use robot *B* which does not have two but only one arm and thus is a little cheaper. To make the software work on both robots nonetheless it would be convenient to be able to transfer the software to a different robot process module and make it work without any further changes. Using high-level programming languages for autonomous robots helps to simplify such tasks by using the provided mechanisms such as parallel execution, failure handling and retrying constructs or even support for different process modules.

CRAM Plan Language (CPL) [Mösenlechner, 2016] is such a language that provides powerful constructs for doing the above. However, CPL is implemented as an extension of the Lisp programming language and deeply relying on its macro mechanisms, which are more powerful and flexible than that of the other more mainstream programming languages, but Lisp is also a rather exotic language nowadays, hence it does not get the attention it could be getting if it was not for Lisp but a more widely spread programming language such as C++ or Python.

The research question that is tackled in the thesis is if it is possible to implement CPL with all its rich features in a more widely accepted programming language such as

Python which is a widely used general-purpose programming language and is currently often used for developing high-level executives by the robotics community.

To answer the research question, around the course of this thesis, a high-level robot control programming language, PyCRAM, that is based on CPL and provides a subset of it is to be developed as an extension of the Python language to make tasks for robots more robust, reliable and flexible and can be used naturally by a Python programmer and be easily and intuitively understood. PyCRAM includes a plan language for the easier implementation of concurrent reactive programs as well as a symbolic plan parametrization to specify entities such as motions, objects and locations by naming these or describing their properties in a more abstract way and process modules to execute the action entities regardless of the platform the program is running on.

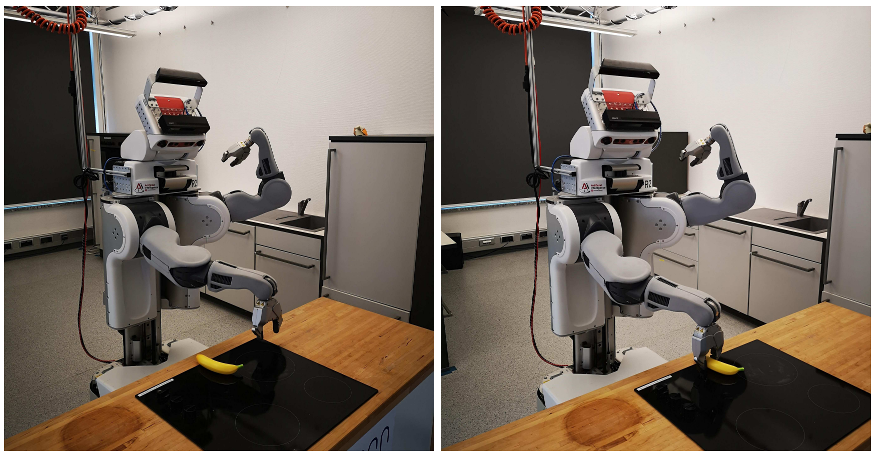


FIGURE 1.1: The PR2 robot retrying a pick up task because the assumed location of the item was wrong.

1.2 Related Work

Author: Dustin AUGSTEN

Among the robot programming community there already are quite a few domain-specific-programming languages or language extensions like the CRAM Plan Language [Mösenlechner, 2016] which this work is based on, or McDermott's RPL [McDermott, 1991] which is the direct ancestor or CRAM. But there are also other concepts, ideas or works which alongside the mentioned ones are presented in this chapter.

CRAM

CRAM is described in more detail in chapter 2.

McDermott's reactive plan language

Author: Andy AUGSTEN

Basically this work is for CRAM what CRAM is for PyCRAM: its model or ancestor. CRAM tries to reimplement this work even if in its own way. What sets the two works apart is that McDermott's RPL, unlike CRAM, is not really utilizing the full capacity of multicore processors but only simulates concurrency by shuffling the interpretation order of expressions randomly. Also RPL programs are evaluated by an interpreter just like PyCRAM while CRAM is compiled to machine code.

PYROBOTS **Author:** Andy AUGSTEN

PYROBOTS [Lemaignan, Hosseini, and Dillenbourg, 2015] is an embedded domain-specific language (eDSL) that has been developed similar to the work presented in this thesis to address the lack of high-level-programming languages for robots, which according to the authors comes from different reasons like the software architecture these languages enforce or because they are not practical as they come with an unfamiliar language or difficulties at the setup etc.

In general PYROBOTS is a lightweight python library providing useful tools. After defining a robot class as an instance of a GenericRobot class that needs to contain all the required low-level controllers and states of the robot, the programmer can start defining actions, declare arbitrary resources and create events that are monitored by the robot and can have callbacks attached to them. Actions are Python functions with an `@action` decorator that turn these into background actions. They are non-blocking by default but can be made blocking by locking predefined resources using a `@lock(RESOURCE)` decorator. In that case the function will block the given resource and any call of a function trying to lock that same resource will always wait for the resource to be available before executing. Also actions can be cancelled at any given time.

Unlike our work, however, PYROBOTS does not try to abstract away different robot specifications, it solely provides an easy to use toolset for basic needs. Also it uses a callback driven approach like most other languages do, while our approach uses the concept of fluents. Additionally, our approach is seemlessly integrated into Python through macros, whereas PYROBOTS is implemented as an extra layer on top of Python.

ROS commander (ROSCo) **Author:** Dustin AUGSTEN

ROSco [Nguyen et al., 2013] is, unlike the previous introduced works, not a domain-specific language but builds upon a state machine concept. The idea is to allow users the rapid creation and usage of behaviours, i.e. opening a drawer, in form of hierarchical finite state machines by using a graphical user interface. Created behaviours can then be used in any environment by just a few simple adjustments. That way a robot can be easily taught to turn the lights off at any place, merely the location of the light switch needs to be adjusted. Similar to our work is that ROSCo is offering an easy to use interface to create, or code in case of PyCRAM, programs, however, these programs are very limited in their logic and are just working off states to fulfill a task while programs coded using PyCRAM can be combined with any algorithms to create smart programs that do not need adjustments at all in new environments if done correctly.

Tell me Dave **Author:** Dustin AUGSTEN

Tell me Dave [Misra et al., 2016] is not another toolbox for writing robot software easier but an approach to have the robot act and fulfill tasks on natural language. This is similar to the designator concept of CRAM and PyCRAM as it uses natural language in form of symbolic description. What is so difficult is that the robot must ground the task described in natural language for the given environment. For example even for a simple command like "boil water" there are several possibilities to achieve the goal. The robot could use a an electric kettle or the stove and a pot to heat the water. The difference to PyCRAM is that [Misra et al., 2016] were trying to find an algorithm for Tell me Dave that will always make the right choice depending on the environment while PyCRAM is returning a solution out of possible infinite ones. That solution can be used or discarded by requesting the next one which is up to the programmer.

1.3 Contributions

The work presented in this thesis introduces a number of libraries and tools for the easier creation of high-level robot control programs that perform more robust, reliable and flexible in domestic environments.

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

1. The PyCRAM Plan Language, an extension of the Python programming language to support the developer implement high-level reactive and concurrent robot control programs. The Plan Language features parallelization and monitoring of competing processes, integration of sensor input and synchronization, error handling mechanisms and concurrent execution methods.
2. Designators, a way of symbolically describing plan parameters such as motions, locations or objects.
3. Process Modules, a possibility for abstracting away the robot's hardware to implement robot independent programs. A program can be implemented very generally and thus only the process modules for the different types of robots need to be implemented for the program to work on any other robot.

While these concepts are not completely new as they were already introduced in CRAM [Mösenlechner, 2016], they are completely reimplemented in Python which is more used and accepted in the robotic community than the Lisp language CRAM is relying on. As Lisp and Python are conceptually and implementation-wise very different programming languages, reimplementing CRAM's features was a challenge, because Python does on the one hand have limitations compared to Lisp but on the other hand also does have certain advantages.

PyCRAM is available at `https://gitlab.informatik.uni-bremen.de/pycram`.

1.4 Reader's Guide

Chapter 2

In this chapter all the foundations that are needed in order to understand this thesis are explained such as CRAM, which serves as model for PyCRAM, or MacroPy, which is especially important to implement most of PyCRAM's features.

Chapter 3

This chapter explains the actual implementation of PyCRAM in detail which consists of three parts: the domain-specific-programming language namely the PyCRAM Plan Language, designators which are used to symbolically describe parameters and process modules which communicate with the robot's hardware.

Chapter 4

In this part of the thesis an example application is presented and explained showing some results of the implementation of PyCRAM and how to use it. This chapter does also include a part on how to set up the working environment for PyCRAM as it is based on Python 3 while the default support for tools in the robotic community is for Python 2.

Chapter 5

Finally, in chapter 5, the results and findings we learned during the implementation of PyCRAM and its sample application are collected and summarized to discuss and answer our initial research question.

Chapter 2

Foundations

This chapter aims to explain all theoretical foundations that are needed in order to understand this thesis. There are three main subjects one should know about: Robotics, CRAM and Python. These main subjects are divided into several subsections. These subsections include further information about the concepts, interfaces and modules that were used to create the work presented in this thesis.

2.1 Robotics

Author: Dustin AUGSTEN

Robotics is the domain this work was created for in the first place which is why it is so important to understand what robotics is all about and why it has become so indispensable for us. Robotics is a branch of engineering and science and is omnipresent in today's society: whether in the production hall of a car manufacturer or, also often described as "bot" or "crawler", a computer program which is specifically collecting information and evaluating them. Robots are supposed to help human or ease and fasten their daily work, however, with no less efficiency or precision. In modern medicine, for example, robots can help a surgeon during a complicated surgery on the brain [Weinstein et al., 2007] or can give back a human without hands the ability to grasp things by using an artificial replacement [Light and Chappell, 2000]. Of course, these are just a few examples and there are many more uses for robots. But for robots to function and to know what they are supposed to do there are often operating systems and computer programs running on their hardware these days. These programs can become big really fast and thus confusing and complicated for the developers, hence, the work presented here deals only with a part of computer science in robotics and is demonstrating an approach to counter these problems by using PyCRAM, a software framework. While this work is not limited to the use of ROS and PR2, which are introduced below, in order to function, these were used to show and test this application.

2.1.1 ROS

Author: Andy AUGSTEN

ROS stands for "Robot Operating System" and is a framework which was created for writing independent robot software [Quigley et al., 2009]. As mentioned above software can become complicated and confusing, especially with all the tasks robots are supposed to fulfill these days. ROS was not only created to provide an robot independent platform but also to help developers create complex software with more ease as ROS provides a huge list of useful tools and libraries. This means for a developer, for example, that the interprocess communication will not have to be coded for the robot every time a different platform is used, or to be coded at all since there might already be a ROS package providing the desired function, so instead ROS can just be used to implement communication between the multitude of processes that run on a mobile manipulation robot. And since the work of the developer will be platform independent other developers will be able to build upon the previous made work to save time and effort. ROS supports easy sharing of work between different robotics groups through its package sharing tools so that an increase in the work flow is achieved to provide high quality products as every group might be an expert on another field like navigation or digital image processing. Moreover does ROS provide a modular design, which means that one can pick which parts of ROS are actually required for the project and which parts not, or simply should be implemented from scratch. ROS provides a few thousand packages with a large scale of application range and on top of that a huge user community with lots of experts [*ROS.org | Is ROS For Me?*]. Both can help build a professional software.

While there are many useful features to ROS, there a few core parts among them that were used in this work, namely: "Nodes", "Topics", "Messages", "actionlib" and "rospy".

Nodes

In ROS a node is a process performing computation. Nodes can communicate with each other, for example, by using topics. Usually a robot control system is comprised of many nodes each of which is responsible for a particular computational process of the robot. One node could be controlling the robot's wheel motors and yet another performs path planning or localization. The benefits of nodes are simple yet effective. Using nodes can help find software crashes easier, since each node is responsible for a certain part of the robot, a system failure or crash can be traced back to the appropriate node to find and solve the error. Also, nodes give the system a better overview and reduce code complexity.

Topics

Topics are buses used by nodes to exchange messages. A node can subscribe or publish to a topic to receive or send data. If a node needs certain information from another node it will subscribe to the relevant topic that the node providing the information is publishing to. A topic can have multiple subscribers and publishers.

Messages

Messages are transferred over topics by nodes for communication between these. They are data structures consisting of fields of primitive data types and arrays of these primitive types, such as integer, floating points, boolean etc, and can have arbitrarily nested structures and arrays. The messages mainly used by other ROS packages are included in the *common_msgs* package as are the *actionlib_msgs* for actions, *diagnostic_msgs* for diagnostics, *geometry_msgs* for geometric primitives, *nav_msgs* for robot navigation and *sensor_msgs* for communication with common sensors.

actionlib

The actionlib library provides an interface for preemptable tasks [*actionlib - ROS Wiki*]. For example, moving or performing a perception task. Once a message is sent to a topic the robot will react to that message, for example, by moving. In order to stop this action the user would have to send another message to the topic with appropriate data. While this can be complicated, the actionlib provides many useful tools to simplify these tasks. The user is able to stop the action of the robot with a single command or make the software wait for the current action to finish before continuing.

rospy

rospy is a client library for python programmers to interact with ROS topics, services and parameters [*rospy - ROS Wiki*]. It implements many useful tools to simplyfy and speed up implementation of one's own code.

2.1.2 PR2

Author: Dustin AUGSTEN

The PR2, or Personal Robot 2, which is shown in Figure 2.1 (p. 10), is a humanoid robot made by Willow Garage. It can navigate and manipulate in human environment and was designed for robot researchers [*Overview | Willow Garage*].

The idea is to one day have robots help us with our daily duty like doing the laundry, cooking or serving food. This does not have to be limited to personal use only but can be extended to restaurants, supermarkets etc. The goal of the ideal robot supporting us in every situation may still seem far away but for this purpose robots like the PR2 were invented because using the PR2 one can easily develop and test new robot applications and reproduce certain behaviors in our actual world.

In order to manipulate objects the PR2 uses its arms which consists of 3 parts: the actual arm, a wrist and a gripper. These can be seen in Figure 2.3 (p. 10). Each part can be controlled using ROS, for example, by sending messages to the corresponding topics. For example one can send a message containing data for the shoulder or upper arm joints to the pr2 arm topic in ROS in order to move the shoulder or the upper arm.

Moreover, quite a few sensors are included like several cameras, a laser scanner, pressure sensors, etc. Figure 2.2 (p. 10) shows the sensors. For this work, however, only the laser scanner was needed for navigation as the PR2 has mainly been used for testing and demonstration purpose only and is just one out of many possible robots PyCRAM can run on.

As seen in Figure 2.3 (p. 10), the PR2 does have a omni directional base with four wheels that can run at a speed of one meter per second in order to move around [*Hardware Specs | Willow Garage*].

The PR2 is supported by a community of 34 institution in 12 countries like several universities [*Join the Community | Willow Garage*], for example the "'University of Bremen", which benefits from its years of experience in robotics. Facilities can share their work and reproduce results of other PR2 users enabling authentic validation and steady progress as the wheel does not have to be reinvented every time.

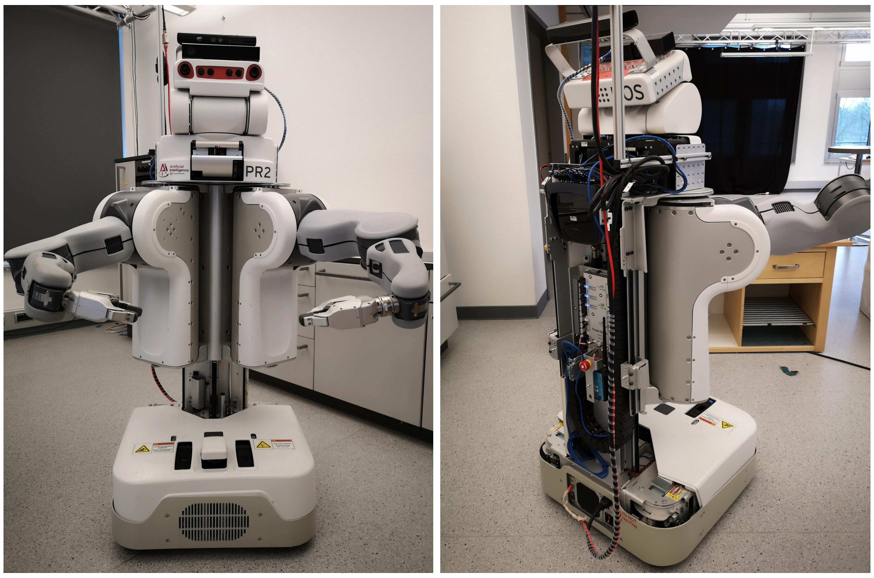


FIGURE 2.1: PR2 front and back view.

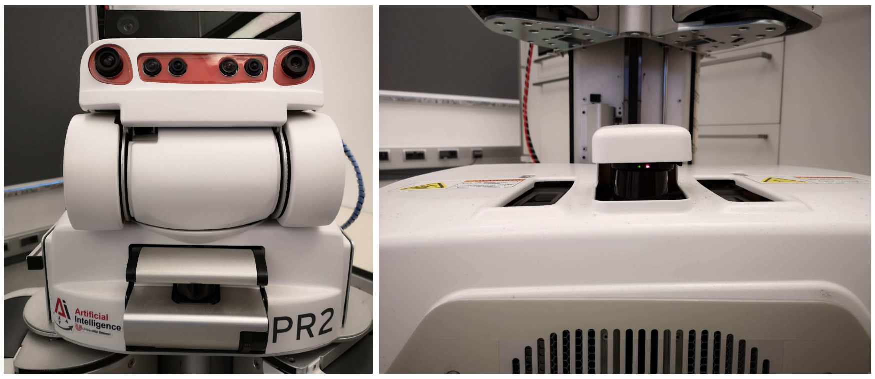


FIGURE 2.2: Some PR2 sensors.

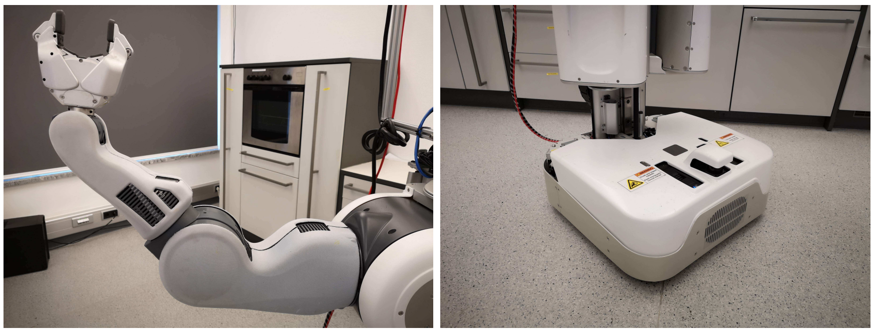


FIGURE 2.3: PR2 arm and the omni directional base.

2.2 CRAM

Author: Andy AUGSTEN

CRAM means "Cognitive Robot Abstract Machine" and is a framework created to help implementing cognitive high-level robot control programs [Mösenlechner, 2016]. CRAM comes with a lot of useful tools and libraries including but not limited to the plan language, symbolic plan parametrization and process modules. While there are some parts implemented in C/C++ the core packages of CRAM are implemented in Common Lisp because it comes along with a useful macro system and a functional programming paradigm, thus it is really flexible. As the work presented in this thesis, as the name PyCRAM already suggests, was modeled on CRAM, there are many similarities. PyCRAM implements the named features of CRAM in its own but similar way. This part of the chapter is further explaining the main ideas of the plan language, symbolic plan parametrization and process modules which were implemented in PyCRAM as well.

2.2.1 Plan Language

Author: Andy AUGSTEN

The plan language of CRAM implements many macros and functions on top of its base programming language Common Lisp to support the programmer create useful programs with multiple concurrent actions for robots. It is based on Drew McDermott's Reactive Plan Language (RPL) [McDermott, 1991] which unlike the CRAM Plan Language only simulates concurrency by shuffling the interpretation order of expressions randomly. Programs implemented using the CRAM Plan Language on the other hand make use of the operating system's native multithreading implementation thus enabling actual concurrency and allowing the use of all processors on multi-core CPUs [Mösenlechner, 2016, p. 23]. The concepts of the language that were also implemented in PyCRAM are fluents and the functions `seq`, `par`, `pursue`, `try-all`, `try-in-order` and `failure-handling`, which are described next.

Fluents

Fluents are proxy-objects that hold a value and provide a notification mechanism [Mösenlechner, 2016, pp. 35–45]. Most libraries use callbacks to handle and react to asynchronous input, which requires complex synchronization mechanisms. Fluents, however, are implemented in a different way, making them thread safe and easy to use in order to add reactivity to control programs running on multiple threads. In addition, fluents can be combined to fluent networks, which are fluents themselves, and allow complex expressions and conditions. Some useful functions that come along with fluents are `(wait-for fluent)`, `(whenever fluent)` and `(pulsed fluent)`, with `fluent` being the variable name of the fluent.

`(pulsed fluent)`: This will return a fluent containing the value `nil` or not `nil` whenever the value of the fluent has been changed or pulsed. It can be used in combination with other expressions to create useful conditions.

`(wait-for fluent)`: The thread executing this expression blocks if the value of the fluent is `nil` (in Lisp this is equivalent to `False` in Python) and waits for it to become not `nil` before continuing. This can also be used in combination with `(pulsed fluent)` like `(wait-for (pulsed fluent))` in order to block until the value of the fluent has changed or the fluent has been pulsed.

`(whenever fluent)`: The body is executed in a never ending loop whenever the value of the fluent is not `nil`. If the variable is `nil` or becomes `nil`, `whenever` blocks and waits for it to become not `nil` before continuing again. Like already mentioned, `whenever` is like a never-ending loop which means it will not stop unless it is canceled by a return statement inside its body. Just like `(wait-for fluent)` it can also be combined with `(pulsed fluent)` in order to make the body execute only whenever the fluents value has changed.

Fluent networks:

Fluent networks are fluents created by combining fluents. When a fluent that is part of a fluent network updates its value the fluent network itself will also update its value. Therefore the most important math operators were overloaded in CRAM and PyCRAM and in addition the operators `AND`, `OR` and `NOT` were implemented returning a fluent containing `nil` or `T` as value instead of the actual value. If that is not enough the programmer can create user defined fluent operators using `fl-funcall`.

For example `(> fl 20)` will create a fluent network containing the value `T` or `nil`. Assuming the value of `fl` is `10` the value of the network will be `nil`. Now whenever the the value of `fl` is updated, the value of the network will be changed as well, making it easier to work with constructs like `(wait-for fluent)` because one could simply use `(wait-for (> fl 20))`. `(wait-for (make-fluent :value ((value fl)> 20)))` for example would not work as the value of the newly created fluent would never change which is why it is so important to have fluent networks.

Fluent pulses:

In order to use `wait-for` and `whenever` in combination on value change, pulses are an important key feature to fluents. Pulse fluents are special fluents that hold the value `T` or `nil` and change only when the value of the fluent they were created from has changed or they were read from. They change their value on read depending on the selected missed pulse handling to be present `once`, `always` and `never`.

`once`: The pulse fluents value becomes true only once. As soon as its read from the value will become `nil`. The value will be true initially if the fluent the pulse fluent is created from has been pulsed before creation.

`always`: For each pulse the dependent fluent has been pulsed the pulse fluent can be read from without changing its value to `nil`.

`never`: Same as once only that the initial value will always be `nil`.

Sequential execution

`seq`: Basically this is identical to Common Lisp's `progn` and will execute the given forms sequentially. Succeeds if all child forms of `seq` succeed and fails otherwise.

`try-in-order`: This will execute all child forms sequentially and only fail if all have failed and then rethrow the errors or succeed when one succeeds.

Parallel execution

`par`: This will run all child forms in parallel. If one child form fails for example by throwing an exception `par` will also fail and rethrow the respective condition object. Accordingly `par` will only succeed if all its child forms have succeeded thus it will only terminate when all its child forms have finished.

`pursue`: This functions just like `par` but unlike `par` it will terminate in any case fail or succeed of any child. If one child fails `pursue` will fail and vice versa for the success of one child. All other child forms are evaporated as soon as `pursue` terminates. This can be used for example to monitor some state and finish once the goal state has been reached in one thread while the second thread performs an action to achieve this goal state.

```
(pursue
  (wait-for goal-reached-fluent)
  (loop do
    ...)) ; slowly approach goal
```

`try-all`: This will also execute all child forms in parallel and only fail if all have failed and then rethrow the errors or succeed when one succeeds.

Error handling

Just like try-catch constructs in other programming languages like Java, the expression `with-failure-handling` allows the programmer to execute failure handling code in case of exceptions with the addition of retrying the main block (try-block in Java).

2.2.2 Symbolic Plan Parametrization

Author: Dustin AUGSTEN

When writing robot control programs it would be most convenient when the programmer can easily specify entities such as motions, locations or even objects by naming these and their properties. CRAM implements designators which are Common Lisp objects describing parameters in a compact way using key-value pairs of symbols [Mösenlechner, 2016, pp. 74–85]. For example the programmer could have the robot grasp for a red cup on the table using the following set of designators:

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

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

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

A designator is not limited to a single solution but can in fact have multiple solutions. The `location-on-table` designator, for example, can have infinite solutions, as there theoretically are infinite different coordinates on a table, despite the table being limited by size. However, each additional key-value pair does add a constraint that limits the number of solutions. While `(type cup)` can reference to any cup in the world, `((type cup)(at location-on-table))` limits the number of cups to the cups that are available on the given `at` constraint.

As of now, CRAM's implementation does support three different designator classes: motion designators, object designators and location designators. Motion designators are meant as input for process modules (see chapter 2.2.3), object designators are describing objects on a symbolic level and location designators describe locations under given constraints. All three types derive from the same class, thus share some common properties, but differ in the way they are handled in the system.

2.2.3 Process Modules

Author: Dustin AUGSTEN

CRAM comes along with process modules. The idea of process modules is to abstract away from different robot specifications [Mösenlechner, 2016, pp. 85–95]. For example if a programmer had access to two different robots, one having one arm and the other one having two arms, (s)he would usually have to adjust the high level code for each robot for any given task, i.e. making popcorn. Process modules, however, provide a robot-independent interface by which the programmer does not need to worry about different controllers. It does not matter whether a robot has wheels or artificial legs in order to navigate through the world: as long as it can do the task in any way and there is a module providing the desired functionality, it will work out of the box using process modules. A module providing this functionality could, for example, accept a motion designator with the following description:

```
((type navigation) (goal p0))
```

where `p0` is a location designator.

The input for process modules must always be motion designators. In addition process modules provide a status fluent which can be queried to learn the current status of the module: running, free, waiting and whether the last action failed or succeeded.

Process modules can be executed synchronously or asynchronously. A synchronous execution will block the module for further executions until the current action has finished. This is especially useful when the robot cannot do multiple tasks at once for example moving to location `p0` and moving to location `p1`. These actions need to be performed sequentially. Since the synchronous execution blocks the module until it has finished the navigation requests can be called simultaneously nonetheless. While synchronous execution of process modules block until the current action has finished there are also asynchronous executions which will not block. This is important as there might be situations in which the robot can perform multiple tasks at once to increase performance or for other reasons but would be blocked otherwise, i.e. opening a cupboard with one arm and lifting a cup with another arm to place it into the cupboard. Without asynchronous process module execution this would take unnecessarily longer.

2.3 Python **Author:** Andy AUGSTEN

Python is a high-level programming language that promotes an easy-to-read programming style, such as structuring blocks by indenting rather than braces. Because of its clear syntax, Python is easy to learn. Various programming paradigms such as object-oriented but also functional programming are supported. In addition, Python offers dynamic typing, which is why Python is often used as a scripting language. Python is commonly interpreted which means that the program source code is not translated by a compiler into an executable file but is read in, analyzed and executed by an interpreter. The translation thus takes place at run-time of the program [Sanner, 1999].

Python is very popular because it is freely available for most major operating systems and is included in most standard Linux distributions. There is also an interface to integrate Python into web servers. There is a large selection of scientific libraries available such as NumPy for numeric calculations. Today, Python is one of the most popular programming languages [Frederickson, 2018].

2.3.1 MacroPy **Author:** Dustin AUGSTEN

In software development, a macro is a sequence of statements or declarations combined under a certain identifier (macro name), which can easily be executed by calling it. This allows the execution of frequent single statements in several places in the program. The difference to a function that can be passed parameters and possibly other functions to be executed, is that you can pass even larger sections of code, which are then inserted in the macro and executed as a whole [Barski, 2011, pp. 339–353].

Unlike Lisp, for example, Python has no official support for macros. The remedy is a library called MacroPy.

"MacroPy was initially created as a final project for the MIT class 6.945: Adventures in Advanced Symbolic Programming, taught by Gerald Jay Sussman and Pavel Panchekha. Inspiration was taken from project such as Scala Macros, Karnickel and Pyxl." [Haoyi, 2013]

To realize macros, MacroPy uses import hooks. As seen in Figure 2.4 (p. 16), the interpreter normally works by transforming the source code into an abstract syntax tree, then compiling it into bytecode at runtime and then executing it. MacroPy intercepts an import, also converts the source code into an abstract syntax tree but transforms it before compiling and proceeding with the loading of the module. Transform means to search for macros and replace them with their definitions. It is not possible to use macros in the command line or in a file that is executed directly, because the source code in this case does not run through the import hook. So one has to create another – so-called bootstrap file – which imports the source code as a module [*30,000ft Overview – MacroPy3 1.1.0 documentation*].

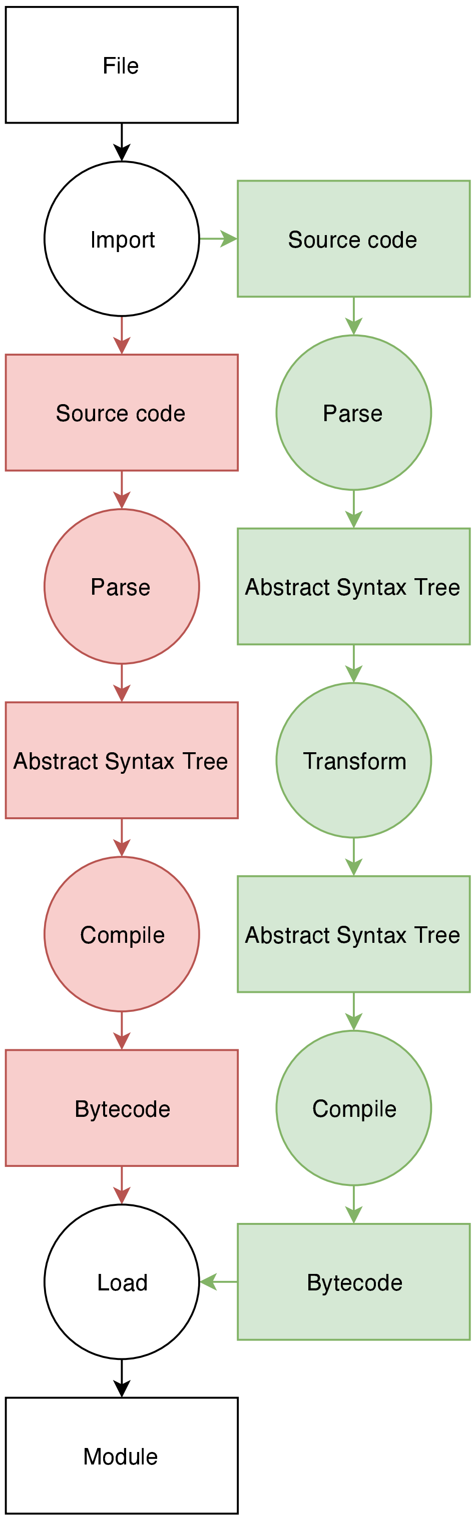


FIGURE 2.4: MacroPy intercepts an import to be able to transform the abstract syntax tree.

Chapter 3

Implementation

3.1 The PyCRAM Software Architecture

Author: Dustin AUGSTEN

Allowing robots to perform complex actions, such as, for example, cooking or tidying up in a dynamic environment, such as a human household, is a whole new challenge for robots compared to industrial applications. While industrial robots perform the same tasks over and over again in a static environment, robots in households must make decisions based on the configuration of the environment.

PyCRAM is a collection of Python modules that can be used to implement complex tasks such as cooking or tidying up. It includes a domain-specific programming language, the PyCRAM Plan Language, and supports so-called designators (symbolic description of plan parameterization). Plans are defined using the PyCRAM Plan Language. Plan parameters are defined using designators. This makes it possible to abstract from the actual hardware of the robot. Process modules are used to communicate with the hardware. These receive commands as designators, which they resolve to generate hardware-specific commands that are then sent to the different components of the robot as can be seen in Figure 3.1 (p. 18).

The programming language we used to implement such a system is Python because it is widely used, easy to learn, and one of the most popular programming languages. The MacroPy library makes Python very flexible in designing a domain-specific programming language.

In this chapter we discuss the implementation of the core components of PyCRAM, namely, the PyCRAM Plan Language, symbolic plan parameterization by designators and process modules.

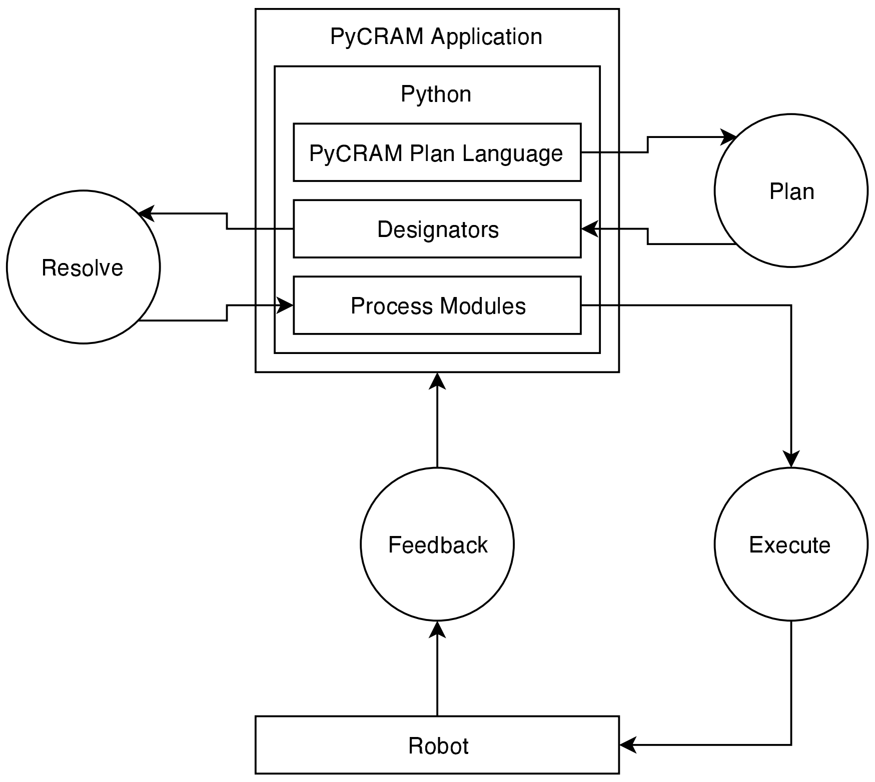


FIGURE 3.1: Plans are defined using the PyCRAM Plan Language. Their parameters are defined by designators, which are resolved by process modules. Process modules communicate with the robot's hardware to execute the plan. The robot gives feedback to the application.

3.2 The PyCRAM Plan Language

Author: Andy AUGSTEN

The PyCRAM Plan Language is based on Lorenz Mösenlechner's CRAM Plan Language [Mösenlechner, 2016]. While CRAM is implemented in Lisp, PyCRAM is implemented in Python 3. CRAM and PyCRAM both are languages for implementing reactive control programs for robots.

The Plan Language supports the developer in the implementation of plans, which are reactive and concurrent control programs for mobile robots. The Plan Language includes support for parallelization and monitoring of competing processes as well as integration of sensor input and synchronization.

3.2.1 Language Syntax

Author: Andy AUGSTEN

The PyCRAM Plan Language is implemented as a domain-specific language in Python. This makes the syntax similar to that of Python, and the full standard library and all the features that Python provides are also available in PyCRAM programs.

The reason we chose Python is its great popularity. The (slightly modified) MacroPy library[1] used for and integrated into PyCRAM offers a very strong macro system,

[1] https://github.com/daugsten93/macropy/. Accessed 01/14/2019.

which is very similar to that of Lisp and makes Python very flexible. The difference to macro systems like the one provided by C/C++ is that it does not build on a simple string substitution, so it can handle not only data but also whole code.

To understand the source code presented in this paper, basic knowledge in Python is required. The implementation of macros may not be easily readable even by experienced Python programmers who have not yet dealt with MacroPy. Let us briefly discuss MacroPy in the following paragraphs.

In MacroPy, a macro is defined as a function and annotated as one of three possible macro types. Here we distinguish between expressions, blocks and decorators. For our work, only block macros are relevant. When called, parameters can be passed to a macro as well as to a function, but above all entire code blocks can be passed, too. The macro can then arbitrarily transform such a code block, which is passed as a tree in the form of an AST (Abstract Syntax Tree) object, before execution. So the difference between a macro and a function is that you can control the execution of the code, which is why macros are so useful when you want to implement a domain-specific language.

For example, you could define a macro `loop` that executes a block of code `x` times, passing `x` as a parameter. The following snippet, stored in the file *print_example.py*, would then execute a `print` statement `5` times.

```
from loop_macro_module import macros, loop

with loop(5):
  print('Hello,␣world!')
```

In the above snippet, `loop_macro_module` is the module that provides the macro `loop`. In the first line we import the `macros` object of the respective module and then the `loop` macro itself. Then we use the macro to execute the code block it returns. That code block is represented in the next snippet:

```
for _ in range(5):
  print('Hello,␣world!')
```

Macros cannot be used in the same module in which they were defined. Instead, the file that uses the macro must be imported as a module in a so-called bootstrap file, which then has to be executed.

```
import macropy.activate
import print_example
```

To understand how the macro is implemented, we first have to learn about the two macros `hq` and `ast_literal` from the MacroPy library. The macro `hq` generates a tree from Python code. Alternatively, it is also possible to build a tree manually, but this requires good understanding of the Abstract Syntax Tree, which Python code is based on. The macro `ast_literal`, in turn, generates Python code from a tree. We need this macro inside the `hq` macro because we pass Python code to the `hq` macro, but the code block and the parameters passed to our macro are passed as a tree.

Now this is our macro implementation, stored in a file named *loop_macro_module.py*:

```
from macropy.core.macros import Macros
from macropy.core.hquotes import macros, hq
from macropy.core.quotes import macros, ast_literal

macros = Macros()

@macros.block
def loop(tree, args, **kw):
  with hq as new_tree:
    for _ in range(ast_literal[args[0]]):
      ast_literal[tree]

  return new_tree
```

The line `macros = Macros()` marks the module as one that provides macros. We use the `block` method of the `macros` object to declare our defined function `loop` as a block macro. The `tree` parameter is the code block passed to our macro as a tree. The parameter `args` contains all passed parameters as a list, including our `x`, which is the number of tries to execute the loop. The macro `hq` places the generated tree in the `new_tree` variable we specify. Our macro then returns this tree.

The syntax of the two macros `hq` and `ast_literal` differs because `hq` is a block macro like our `loop` macro is one, too, and `ast_literal` is an expression macro: `hq` uses the "with" syntax, whereas `ast_literal` expects the tree to be passed in square brackets.

3.2.2 Fluents

Author: Andy AUGSTEN

Fluents are synchronized proxy objects, which are used as variables with changing values. They allow threads to watch them and wait for specific changes as well as to change their value. Fluents are very powerful when it comes to implementing reactivity. The base idea of fluents goes back to the fluent calculus [Thielscher, 1998].

Fluents in PyCRAM

To create a fluent, its constructor must be called. This can be given a value and a name as a parameter. If no value is passed, this corresponds to the value `None`. If no name is passed, a random string is generated.

```
f = Fluent(1)
```

This binds the variable f to a newly created fluent with the value 1. The value can be read by the function get_value and changed by the function set_value.

```
f.get_value() # returns 1
f.set_value(2) # sets the value to 2
f.get_value() # returns 2
```

In PyCRAM, control programs communicate with a robot through a middleware which performs an asynchronous callback for each new sensor input. In this callback, the value of a fluent is changed and the control program is executed in parallel. For example, our program can receive any change by moving the robot as a callback and update the fluent containing the robot's location.

By calling the function wait_for, the program is able to block the execution and wait for the value of a fluent to become not None. For example, one can wait for a gool_reached fluent to become true.

```
goal_reached.wait_for() # blocks if the value is None, no
   effect otherwise
```

We can combine Fluents to create a so-called fluent network. A fluent network is itself a fluent underlying a function that determines its value. The special feature is that the value is always re-evaluated and can therefore change if the underlying function, for example, takes into account the value of another fluent. A trivial example is the following code, which waits for the value of a fluent divided by 3 to become less than 1.

```
(f / 3 < 1).wait_for()
```

For this purpose, the comparison functions <, <=, ==, !=, > and >= as well as the arithmetic functions +, -, * and / are overloaded so that they return a fluent network if one of the two parameters is a fluent. In addition, the functions IS and IS_NOT are defined for the comparison functions is and is not, which cannot be overloaded. The same applies to the logical operators and, or and not, here the functions AND, OR and NOT have been defined accordingly.

The value of the generated fluent network is None as long as the condition is not met, otherwise True. The value of an unfulfilled condition is therefore None, not False, because wait_for assumes this (block the current thread as long as the value is None).

Fluent networks can also be created using programmer's own functions by passing these instead of a value when generating the fluent. So, instead of fluent_net = f / 3, the following would also be possible:

```
fluent_net = Fluent(lambda: f.get_value() / 3)
```

In this case, however, the fluent network cannot know if the value of another fluent was used and hence the fluent network should be added to it as a child. This causes the condition underlying the invocation of the `wait_for` function on the network to be notified when the value of the parent fluent, and thus the value of the network, changes.

```
f.add_child(fluent_net)
(fluent_net < 1).wait_for()
```

To always execute a code block when the value of a fluent changes and is not `None`, the `whenever` macro can be used. It is to be understood as a loop executing `wait_for` on a fluent followed by the code block in each pass. The following example illustrates how to call the `move` function for each new location stored in the fluent `loc`.

```
loc = Fluent()

with whenever(loc):
  move(loc.get_value())
```

For this purpose there can be a function `update_location`, which is a callback that is called every time a new goal pose is entered by a robot user.

```
def update_location(location):
  loc.set_value(location)
```

Sometimes you want to wait for a value to update, whether it is a `None` or not. To do this, call the `pulsed` function, which returns a fluent, the value of which changes to `True` as soon as the value of the parent fluent changes.

```
with whenever(loc.pulsed()):
  move(loc.get_value())
```

A fluent generated by calling the `pulsed` function also changes its value to `True` when the `pulse` function is called on the parent fluent. This can be useful for releasing the `wait_for` block without the value changing.

One problem that can occur when using `whenever`, is that the value of a fluent may change again while the code block is still executing. To counter this, the function `pulsed` can be given a parameter of the type `Behavior` (enumeration from the `fluent` module) to indicate how `whenever` should handle missed pulses. Possible values are:

1. Behavior.NEVER: Missed impulses are ignored.
2. Behavior.ONCE: The code block is executed one more time, no matter how many missed impulses have occurred.
3. Behavior.ALWAYS: The code block is executed again for each missed pulse.

```
with whenever(loc.pulsed(Behavior.ONCE)):
  # execute once more if missed pulses occur
  move(loc.get_value())
```

Implementation

Fluents are implemented as a class. When the constructor is called, a Condition object [Hoare, 1974] and a Lock object [Dijkstra, 2001] are created and stored in the variables _cv (Condition Variable) and _mutex. The Condition object is used to inform the function wait_for if the function pulse was called. The Lock object is for synchronization. The variables _pulses, _children and _handle_missed are assigned default values. _pulses is there for the macro whenver to know how often the pulse function was called while the code block was still executing. _children serves the fluent networks, here a parent fluent deposits his children in order to be able to call the function pulse on them if it was also called on itself. _handle_missed is again for the macro whenever to know how to handle missed calls. The parameters value and name passed to the constructor are stored in the variables _value and name. If the parameter name is not given or if it corresponds to None, a random string is generated.

Variable names beginning with "_" indicate variables that are not directly accessible. This has become a convention in Python because there are no access modifiers, such as there are in languages like Java.

```
class Fluent:
  def __init__(self, value = None, name = None):
    self._cv = Condition()
    self._mutex = Lock()
    self._pulses = 0
    self._children = []
    self._handle_missed = Behavior.NEVER
    self._value = value

    if name is not None:
      self.name = name
    else:
      self.name = str(uuid4())
```

There are getter and setter functions for the value. The reason for this is firstly that we can use the Lock object to grant thread security, and second that if the value is a function we can call it and return its return value.

```
def get_value(self):
  with self._mutex:
    if callable(self._value):
      return self._value()

    return self._value

def set_value(self, value):
  with self._mutex:
    self._value = value

  self.pulse()
```

The function `pulse` increases the pulse counter for each child object, which are generated inter alia by calling the function `pulsed`, and then calls the same function on it. Then the condition object is informed of a possible change. There may be a change in the children generated by the `pulsed` function, which change their value to `True` if the impulse counter is not `0`, and after a call to the `set_value` function, which in turn calls `pulse`.

```
def pulse(self):
  for child in self._children:
    with child._mutex:
      child._pulses += 1

    child.pulse()

  with self._cv:
    self._cv.notify()
```

`pulsed` creates a new `Fluent` object and sets a function that returns `True` if the pulse counter is not 0 and otherwise `None`, as its value. How the macro should handle missed impulses is passed to the function as a parameter. The default is `2`, which is the same as `Behavior.ONCE`. The value of the parameter is stored in the instance variable `_handle_missed`. Then the new fluent is added to the current fluent as a child so that an impulse on the parent fluent can also affect the child fluent. The `Fluent` object created in this way is then returned.

```
def pulsed(self, handle_missed = 2):
  fluent = Fluent()

  def value():
    if fluent._pulses != 0:
      return True
    else:
      return None

  fluent.set_value(value)
  fluent._handle_missed = handle_missed
  self.add_child(fluent)
  return fluent
```

For the different comparison operators that are overridden by the `Fluent` class, a helper function `_compare` has been defined. This expects the operator to be used and the object with which the `Fluent` object is to be compared to be passed as parameters. A new `Fluent` object is created. The function then checks whether the second object is also a fluent, because in this case `get_value` must be called on both objects in order to work with their values. In any case, a function that returns `True` or `None` is defined and set as the value of the new fluent. In addition, the new object is added to the parent fluent or the two parent fluents from which it originated as child fluent. It will then be returned. The following is an implementation of the operators < (`__lt__`) and <= (`__leq__`) is shown.

```
def _compare(self, operator, other):
  fluent = Fluent()

  if type(other) == Fluent:
    def value():
      if operator(self.get_value(), other.get_value()):
        return True
      else:
        return None

    other.add_child(fluent)
  else:
    def value():
      if operator(self.get_value(), other):
        return True
      else:
        return None

  self.add_child(fluent)
  fluent.set_value(value)
  return fluent

def __lt__(self, other):
  return self._compare(operator.lt, other)

def __leq__(self, other):
  return self._compare(operator.leq, other)
```

The other comparison operators >, >=, ==, `!=`, `is` and `is not` are defined in a similar fashion. `is` and `is not`, however, cannot be overloaded and hence are represented by the functions `IS` and `IS_NOT`.

```
def IS(self, other):
  return self._compare(operator.is_, other)

def IS_NOT(self, other):
  return self._compare(operator.is_not, other)
```

It is identical for the functions `AND` and `OR`, which represent the logical operators `and` and `or`, but they have to do without a helper function because these two operators do not exist as functions. The implementation of `AND` is shown below.

```
def AND(self, other):
  fluent = Fluent()

  if type(other) == Fluent:
    def value():
      if self.get_value() and other.get_value():
        return True
      else:
        return None

    other.add_child(fluent)
  else:
    def value():
      if self.get_value() and other:
        return True
      else:
        return None

  self.add_child(fluent)
  fluent.set_value(value)
  return fluent
```

The `NOT` function is also very similar, but there is no second operand.

```
def NOT(self):
  def value():
    if not self.get_value():
      return True
    else:
      return None

  fluent = Fluent(value)
  self.add_child(fluent)
  return fluent
```

The mathematical operators always have to overwrite two functions. The one function is called when the first operand is a fluent. The second operand can also be, but does not necessarily have to be a fluent. The other function is called if the first operand is not a fluent, but the second one is. Since very often the same thing is done here, there is again a helper function, `_math`, for this purpose.

```
def _math(self, operator, operand, other):
  fluent = Fluent()

  if type(operand) == Fluent:
    if type(other) == Fluent:
      value = lambda: operator(operand.get_value(), other
         .get_value())
      other.add_child(fluent)
    else:
      value = lambda: operator(operand.get_value(), other
         )

    operand.add_child(fluent)
  else:
    value = lambda: operator(operand, other.get_value())
    other.add_child(fluent)

  fluent.set_value(value)
  return fluent

def __add__(self, other):
  return self._math(operator.add, self, other)

def __radd__(self, other):
  return self._math(operator.add, other, self)

def __sub__(self, other):
  return self._math(operator.sub, self, other)

def __rsub__(self, other):
  return self._math(operator.sub, other, self)

...
```

The function `wait_for` calls the function of the same name on the condition object `_cv` and returns its return value. A lambda function is passed as predicate, which returns `True` if the call of `get_value` does not return `None`. Here we are based on the Lisp implementation, in which the value does not have to be `None` (or `nil` is Lisp), because in Lisp this corresponds to the value `False`. Optionally, the function can be given a parameter `timeout`.

```
def wait_for(self, timeout = None):
  with self._cv:
    return self._cv.wait_for(lambda: self.get_value() is
       not None, timeout)
```

The `whenever` macro uses the function `_block` from the `helper` module, so to understand `whenever` let us first look at `_block`. The function `_block` receives as parameter a tree and puts it into a new tree, in which the transferred tree is surrounded by a condition. This way several statements are put in a block because otherwise they can not always be nested (for example, within the macro `par`, which executes each statement in its own thread).

```
def _block(tree):
  with q as new_tree:
    # Wrapping the tree into an if block which itself is a
       statement that contains one or more statements.
    # The condition is just True and therefor makes sure
       that the wrapped statements get executed.
    if True:
      ast_literal[tree]

  return new_tree
```

`Behavior` is a class that inherits from the `Enum` class and defines the three values `NEVER`, `ONCE` and `ALWAYS`. These represent behaviors on how to deal with missed pulses inside the `whenever` macro.

```
class Behavior(Enum):
  NEVER = 1
  ONCE = 2
  ALWAYS = 3
```

`whenever` runs a `while` loop, within which the `wait_for` function is called on the fluent passed as a parameter. Then the code block passed to the macro is executed. The following code describes the logic for how to deal with missed impulses. If the behavior for this is set to `Behavior.NEVER`, the pulse counter is set to `0` so that the value of a fluent generated by calling the pulsed function is `None` again. Otherwise the pulse counter is decremented, so the value remains `True` if this was greater than `1` and the code block is executed again. If the counter is greater than `1` and the behavior is set to `Behavior.ONCE`, the counter is set to `1` so that the code block is executed only once more.

```
@macros.block
def whenever(tree, args, **kw):
  with hq as new_tree:
    _fluent = ast_literal[args[0]]

    while True:
      _fluent.wait_for()
      ast_literal[tree]

      if _fluent._handle_missed == Behavior.NEVER:
        with _fluent._mutex:
          _fluent._pulses = 0 # Ignore missed pulses
      else:
        with _fluent._mutex:
          _fluent._pulses -= 1

          if _fluent._pulses > 1 and _fluent._handle_missed
              == Behavior.ONCE:
            _fluent._pulses = 1 # Execute body only once
               more

  return _block(new_tree)
```

3.2.3 PyCRAM Plan Language Expressions

Author: Dustin AUGSTEN

The domain-specific language PyCRAM Plan Language, in addition to fluents, also extends Python to include sequential and parallel execution macros with error handling. This chapter discusses the different possibilities that PyCRAM offers for this and their properties as well as the underlying implementation.

Sequential execution

Python code is actually always executed sequentially. What differentiates sequential execution with PyCRAM is that it automatically catches errors and makes them easier to handle.

PyCRAM language statements for sequential execution are `seq` and `try_in_order`.

`seq`: All statements passed to the macro `seq` are executed one after the other. Like all PyCRAM Plan Language expressions, this macro returns `State.SUCCEEDED` or `State.FAILED` as fluent. It succeeds if all statements could be executed without error, otherwise it is considered as failed. If one instruction fails, the others will not run. The current thread itself is not interrupted if an error occurs, it stays alive and errors are stored as a list in a variable which can be given as optional parameter.

The following example first executes statement `s1` and then statement `s2`. If an error occurs at the first instruction, the second one is no longer executed. The thread would not be interrupted and the error would be stored in the list `errors`. The fluent `s` would have the value `State.FAILED` or `State.SUCCEEDED` if both statements can be executed without error.

```
from pycram.language import macros
from pycram.language import seq

with seq(errors) as s:
  s1
  s2
```

`try_in_order`: Again, this macro executes all given statements sequentially, but it succeeds if only a single statement succeeded and fails only if all statements have failed. This also means that as soon as one statement succeeded, the others will not be executed.

In the following example, statement `s1` is executed first and statement `s2` is executed only if statement `s1` has failed. If `s2` also fails, then the fluent `s` has the value `State.FAILED`, otherwise `State.SUCCEEDED`.

```
from pycram.language import try_in_order

with try_in_order(errors) as s:
  s1
  s2
```

Parallel execution

The parallel execution expressions offered by PyCRAM are based on the `threading` module. The expressions make it possible to execute several statements in separate threads without having to create, start and collect a thread for each statement.

Unlike the Lisp implementation, the other threads do not terminate when one has completed successfully or failed because threads cannot be interrupted in Python. This is also not desirable because a thread could currently be in a critical section where a lock object is not resolved when the thread is interrupted, or because the thread cannot clean up behind it to possibly recover occupied memory again. To counteract this problem, it is possible to access the current status within the macro. So each thread can know if the macro has done its job and then schedule properly.

Language statements for parallel execution are `par`, `try_all` and `pursue`.
`par`: The macro `par` executes each submitted statement in a separate thread and completes successfully if all statements could be executed without error, otherwise it will fail.

The next example shows how the instructions `s1` and `s2` are executed at the same time. If one of the two statements results in an error, the fluent `s` has the value `State.FAILED`, otherwise `State.SUCCEEDED`.

```
from pycram.language import par

with par(errors) as s:
  s1
  s2
```

If one wishes to execute the instructions `s1` and `s2` in one thread and only the instruction `s3` in another thread, one must package the two individual instructions `s1` and `s2` into a block, which in turn is a single instruction. This is accomplished by nesting the two statements in a condition.

```
with par(errors) as s:
  if True:
    s1
    s2
  s3
```

Alternatively, one could also use the macro `seq` for this purpose. However, as `seq` throws no errors, these are not passed to `par` and thus not stored in `errors` in this case.

```
with par(errors) as s:
  with seq as state:
    s1
    s2
  s3
```

If a thread goes through a loop and should interrupt it as soon as another thread has failed, that is, if the value of the value has been set to `State.FAILED`, then this can easily be queried on every run.

```
with par(errors) as s:
  s1
  while True:
    if s.get_value() == State.FAILED:
      break

    s2
```

Since the status can only be `State.SUCCEEDED` when all threads are terminated, within the macro the fluent can only have the value `None` or `State.FAILED`. Hence in this case it is also sufficient to check whether the value is not `None`.

```
with par(errors) as s:
  s1
  while True:
    if s.get_value():
      break

    s2
```

`try_all`: This macro succeeds if any of the statements do and fails if all have failed.

In the following example, the statements `s1` and `s2` are executed in parallel. If one of the two statements is successful, the fluent `s` has the value `State.SUCCEEDED`, otherwise `State.FAILED`.

```
from pycram.language import try_all

with try_all(errors) as s:
  s1
  s2
```

`pursue`: The macro `pursue` succeeds if one of the statements succeeds and fails if one of the statements does.

In this example, the instructions `s1` and `s2` are executed in parallel. If one completes successfully, the fluent `s` has the value `State.SUCCEEDED`. If one of the instructions fails, the value of the fluent is `State.FAILED`.

```
from pycram.language import pursue

with pursue(errors) as s:
  s1
  s2
```

`pursue` can be used to pursue two alternative goals and if one goal is reached, aborting the other. This can be done, again, by checking the state.

Error handling

Sometimes it is desired that a code gets re-executed when an error occurs. For this purpose, there is the macro `failure_handling`, which puts the passed code into a `retry` function, which in turn can be called in the code itself, and calls it. If you want to limit the number of possible attempts, you can pass this limit as parameter.

The following example executes the code c up to 5 times if an error occurs during execution.

```
from pycram.language import failure_handling

with failure_handling(5):
  try:
    c
  except Exception as e:
    retry()
```

If passing a negative or no parameter at all, the number of attempts is unlimited and the code is executed until there is no more error.

```
from pycram.language import failure_handling

with failure_handling():
  try:
    c
  except Exception as e:
    retry()
```

Implementation

The macros defined for sequential and parallel execution always return a fluent whose value is of the type `State`. `State` is a class that inherits from `Enum`, and defines the two values `SUCCEEDED` and `FAILED`.

```
class State(Enum):
  SUCCEEDED = 1
  FAILED = 2
```

To keep the code short and clean, we have defined some helper functions.

The function `_init` is used to initialize our macros. It will be given at least one parameter and optionally a second one. The first parameter `target` is the variable in which the state of the macro is stored. This is the variable that is after the "as" when the macro is called as in `with pursue(e)as s`. This variable stores a new `Fluent` object. Then a second variable `_exceptions` is created and assigned an empty list. If the second parameter `threads` has the value `True` (default as this happens more often), then another variable `_threads`, which again is assigned an empty list, is created. All this is returned as an abstract syntax tree.

```
def _init(target, threads = True):
  with hq as tree:
    ast_literal[target] = Fluent()
    _exceptions = []

  if threads:
    with q as temp_tree:
      _threads = []

    tree.append(temp_tree)

  return tree
```

The `_state` function expects as the first parameter the variable storing the state and optionally a new state as the second parameter. The variable is an object of the type `ast.Name` with `ast.Store` as the expression context, because macros typically assign a value to this variable. We have already assigned a fluent to the variable and now want to access it in order to call the functions `get_value` and `set_value` on it. To make this possible, we need to convert the object to a new object of type `ast.Name` with `ast.Load` as the expression context. Then it is checked whether a second parameter is given. If not, then the current state should be returned. This is returned as a tree by means of the macro `q`, so that the macros can process it more easily. If a second parameter is given, its value is assigned to the variable, provided that no value has been set before. Because the state of our macros is final, once set this should not change. Also, the setting of the value does not happen here, but is returned as a syntax tree and then executed by the macro itself.

```
def _state(target, state = None):
  target_load = ast.Name(target.id, ast.Load())

  if state is None:
    return q[ast_literal[target_load].get_value()]

  with hq as tree:
    if not ast_literal[target_load].get_value():
      ast_literal[target_load].set_value(state)

  return tree
```

The function `_exceptions` receives as parameters the tree generated by the macro and the list of arguments passed to the macro. If the number of arguments is greater than `0` and thus a variable in which a list of all errors is to be stored is given, the function generates a tree which assigns the value of the previously defined variable `_exceptions` to it. The tree is attached to the tree created by the macro.

```
def _exceptions(tree, args):
  if len(args) > 0:
    with hq as new_tree:
      ast_literal[ast.Name(args[0].id, ast.Store())] =
         unhygienic[_exceptions]

    tree.append(new_tree)
```

The next function is `_thread`. This one is only intended for the macros defined for parallel execution and expects the tree generated by the macro as a parameter. The function creates a subtree that creates a new `Thread` object. The function `_func` defined in the macro is passed to the constructor of the `Thread` object. The object is added to the list `_threads` and then the thread is started. The subtree is appended to the tree created by the macro.

```
def _thread(tree):
  with hq as new_tree:
    _thread = Thread(target = unhygienic[_func])
    unhygienic[_threads].append(_thread)
    _thread.start()

  tree.append(new_tree)
```

The last helper function `_join` is the counterpart to this. It also expects the tree generated by the macro as a parameter and creates a subtree that collects all the threads stored in the list `_threads`. Then the subtree is attached to the transferred tree.

```
def _join(tree):
  with hq as new_tree:
    for _thread in unhygienic[_threads]:
      _thread.join()

  tree.append(new_tree)
```

The macro `seq` generates a new tree by calling the function `_init`. For each statement in the code passed as a tree, it is first checked whether no state has yet been set. In this case the statement will be tried. If an error occurs, the state is set to `State.FAILED` and the error is added to the list `_exceptions`. At the end the function `_state` tries to set the state to `State.SUCCEEDED`, which only happens if no value has been set before. By calling the helper function `_exceptions` the errors are assigned to the passed parameter.

```
@macros.block
def seq(tree, target, args, **kw):
  new_tree = _init(target, False)

  for statement in tree:
    with hq as temp_tree:
      if ast_literal[_state(target)] is None:
        try:
          ast_literal[statement]
        except Exception as e:
          ast_literal[_state(target, State.FAILED)]
          unhygienic[_exceptions].append(e)

    new_tree.append(temp_tree);

  new_tree.append(_state(target, State.SUCCEEDED))
  _exceptions(new_tree, args)
  return _block(new_tree)
```

`try_in_order` is identical to `seq`, except that here the state is changed to `State.SUCCEEDED` if an instruction was executed successfully. In the end, it is then logically attempted to set the state to `State.FAILED` if it has not yet been assigned a value.

```
@macros.block
def try_in_order(tree, target, args, **kw):
  new_tree = _init(target, False)

  for statement in tree:
    with hq as temp_tree:
      if ast_literal[_state(target)] is None:
        try:
          ast_literal[statement]
          ast_literal[_state(target, State.SUCCEEDED)]
        except Exception as e:
          unhygienic[_exceptions].append(e)

    new_tree.append(temp_tree);

  new_tree.append(_state(target, State.FAILED))
  _exceptions(new_tree, args)
  return _block(new_tree)
```

par behaves similarly to `seq`, but executing the statements here is wrapped in a function `_func`, and the helper function `_thread` is called to start a thread that performs this function. After executing all statements, the `_join` function is called to wait for all threads to terminate.

```
@macros.block
def par(tree, target, args, **kw):
  new_tree = _init(target)

  for statement in tree:
    with hq as temp_tree:
      def _func():
        try:
          ast_literal[statement]
        except Exception as e:
          ast_literal[_state(target, State.FAILED)]
          unhygienic[_exceptions].append(e)

    new_tree.append(temp_tree)
    _thread(new_tree)

  _join(new_tree)
  new_tree.append(_state(target, State.SUCCEEDED))
  _exceptions(new_tree, args)
  return _block(new_tree)
```

`try_all` behaves to `par` as `try_in_order` does to `seq`. It is identical, but changes the state to `State.SUCCEEDED` if a statement was executed successfully. At the end, the state is set to `State.FAILED` if no value has been assigned previously.

```
@macros.block
def try_all(tree, target, args, **kw):
  new_tree = _init(target)

  for statement in tree:
    with hq as temp_tree:
      def _func():
        try:
          ast_literal[statement]
          ast_literal[_state(target, State.SUCCEEDED)]
        except Exception as e:
          unhygienic[_exceptions].append(e)

    new_tree.append(temp_tree)
    _thread(new_tree)

  _join(new_tree)
  new_tree.append(_state(target, State.FAILED))
  _exceptions(new_tree, args)
  return _block(new_tree)
```

The macro `pursue` combines the macros `par` and `try_all`. It sets the state to `State.SUCCEEDED` if an instruction was executed successfully and to `state.FAILED` if one failed.

```
@macros.block
def pursue(tree, target, args, **kw):
  new_tree = _init(target)

  for statement in tree:
    with hq as temp_tree:
      def _func():
        try:
          ast_literal[statement]
          ast_literal[_state(target, State.SUCCEEDED)]
        except Exception as e:
          ast_literal[_state(target, State.FAILED)]
          unhygienic[_exceptions].append(e)

    new_tree.append(temp_tree)
    _thread(new_tree)

  _join(new_tree)
  _exceptions(new_tree, args)
  return _block(new_tree)
```

Our last macro, `failure_handling`, first checks if a parameter has been passed. If not, the variable in which the arguments are stored is assigned a new list with a negative value as the only element. So this negative value is the default value for our parameter, which is optional. A variable `_retries` is created and assigned a fluent with the value `0` to it. This is the counter for the already performed attempts to execute the passed code. Then a function `retry` is defined which checks whether the number of attempts is limited by a positive number passed as a parameter. If this is the case, it is checked whether the counter has reached this number. If so, it will be interrupted at this point. Otherwise, the counter is incremented and the code is executed. The function is then called once to start the process.

```
@macros.block
def failure_handling(tree, args, **kw):
  if len(args) == 0:
    args = [q[-1]]

  with hq as new_tree:
    _retries = Fluent(0)

    def retry():
      if ast_literal[args[0]] >= 0 and _retries.get_value()
          > ast_literal[args[0]]:
        return

      _retries.set_value(_retries.get_value() + 1)

      ast_literal[tree]

    retry()

  return _block(new_tree)
```

3.3 Symbolic Plan Parametrization

Author: Dustin AUGSTEN

For a control program to be as general and flexible as possible, one should make decisions based on parameters such as the current location of the robot. Designators make it easy to describe motions as well as locations and objects as key-value pairs. The following describes a round object.

```
[('shape', 'round')]
```

More key-value pairs allow the round object to be further limited. For example, we can also define that it must be yellow.

```
[('shape', 'round'), ('color', 'yellow')]
```

We further limit the object that it must be at the location `loc`. In this case, `loc` itself could be a designator that describes the location.

```
[('shape', 'round'), ('color', 'yellow'), ('at', loc)]
```

Basically we limit the possible solutions of a designator by each key-value pair a little further. In the first example, the designer describes each round object, in the last example it describes round objects that are also yellow and at the given location. We assume that `loc` is also a designator and, thus, describes several possible locations that meet all properties. Designators, thus, receive properties, which are defined by constraints, and can be evaluated, so that either no, exactly one or several solutions are returned.

3.3.1 Designator Concepts

Author: Andy AUGSTEN

In the Lisp implementation CRAM, there are various types of designators, these being Motion Designators, Object Designators and Location Designators. For our purposes, we have used and implemented only one type of designator, the Motion Designator to describe motions. Although one would use Object Designators to describe objects and Location Designators to describe locations, the different types of designators are similar in many of their properties. For this reason, there is the superclass `Designator`, from which in our case `MotionDesignator` inherits.

Below are methods of the `Designator` class.

`equate`: When a designator is initialized, its properties cannot be changed anymore, and if a designator has calculated a solution, it should not change either. If two designators describing the same entity provide a different solution, then two designators must be created, which can then be equated by the `equate` function. Two equated designators always describe the same entity. This way a chain of equated designators can be built to track the changes of a designator over time.

`equal`: The `equal` function returns `True` if two designators are the same and thus describes the same entity, otherwise `False`.

`first`: This function returns the first ancestor in the chain of equated designators, id est the beginning of the equated chain.

`current`: This one returns the most recent designator, this being the last one equated to the current designator or one of its equated designators, id est the end of the equated chain.

`copy`: Calling this function creates a new designator that has the same properties. Additional properties, which are then joined, can be passed as parameter. The transferred properties are dominant which means if their key already exists, the values are overwritten.

`make_effective`: This function can be used to manually create an effective designator of the same type. In general, a designator becomes an effective one by referencing. An effective designator describes a low-level data structure that corresponds to a specific entity in the world. Id est an abstract object designator becomes a real object from the environment, or a location designator turns into a 3D pose. Properties can be passed as parameter to the function. If none are given, the properties of the current designator are taken over. The second parameter is the low-level data structure, the default value is `None`. The third parameter can be the timestamp for creating the reference, the default value is the current timestamp.

`newest_effective`: The `newest_effective` function returns the latest effective designator in the chain of equated designators.

`prop_value`: Calling this function returns the value of the passed key.

`check_constraints`: This function expects properties as parameter and returns `True` if they are satisfied, otherwise `False`. The properties are passed as a list. Each element can be a tuple, in which case the first value is the key of a property that must match the second value. If the element is not a tuple, it simply corresponds to the key of a property, which must not be `None`.

The following example checks whether the `'shape'` property has the value `'round'` and if the `'color'` property is given.

```
d = Designator([('shape', 'round'), ('color', 'yellow'), ('
    at', location)])
d.check_constraints([('shape', 'round'), 'color']) # True
```

`make_dictionary`: The function `make_dictionary` returns the given properties as a dictionary. The properties to be considered are transferred as a list. If an element of the list is a tuple, a property with the first value as a key and the second value as its value is added to the dictionary. If the element is not a tuple, then the property, which has the element as a key, is added.

In the following example we apply the function to our designator `d`.

```
d.make_dictionary(['shape', ('color', 'red'), ('foo', 'bar'
    ), 'foobar'])
```

As a result we get:

```
{'shape': 'round', 'color': 'red', 'foo': 'bar', 'foobar':
    None}
```

Implementation

When you create a designator, its properties are passed as parameter. Optionally, a parent designator can be handed over, which is then equated with this. Apart from the properties, all instance variables are assigned their default values. Only the variable `timestamp` should be accessible, for all others there are corresponding functions.

```
class Designator:
  def __init__(self, properties, parent = None):
    self._mutex = Lock()
    self._parent = None
    self._successor = None
    self._effective = False
    self._data = None
    self.timestamp = None
    self._properties = properties

    if parent is not None:
      self.equate(parent)
```

The function `equate` first checks whether the passed parent designator is already equated with this designator. If so, it will stop here because there is nothing to do. Otherwise, the two designators will be equated.

```
  def equate(self, parent):
    if self.equal(parent):
      return

    youngest_parent = parent.current()
    first_parent = parent.first()

    if self._parent is not None:
      first_parent._parent = self._parent
      first_parent._parent._successor = first_parent

    self._parent = youngest_parent
    youngest_parent._successor = self
```

`equal` checks whether the designator is equal to the given one. Two designators are equal if their first ancestors are identical.

```
  def equal(self, other):
    return other.first() is self.first()
```

The functions `first` and `current` return the first ancestor or the most recent equivalent designator. For this purpose, it is checked whether the parent designator or the child designator is `None`. If so, the current designator will be returned. Otherwise, the respective function is called recursively to the parent or child designator.

```
def first(self):
  if self._parent is None:
    return self

  return self._parent.first()

def current(self):
  if self._successor is None:
    return self

  return self._successor.current()
```

`copy` copies the properties of the designator and merges them with the submitted ones. If necessary, properties are overwritten here, with the new ones being dominant. A new designator of the same class is created, passing the assembled properties to the constructor.

```
def copy(self, new_properties = None):
  properties = self._properties.copy()

  if new_properties is not None:
    for key, value in new_properties:
      replaced = False

      for i in range(len(properties)):
        k, v = properties[i]

        if k == key:
          properties[i] = (key, value)
          replaced = True

      if not replaced:
        properties.append((key, value))

  return self.__class__(properties)
```

The `make_effective` function checks if the passed properties are `None`. If so, the properties of the current designator are applied. A new designator of the same type is created, it is marked as effective and its low-level data structure is set to the one passed. If a timestamp has been passed, this is also set, otherwise the current one will be used. The newly created designator is then returned.

```
def make_effective(self, properties = None, data = None,
    timestamp = None):
  if properties is None:
    properties = self._properties

  desig = self.__class__(properties)
  desig._effective = True
  desig._data = data

  if timestamp is None:
    desig.timestamp = time()
  else:
    desig.timestamp = timestamp

  return desig
```

`newest_effective` defines an internal `find_effective` function that calls itself recursively, always passing the parent designator to find the latest effective designator. The current designator is returned if this is `None` – in that case there is no effective designator in the chain – or if it is an effective one. The function is first called with the newest designator in the chain as parameter.

```
def newest_effective(self):
  def find_effective(desig):
    if desig is None or desig._effective:
      return desig

    return find_effective(desig._parent)

  return find_effective(self.current())
```

The `prop_value` function iterates over all properties to find those with the correct key and return its value. If no property was found, `None` is returned.

```
def prop_value(self, key):
  for k, v in self._properties:
    if k == key:
      return v

  return None
```

To check the passed properties, the function `check_constraints` iterates over them. For each element it is checked if it is a tuple. If so, it checks if the value does not match to return `False` in this case. If it is not a tuple, it checks if the value is `None` and then returns `False`. Otherwise, all properties are met and `True` is returned.

```
def check_constraints(self, properties):
  for prop in properties:
    if type(prop) == tuple:
      key, value = prop

      if self.prop_value(key) != value:
        return False
    else:
      if self.prop_value(prop) is None:
        return False

  return True
```

`make_dictionary` creates a new `Dictionary` object and iterates over the passed properties. If an element is a tuple, it is added to the dictionary as a property. If it is not a tuple, a new property with the item as the key and the value returned by `prop_value` as the value is added. The dictionary is then returned.

```
def make_dictionary(self, properties):
  dictionary = {}

  for prop in properties:
    if type(prop) == tuple:
      key, value = prop
      dictionary[key] = value
    else:
      dictionary[prop] = self.prop_value(prop)

  return dictionary
```

3.3.2 Designator Resolution

Author: Dustin AUGSTEN

The API provided by the `Designator` class is the same for all types of Designators. But resolving a designator may differ depending on the type, so this must be implemented by the user. For this purpose, the class `Designator` can be inherited from to override the functions which are responsible for the resolution.

The `reference` function should dereference a designator and return the low-level data structure or throw a `DesignatorError` if it is not an effective designator. So that the user does not have to worry about thread safety and because the variable `_effective` must always be set to `True` and `_timestamp` – if not yet set – to the current timestamp, there is a second function `_reference`. `reference` calls this function, so this one can be overridden instead.

```
def _reference(self):
  pass

def reference(self):
  with self._mutex:
    ret = self._reference()

  self._effective = True

  if self.timestamp is None:
    self.timestamp = time()

  return ret
```

`next_solution` should return a second solution for an effective designator or `None` if none exists. However, this is not meant to return the designator's low-level data structure, but a new designator with the same characteristics as the source designator, because they both describe the same entity.

```
def next_solution(self):
  pass
```

To return a list of all solutions as a low-level data structure, the API also provides a `solutions` function. This uses the functions to be overridden by the user and is thus the same for all types of designators and therefore does not need to be overwritten. As optional parameter the function can be given a value and if this is not `None` (basically this is a boolean value, but here we check to `None` to get closer to the Lisp implementation) instead of the current designator the first in the chain of equated designators is used. The function returns a generator that calls the `reference` function within a loop, returns the return value and then calls `next_solution` until the latter returns `None`.

```
  def solutions(self, from_root = None):
    if from_root is not None:
      desig = self.first()
    else:
      desig = self

    def generator(desig):
      while desig is not None:
        try:
          yield desig.reference()
        except DesignatorError:
          pass

        desig = desig.next_solution()

    return generator(desig)
```

DesignatorError is a simple class inheriting from Exception.

```
class DesignatorError(Exception):
  def __init__(self, *args, **kwargs):
    Exception.__init__(self, *args, **kwargs)
```

Motion Designators

The following sections discuss the implementation of the MotionDesignator class. A motion designator is a type of designator that describes motions. The class provides a public instance variable resolvers. This is a list in which the user must store the resolvers to be used by the class. Resolvers are functions that take a designator as a parameter and return a list of solutions. A solution may itself be a generator or a generator function that generates solutions.

```
class MotionDesignator(Designator):
  resolvers = []
```

During initialization, two variables _solutions and _index are defined and initially assigned default values. _solutions gets assigned a generator list in the function _reference. This is a list based on a generator. The generator will generate the solutions, the list stores them so that they can be accessed again. _index stores the index of the current solution because a call to the reference function should always return the same. For a new designator returned by calling the function next_solution, the value is incremented.

```
def __init__(self, properties, parent = None):
  self._solutions = None
  self._index = 0
  Designator.__init__(self, properties, parent)
```

`_reference` checks if `_solutions` has already been assigned a value to. If not, a generator is defined. This traverses through all resolvers, executes them and then traverses through the returned solutions. If it is a generator function, it is called to get the generator. If the solution is a generator, a loop will be run in which the next element of the generator is always returned. Otherwise, the current element will be returned as a solution. The generator generated by the function `_reference` is packaged as a generator list and stored in `_solutions`. If a low-level data structure already exists, it will be returned. Otherwise, the element at position `_index` in the generator list is stored in `_data` and returned. If an error occurs because the list is empty, it is not an effective designator and a `DesignatorError` is thrown.

```
def _reference(self):
  if self._solutions is None:
    def generator():
      for resolver in MotionDesignator.resolvers:
        for solution in resolver(self):
          if isgeneratorfunction(solution):
            solution = solution()

          if isgenerator(solution):
            while True:
              try:
                yield next(solution)
              except StopIteration:
                break
          else:
            yield solution

    self._solutions = GeneratorList(generator)

  if self._data is not None:
    return self._data

  try:
    self._data = self._solutions.get(self._index)
    return self._data
  except StopIteration:
    raise DesignatorError('Cannot␣resolve␣motion␣
      designator')
```

The function `next_solution` calls the function `reference` to ensure that `_solutions` is set. If an error occurs, it is intercepted and ignored because `next_solution` should not throw an error when called on a non-effective designator. This would then do the function `reference` later when it is called again. It is checked if there is another solution in `_solutions`. If so, a new designator with the same properties is created and returned. The value for the variable `_solutions` is adopted, that of `_index` is incremented. If no other solution exists, `None` is returned.

```
  def next_solution(self):
    try:
      self.reference()
    except DesignatorError:
      pass

    if self._solutions.has(self._index + 1):
      desig = MotionDesignator(self._properties, self)
      desig._solutions = self._solutions
      desig._index = self._index + 1
      return desig

    return None
```

`GeneratorList` is a class implemented by us and added the `helper` module. When initializing, a generator or a generator function is passed as a parameter. If it is a generator function, it is called to get the generator. The generator is stored in the variable `_generator`. A list for the generated elements is created and stored in `_generated`.

```
class GeneratorList:
  def __init__(self, generator):
    if isgeneratorfunction(generator):
      self._generator = generator()
    else:
      self._generator = generator

    self._generated = []
```

The `get` function is passed an index. It will loop through as long as the number of generated elements is less than or equal to the index. Within the loop, an element is always generated and stored in `_generated`. Then the element at the given position is returned.

```
  def get(self, index = 0):
    while len(self._generated) <= index:
      self._generated.append(next(self._generator))

    return self._generated[index]
```

The `has` function can be used to check if there is an element at a given position. The position is passed to the function as a parameter. The position is then passed to the get function. If there was no error, the element exists and `True` is returned, otherwise `False`.

```
  def has(self, index):
    try:
      self.get(index)
      return True
    except StopIteration:
      return False
```

3.4 Process Modules

Author: Andy AUGSTEN

One of the most important features that PyCRAM offers is to abstract from the actual hardware of the robot when implementing control programs. To this end, plans are designed using designators, and these are executed by process modules that communicate with the robot's hardware. Thus, plans can be implemented very generally and only process modules corresponding to the different types of robots that communicate with the respective hardware need to be implemented.

Any number of process modules covering different areas can be implemented for each robot. Thus, e.g. for the PR2, a left arm process module and a right arm process module are implemented so that they can work in parallel. A robot with only one arm can handle the provided designator differently, i.e. in this case ignore which arm the motion should be executed with and just use the same plan.

3.4.1 The PyCRAM Process Module Interface

Author: Dustin AUGSTEN

The class `ProcessModule` provides a variable `resolvers`. This is a list in which all resolvers must be stored. Resolvers for process modules are functions that receive a designator as a parameter and check its properties to select a suitable process module and return it, or `None` if none exists.

```
class ProcessModule:
  resolvers = []
```

The function `perform` receives a designator as parameter. It traverses through all resolvers and tries to find a process module. If a module is found, the `execute` function is called on it.

```
  def perform(designator):
    for resolver in ProcessModule.resolvers:
      pm = resolver(designator)

      if pm is not None:
        return pm.execute(designator)
```

When initializing a process module, the two variables _running and _designators are created. _running is a fluent that indicates whether the process module is currently running and thus processing a designator. _designators is a list in which all designators still to be processed are stored.

```
def __init__(self):
    self._running = Fluent(False)
    self._designators = []
```

execute is a function that receives a designator as a parameter and should process it. To do this, the designator is stored in the _designators list and the wait_for function is called on the fluent network which results from comparing the fluent _running with the value False. That means if the process module is busy it will wait for it to be available again. If the thread is not currently blocked, the value of _running will be set to True. Then, the designator, which is at the first position in the _designators list, is passed to the _execute function and removed from the list. Then the value of _running is set to False again. If a new process module is created, then it must inherit from the class ProcessModule and override the function _execute to implement the communication with the hardware.

```
def _execute(self, designator):
    pass

def execute(self, designator):
    self._designators.append(designator)
    (self._running == False).wait_for()
    self._running.set_value(True)
    designator = self._designators[0]
    ret = self._execute(designator)
    self._designators.remove(designator)
    self._running.set_value(False)
    return ret
```

3.4.2 Synchronous and Asynchronous Execution

Author: Andy Augsten

A process module is always responsible for one component of a robot and, thus, synchronous. An arm, for example, can only perform one movement at a time. If you try to perform several actions at the same time, they will be accumulated in a queue and executed one after the other. But if two arms are available to the robot, it is possible for one arm to execute a movement asynchronously to the other arm. So, to execute something asynchronously, you simply have to implement several process modules. In theory, it would also be possible to define multiple process modules that can be executed asynchronously to each other for one component, but in practice this does not always make sense.

The queue in which actions are accumulated is implemented as a simple list to which all designators, which are passed to the execute function, get added. The function calls wait_for on a fluent to block execution if it is already running in another thread. As soon as this specific thread finished, the function claims execution for the current thread, fetches the first designator in the list, processes it, removes it from the list and then releases execution again.

Chapter 4

Application

This chapter is all about the usage of PyCRAM which is explained and shown here using a concrete example application. The first part of this chapter will explain the example application running on the PR2 and the used concepts. After that part there is a part explaining how to set up the PyCRAM environment as this, due to the limited Python 3 support of ROS, does not work out of the box.

4.1 Demonstration Scenario

Author: Andy AUGSTEN

In order to test and verify PyCRAM, we wanted to implement an application that uses all of the CRAM concepts that were implemented in PyCRAM at this point. We decided to use the PR2 for this scenario and keep the task as simple as possible yet include all concepts.

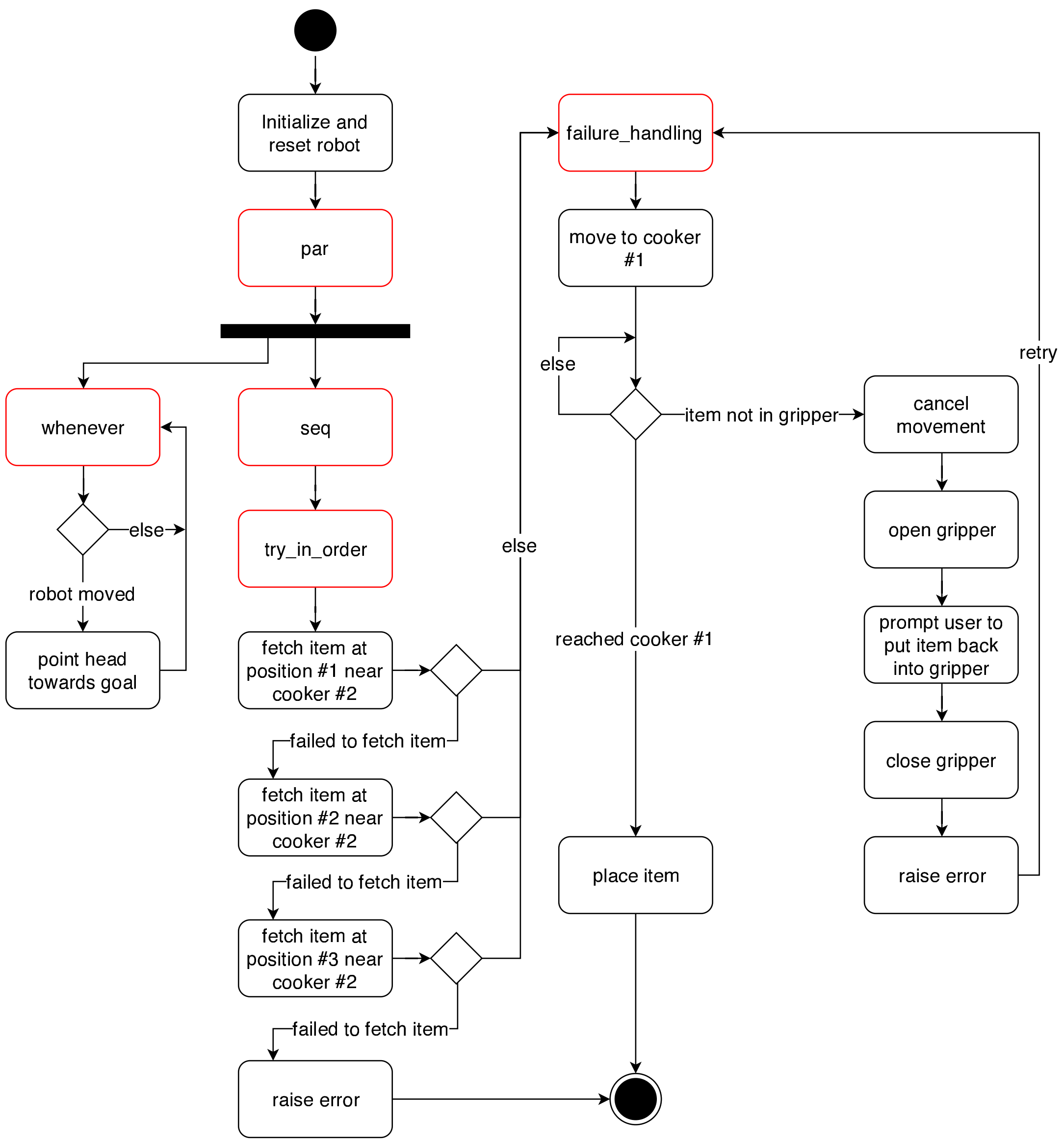


FIGURE 4.1: An activity diagram of the demo application. The red outlined activities are the PyCRAM language expressions that were used in our application.

In our example scenario the PR2 executes fetch and place tasks in a laboratory test kitchen which is shown in Figure 4.2 (p. 57). Our application, which can be seen as activity diagram in Figure 4.1 (p. 56), will first send the PR2 to a default start position at which the demo will start. Starting at this point the robot will always be looking at its goals simultaneously while doing its job. For example, if the robot is moving to position p_0 it will look at p_0 while driving towards it. If the robot is trying to fetch an item at position p_1 it will look at position p_1 while trying to fetch it and so on. This behavior is illustrated in Figure 4.3 (p. 57).

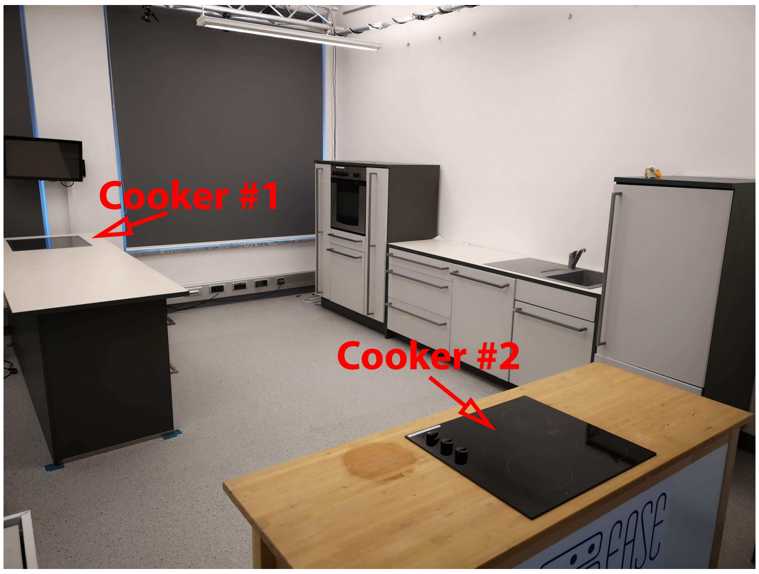


FIGURE 4.2: The laboratory test kitchen.

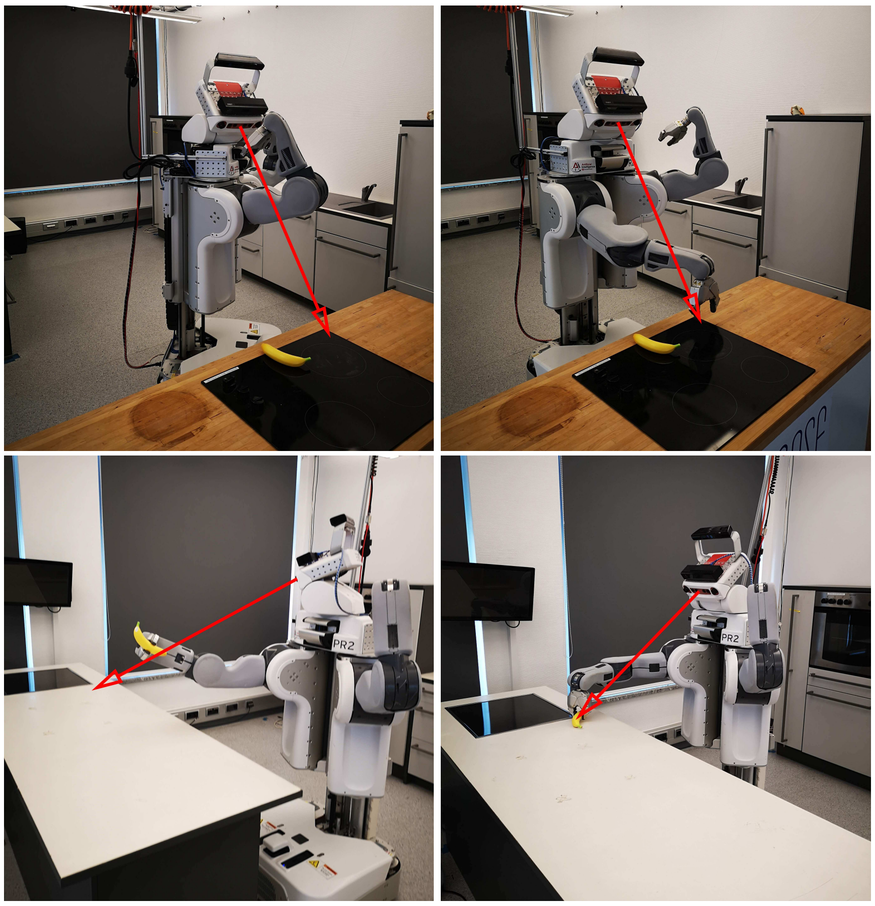
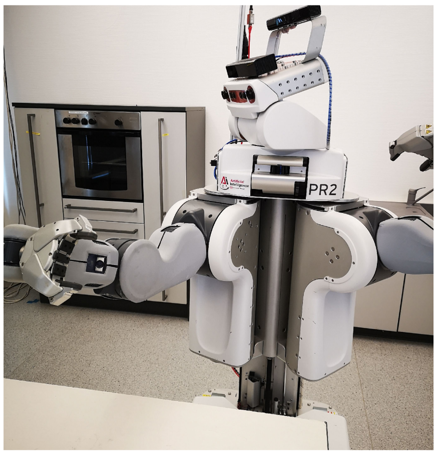
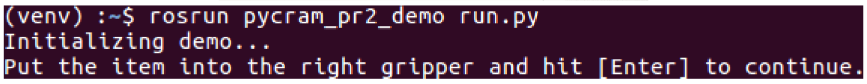


FIGURE 4.3: The PR2 simultaneously moving and looking towards its goal.

Next, it will move to a position in front of cooker #2 where it will try to fetch an item. If there is no item to fetch at the position the robot is standing at it will move a little and then try to fetch again. Altogether there are three attempts to fetch an item and the demonstration application will only continue if the PR2 successfully fetches an item at this point or terminate otherwise. Figure 1.1 (p. 2) shows how the robot tries to fetch an item from different locations. Assuming the PR2 was able to fetch an item, because this is a demonstration scenario to test PyCRAM and of course we placed an item at one of the three positions it is trying to grab an item from, it will then move to cooker #1. While moving to cooker #1 the PR2 will be tracking its gripper position to check whether it lost the item on its way. In case it loses the item it will immediately stop and the user will be requested to put the item back into the gripper, as can be seen in Figure 4.4 (p. 58). Only after the item has been successfully put back into the gripper the robot will continue moving to cooker #1. This step will repeat every time the item is lost on its way to cooker #1. We have tested this by stealing the robot's item multiple times on its way and once even confirming to have put the item back into the gripper without actually doing so. After reaching cooker #1 the last step is to place the item on it as seen in Figure 4.5 (p. 59). In order to make all this work we have implemented several functions and other lines of code that will be explained here. The code was split among five different files: *control.py*, *demo.py*, *motion_designators.py*, *process_modules.py* and *run.py*.





```
(venv) :~$ rosrun pycram_pr2_demo run.py
Initializing demo...
Put the item into the right gripper and hit [Enter] to continue.
```

FIGURE 4.4: The PR2 lost the item. The user is prompted to put the item back into the gripper.

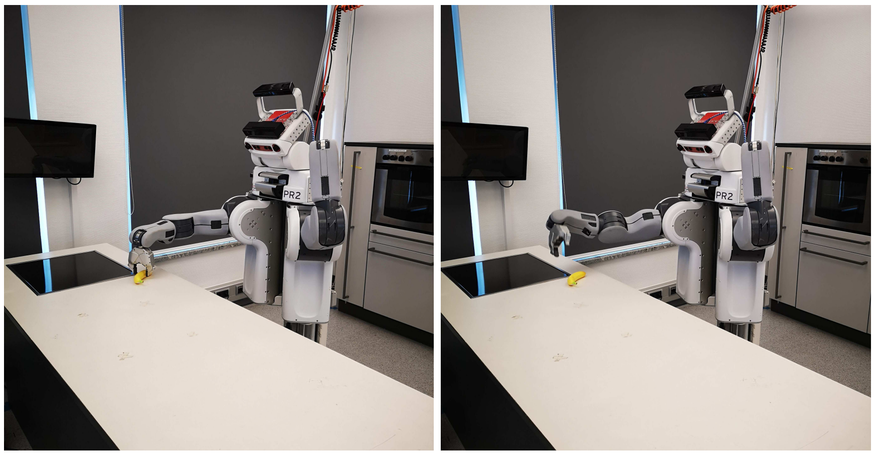


FIGURE 4.5: The PR2 placing the item.

PR2 has a number of interfaces, implemented as ROS actions, for the user to control it. These interfaces are used in the control.py file which is explained in more detail below. There is an interface to set the height of the torso, which expects a value for it's height. There are interfaces with which the programmer can set the gripper and the arms, which expect a value for how far the gripper should be opened or angles for the arm joints of the robot. Another interface can control the head and expects a position at which the cameras will point at. And of course there is also an interface to move the robot around, which expects a new position and an orientation the robot should face towards. But there are also interfaces with which the programmer can track positions and values of the robot by subscribing to these interfaces and handling callbacks or interfaces with which actions can be cancelled.

control.py

This file is handling all the controls for the PR2 robot.

The imports below are required to use the `signal` and `sys` packages which are used to track whether there was an interrupt signal for the running process and to stop all control actions in such a case.

```
import signal
import sys
```

Aside from `actionlib` and `rospy` which are used to handle message communication and actions the other imports are just different message types that were used.

```
import actionlib
import rospy

from geometry_msgs.msg import PoseWithCovarianceStamped
from move_base_msgs.msg import MoveBaseAction, MoveBaseGoal
from pr2_controllers_msgs.msg import ...
from trajectory_msgs.msg import JointTrajectoryPoint
```

The signal package includes the signal function which tells the interpreter what function to call on given signal. In our case exit is executed when SIGINT is received for the running process. Like already mentioned this is important for us to cancel all running actions on the robot when the process is killed.

```
signal.signal(signal.SIGINT, exit)
```

Bellow is the initialization part. The robot is receiving messages for each of its parts like the arms or the grippers and acts accordingly. The actionlib library provides a simple client that can be assigned to ROS topics to send the required messages. We have eight action clients to communicate with the robot's parts, which are the move, head, left and right arm, left and right gripper, torso and sound client.

```
move_client = actionlib.SimpleActionClient("/
   nav_pcontroller/move_base", MoveBaseAction)
move_client.wait_for_server()

head_client = actionlib.SimpleActionClient("/
   head_traj_controller/point_head_action", PointHeadAction
   )
head_client.wait_for_server()

...
```

Following is the function that is executed when the process receives a SIGINT. This function will tell all clients that were created earlier to cancel all goals, which basically are actions caused by the messages received for the client, and terminate. As a result, if the user interrupts the robot's program, all the robot actions will be cancelled and the robot will stop moving.

```
def exit(sig, frame):
  move_client.cancel_all_goals()
  head_client.cancel_all_goals()
  ...
  sys.exit()
```

The following two functions are using the rospy library to subscribe to given topics, receive a message of given type and save a value of that message to the given parameter fluent fl. Since the application code is robot-independent and we do not know how many grippers a robot has, the gripper value for subscribe_gripper is a number instead of l or r for left or right.

```python
def subscribe_pose(fl):
  rospy.Subscriber('/amcl_pose', PoseWithCovarianceStamped,
      lambda msg: fl.set_value(msg.pose))

def subscribe_gripper(gripper, fl):
  rospy.Subscriber('{}_gripper_controller/state'.format('l'
      if gripper == '0' else 'r'), JointControllerState,
     lambda msg: fl.set_value(msg.process_value))
```

The function move_to will create a new MoveBaseGoal message, adapt the settings of the pose parameter as goal and send the message to the robot using the move_client that was created earlier using actionlib. The parameter wait is True by default and defines whether the function should wait for the goal to be reached or not. In addition the user can define the frame the robot is supposed to do the movement in. By default that frame is "base_footprint". For example if the frame is set to "base_footprint" the point of origin is at the robot's position with the x-axis facing the way the robot faces while the frame "map" has it's origin at the origin of the map the robot is performing in, in our case the kitchen.
The function cancel_movement will call the already defined function cancel_all_goals on the move_client and stop the current movement of the robot.

```python
def move_to(pose, wait=True, frame="base_footprint"):
  msg = MoveBaseGoal()
  msg.target_pose.header.frame_id = frame
  msg.target_pose.pose.position.x = pose.position.x
  ...
  move_client.send_goal(msg)
  if wait:
    move_client.wait_for_result()

def cancel_movement():
  move_client.cancel_all_goals()
```

Just like the function move_to, the code below creates a new message and sends it to the robot using the appropriate action client after adapting the x, y and z coordinates. The parameters wait and frame serve the same purpose.

```
def point_head_to(x, y, z, wait=True, frame="base_footprint
   "):
  msg = PointHeadGoal()
  msg.target.header.frame_id = frame
  msg.target.point.x = x
  msg.target.point.y = y
  msg.target.point.z = z
  msg.max_velocity = 1.0
  head_client.send_goal(msg)
  if wait:
    head_client.wait_for_result()
```

The arm of the PR2 can be controlled by changing the values of the arm joints. Each arm joint has a motor which can be controlled by changing the joint value for that motor. The motor will then move that part of the arm to a specific angle. Altogether the PR2 has seven different arm joints:

shoulder pan joint	-	moves the shoulder of the robot back and forth
shoulder lift joint	-	moves the shoulder of the robot up and down
upper arm roll joint	-	turns the upper arm of the robot
elbow flex joint	-	stretches or bows the elbow of the robot
forearm roll joint	-	turns the forearm of the robot
wrist flex joint	-	moves the wrist towards the body of the robot or away from it
wrist roll joint	-	turns the wrist of the robot

Again a new message of type `JointTrajectoryGoal` is created to which a few adjustments are made. The parameter arm should be either `l` or `r` for left or right and is parsed into the `joint_names` for the message to be sent and the parameter positions is an array containing the positions for the named joints in the exact same order.

The function `set_gripper` is analog.

```
def set_arm_joints(arm, positions, wait=True):
  msg = JointTrajectoryGoal()
  msg.trajectory.joint_names = ['{}_shoulder_pan_joint'.
     format(arm), '{}_shoulder_lift_joint'.format(arm), '{}
     _upper_arm_roll_joint'.format(arm), '{}
     _elbow_flex_joint'.format(arm), '{}_forearm_roll_joint
     '.format(arm), '{}_wrist_flex_joint'.format(arm), '{}
     _wrist_roll_joint'.format(arm)]
  msg.trajectory.points = [JointTrajectoryPoint()]
  msg.trajectory.points[0].positions = positions
  msg.trajectory.points[0].velocities = [0, 0, 0, 0, 0, 0,
     0]
  if arm.lower() == 'l':
    l_arm_client.send_goal(msg)
    if wait:
      l_arm_client.wait_for_result()
  elif arm.lower() == 'r':
    r_arm_client.send_goal(msg)
    if wait:
      r_arm_client.wait_for_result()

def set_gripper(gripper, pos, wait=True, max_effort=25):
  msg = Pr2GripperCommandGoal()
  msg.command.position = pos
  msg.command.max_effort = max_effort
  if gripper.lower() == 'l':
    l_gripper_client.send_goal(msg)
    if wait:
      l_gripper_client.wait_for_result()
  elif gripper.lower() == 'r':
    r_gripper_client.send_goal(msg)
    if wait:
      r_gripper_client.wait_for_result()
```

The remaining functions are analog to the function `move_to`.

run.py

Like already mentioned in the foundations chapter about MacroPy, it is necessary to import MacroPy in seperate file to make it work. Since PyCRAM is using MacroPy it is necessary here too if one wants to use macros defined in PyCRAM, hence, *run.py* will import `macropy` and `demo`, which contains the code for our demo application. One needs to run *run.py*, i.e. this is the entry point of the program to start the demo application.

```
#!/usr/bin/env python

import macropy.activate
import demo
```

4.2 Used Concepts

Author: Dustin AUGSTEN

Although *control.py* and *run.py* are important for the application, the following three files demonstrate the concepts of PyCRAM. This chapter will explain how we used designators, process modules and the language expressions of PyCRAM.

4.2.1 Designators

The following examples will show how to properly define motion designators in PyCRAM using Python code.

motion_designators.py

First of all it is necessary to import `MotionDesignator`.

```
from pycram.designator import MotionDesignator
```

In the next step, a function is created that will serve as a resolver for given designator `desig`. A resolver is a function that takes a designator as a parameter and returns a list of solutions. For example, the resolver `pr2_motion_designators` below would return a list of solutions containing `[('cmd', 'set_gripper'), ('position', 1), ('wait', True), ('max_effort', 25)]` for the designator `MotionDesignator([('type', 'setting_gripper'), ('gripper', 'O'), ('position', 1)])`.

```
def pr2_motion_designators(desig):
  solutions = []
```

The resolver function `pr2_motion_designator` must contain a block as shown below for each different "type" of designator there is. The type `'moving'` for example has four arguments. While the arguments `'wait'` and `'frame'` are optional, the arguments for `'type'` and `'pose'` are required. The expression `desig.check_constraints(props)` will check if the arguments defined in `props` are set. For example `desig.check_constraints([('frame', 'map')])` is true only if the argument `'frame'` is set to `'map'` while `desig.check_constraints(['frame'])` is true if `'frame'` is set at all, aside from its actual value. Using this one can easily create such a nested block of code for every type, check for the set arguments and adapt the values of the arguments or use default ones. In line 9 the values of all arguments, aside from type, which is replaced with a `'cmd'` argument, are adopted and appended to solutions while in line 11 the value for `'frame'` was set to a custom defined value since `'frame'` is not set at this point. It is not necessary to replace type with `'cmd'` but depends on what is checked in process modules. We check in process modules for `'cmd'` so we did it this way. Finally the resolver will return the solutions list.

```
if desig.check_constraints([('type', 'moving'), 'pose']):
  if desig.check_constraints(['wait']):
    if desig.check_constraints(['frame']):
      solutions.append(desig.make_dictionary([('cmd', 'move'), 'pose', 'wait', 'frame']))

    solutions.append(desig.make_dictionary([('cmd', 'move'), 'pose', 'wait', ('frame', 'base_footprint')]))

  if desig.check_constraints(['frame']):
    solutions.append(desig.make_dictionary([('cmd', 'move'), 'pose', ('wait', True), 'frame']))

  solutions.append(desig.make_dictionary([('cmd', 'move'), 'pose', ('wait', True), ('frame', 'base_footprint')]))
```

Just like it is done for the designator of type moving such a block has to be defined for each type of designator the programmer wants to use.

```
  if desig.check_constraints([('type', 'setting_arm_joints'), 'positions']):
  if desig.check_constraints(['wait']):
    solutions.append(desig.make_dictionary([('cmd', 'set_arm_joints'), 'positions', 'wait']))

  solutions.append(desig.make_dictionary([('cmd', 'set_arm_joints'), 'positions', ('wait', True)]))

...

return solutions
```

And the last step is to append all resolvers to `MotionDesignator` which in our case is just the defined `pr2_motion_designators` function.

```
MotionDesignator.resolvers.append(pr2_motion_designators)
```

4.2.2 Process Modules

The motion designators we are using in our demo application are designators which are supposed to perform a motion. This is done in combination with process modules. Here we are going to show how we used the process modules of PyCRAM for our demo application.

process_modules.py

Since the process modules are going to perform motion or control actions in our case it is necessary to import our control functions and, of course, ProcessModule.

```
import control as c

from pycram.process_module import ProcessModule
```

The next step is to define a process module for each independent part of the robot. This means that if we want to use, for example, two arms simultaneously we have to create a process module for each of the two arms because a process module will always block until it has finished its action which would prevent simultaneous arm movements if there was one process module for both arms together.
A process module is defined by defining a class that inherits from ProcessModule and overwrites its _execute function, which tells the process module what to do. The programmer can use the given designator and get the parsed solution for it, which was done in *motion_designators.py* earlier, by calling the reference function in the _execute function. This was also done in the code block below and then the 'cmd' argument that was replaced earlier in the *motion_designators.py* file is checked. Then the appropriate function in *control.py* is simply called using the values given in the designator.

```
class Pr2Navigation(ProcessModule):
  def _execute(self, designator):
    solution = designator.reference()

    if solution['cmd'] == 'move':
      c.move_to(solution['pose'], solution['wait'],
         solution['frame'])
```

Like already mentioned, the same is done for every other independent part of the robot.

```
class Pr2Head(ProcessModule):
  def _execute(self, designator):
    solution = designator.reference()

    if solution['cmd'] == 'point_head':
      coordinates = solution['coordinates']
      c.point_head_to(coordinates[0], coordinates[1],
         coordinates[2], solution['wait'], solution['frame'
         ])

class Pr2ArmL(ProcessModule):
  def _execute(self, designator):
    solution = designator.reference()

    if solution['cmd'] == 'set_arm_joints':
      c.set_arm_joints('l', solution['positions'], solution
         ['wait'])

...
```

After all process modules have been defined an instance for each part need to be created since they basically are just classes and are useless without instancing them. The created instances are used in the process module resolver function that yet has to come.

```
pr2_navigation = Pr2Navigation()
pr2_head = Pr2Head()
pr2_arm_l = Pr2ArmL()
...
```

Now, just like it was done for the motion designators, the final step is to create resolver functions, in our case only one for the PR2, and add them to the resolvers of `ProcessModule`. Since we do not need any other arguments other than the type, we can use the given designator right away without getting its solution by calling the `reference` function. Each type is checked and the appropriate process module instance is returned. For example, a moving motion is dispatched to the `pr2_navigation` process module, setting the values for the arm joints to the `pr2_arm_l` or `pr2_arm_r` process module, depending on the given arm argument and so on. In our code the subscribing type designators are always performed synchronously, which is why they share the same process module while all other designator types are asynchronous. That is because we only need to subscribe to the PR2's position and the gripper's position once at the beginning of the application code and does not need to be done in parallel.

```
def available_pr2_process_modules(desig):
  if desig.check_constraints([('type', 'moving')]):
    return pr2_navigation

  if desig.check_constraints([('type', 'pointing_head')]):
    return pr2_head

  if desig.check_constraints([('type', 'setting_arm_joints'
     )]):
    if desig.check_constraints([('arm', '0')]):
      return pr2_arm_l

    if desig.check_constraints([('arm', '1')]):
      return pr2_arm_r

  ...

ProcessModule.resolvers.append(
   available_pr2_process_modules)
```

4.2.3 Language Expressions

The final step for our demo application is the demo itself, which is all included in the *demo.py* file. Here all the concepts of PyCRAM are put to use including the defined motion designators, process modules, fluents and the language expressions `failure_handling`, `par`, `seq`, `try_in_order`, `wait_for` and `whenever`. Although `pursue` was not explicitly used in this demo it works the same as `par` and was tested multiple times during the development.

demo.py

Below are all the dependencies that are used. It is really important to import `macros` from `fluent` and `language` for all the functions that use MacroPy to work which are nearly all language expressions of PyCRAM including `whenever` of fluents.

```
import math
import motion_designators
import process_modules
import rospy

from geometry_msgs.msg import Pose
from pycram.designator import MotionDesignator
from pycram.fluent import macros, Fluent, whenever
from pycram.language import macros, failure_handling, par,
   seq, State, try_in_order
from pycram.process_module import ProcessModule
```

At some parts of our code it was important to track the robot and the gripper positions, therefore we created fluents for these positions whose values are frequently updated.

```
pose = Fluent()
r_gripper_state = Fluent()
```

A helper function will reset the robot and its parts to default positions upon demo launch. The values for the positions were gathered beforehand. This is also the first time a language expression, namely `par`, was used in order to move all parts to their start position simultaneously. The language expression `par`, just like most of them, returns a state fluent which can hold the value `SUCCEEDED` or `FAILED` to describe the macro execution status. Since it is a fluent `wait_for` can be used on it which will block until the value of `state` has been set. This is after `par` has finished and all start positions are set. Afterwards the robot base will move to its start position. The reason why the movement is not done also within `par` is simply to prevent the robot hitting anything with its parts while moving within the kitchen.

```
def reset_robot():
  with par as state:
    ProcessModule.perform(MotionDesignator([('type', '
       setting_gripper'), ('gripper', '0'), ('position', 1)
       ]))
    ProcessModule.perform(MotionDesignator([('type', '
       setting_gripper'), ('gripper', '1'), ('position', 1)
       ]))
    ProcessModule.perform(MotionDesignator([('type', '
       setting_arm_joints'), ('arm', '0'), ('positions',
       [...])]))
    ProcessModule.perform(MotionDesignator([('type', '
       setting_arm_joints'), ('arm', '1'), ('positions',
       [...])]))
    ProcessModule.perform(MotionDesignator([('type', '
       setting_torso'), ('position', 0.3)]))

  state.wait_for()
  goal = Pose()
  goal.orientation.w = 1.0
  ProcessModule.perform(MotionDesignator([('type', 'moving'
     ), ('pose', goal), ('frame', 'map')]))
```

Below is where the demo actually starts. A new ROS node is initialized so the communication with ROS and the robot works, the motion designators `subscribing_pose` and `subscribing_gripper` are performed using process modules which will save the current positions of the robot and the right gripper to the previously created fluents `pose` and `r_gripper_state` and the helper function `reset` is called to get the robot to its default starting positions. Moreover two more variables `goal` and look_goal are defined. They will be overwritten several times during the run time. The variable goal holds the information for the robot's next goal position in the kitchen while the variable look_goal, as the name indicates, is the position the robot is supposed to look at. These two positions were separated because the robot continuously moves its head to keep looking at set position while moving around and if the robot would just look at its moving goal it would look either at the ground or the top when reaching the goal because it would stand on the goal, hence the look_goal is separated which basically is the position of the object the robot is going to interact with when reaching the goal.

```
print('Initializing␣demo...')
rospy.init_node('demo')
ProcessModule.perform(MotionDesignator([('type', '
   subscribing_pose'), ('fl', pose)]))
ProcessModule.perform(MotionDesignator([('type', '
   subscribing_gripper'), ('gripper', '1'), ('fl',
   r_gripper_state)]))

reset_robot()

goal = Pose()
look_goal = Pose()
```

Another helper function is grab_item which will grab an item in front of the set goal. par is used to move the robot to its goal and simultaneously start grabbing when getting close enough to the goal. Using Python's math library we checked if the distance or value of the hypotenuse towards the goal is less than 10 centimeters. Since this is done within a while loop combined with a sleep it will be checked every 0.1 second. Once the condition is fulfilled the robot will start grabbing which is just a sequence of setting arm joints, gripper positions and leaving the loop. When the loop has been left and the robot reached its goal we added an additional check for the gripper's position. It will be checked if the gripper is opened less than 1 centimeter which indicates in our demo that no item has been grabbed. In this case an error is raised that can be handled by the function caller. The grab_item function just shows another possible use case for par.

```
def grab_item():
  with par as state:
    ProcessModule.perform(MotionDesignator([('type', '
       moving'), ('pose', goal), ('frame', 'map')]))
    while True:
      p = pose.get_value().pose.position
      if math.hypot(goal.position.x-p.x, goal.position.y-p.
         y) < 0.1:
        joints1 = ...
        joints2 = ...
        joints3 = ...

        ProcessModule.perform(MotionDesignator([('type', '
           setting_arm_joints'), ('arm', '1'), ('positions'
           , joints1)]))
        rospy.sleep(3)
        ProcessModule.perform(MotionDesignator([('type', '
           setting_arm_joints'), ('arm', '1'), ('positions'
           , joints2)]))
        rospy.sleep(3)
        ProcessModule.perform(MotionDesignator([('type', '
           setting_gripper'), ('gripper', '1'), ('position'
           , 0)]))
        rospy.sleep(3)
        ProcessModule.perform(MotionDesignator([('type', '
           setting_arm_joints'), ('arm', '1'), ('positions'
           , joints3)]))
        break

      rospy.sleep(0.1)

  if r_gripper_state.get_value() < 0.01:
    ProcessModule.perform(MotionDesignator([('type', '
       setting_gripper'), ('gripper', '1'), ('position', 1)
       ]))
    raise
```

The next function was just created to make the code look cleaner. It is just a sequence of setting arm joints and gripper positions to predefined values. This function will open the gripper in front of the robot.

```
def place_item():
  joints1 = ...
  joints2 = ...

  ProcessModule.perform(MotionDesignator([('type', '
     setting_arm_joints'), ('arm', '1'), ('positions',
     joints1)]))
  rospy.sleep(3)
  ProcessModule.perform(MotionDesignator([('type', '
     setting_gripper'), ('gripper', '1'), ('position', 1)])
     )
  rospy.sleep(3)
  ProcessModule.perform(MotionDesignator([('type', '
     setting_arm_joints'), ('arm', '1'), ('positions',
     joints2)]))
```

At this point all variables have been initialized, the robot has been reset and helper functions are defined.

As written above, the robot should always look towards a set position, while performing any other actions. This was done using `par` again. While on the one hand the robot will execute a sequence of code for the demo on the other hand it will keep moving its head. In `par` we used the language expressions `seq` for the first part and the expression `whenever` for the moving head part. `whenever` is combined with the robot's position fluent that was created earlier and a `pulse` call. This means that whenever the value of `pose` has been changed the fluent `pose` will be pulsed and the body of `whenever` will execute. The body of `whenever` will perform the motion designator `pointing_head` towards the `look_goal`. This is how the robot will always look at the set position because it keeps moving its head whenever a movement has been made.

```
with par as s:
  with whenever(pose.pulsed()):
    ProcessModule.perform(MotionDesignator([('type', '
       pointing_head'), ('coordinates', [look_goal.position
       .x, look_goal.position.y, look_goal.position.z]), ('
       wait', False), ('frame', 'map')]))
    rospy.sleep(0.2)
  with seq as s:
```

The first step of the demo is for the robot to drive to cooker #2 and try to grab an item. If it fails it will move a little and try again altogether three times. This is how it was described earlier and how it is done here. Using the language expression `try_in_order` this can be easily implemented. It executes given expressions one after another until one succeeds. We put each grabbing part into an `if True` block so the whole grabbing part including setting the positions is handled as one expression. Else `try_in_order` would finish after setting `goal.position.x` already. Since the helper function `grab_item` already includes the moving part only setting the goals and calling the function is needed in each block. We have defined a number of fixed poses in the kitchen. These poses are all next to each other at cooker #2 and can be seen in the code snippet below. The robot is trying to grasp an object from each of these fixed poses and the `grab_item` function will raise an error if no item could be grabbed which will set the value of `state` to `FAILED` and continue to the next block. If one block succeeded or there are no more blocks left `try_in_order` is left and we check for the value of `state`. Only if its value is not `FAILED` an item has been successfully grabbed and the demo will continue.

```
with try_in_order as state:
  if True:
    goal.position.x = 0.1
    goal.position.y = -0.6
    goal.orientation.z = -0.700573543807
    goal.orientation.w = 0.713580205526

    look_goal.position.x = 0.1
    look_goal.position.y = -1.1
    look_goal.position.z = 0.8

    grab_item()
  if True:
    goal.position.x = -0.2
    goal.position.y = -0.6
    goal.orientation.z = -0.700573543807
    goal.orientation.w = 0.713580205526

    look_goal.position.x = -0.2
    look_goal.position.y = -1.1
    look_goal.position.z = 0.8

    grab_item()
  if True:
    goal.position.x = 0.4
    goal.position.y = -0.6
    goal.orientation.z = -0.700573543807
    goal.orientation.w = 0.713580205526

    look_goal.position.x = 0.4
    look_goal.position.y = -1.1
    look_goal.position.z = 0.8

    grab_item()

if state.get_value() == State.FAILED:
  print('Could␣not␣grab␣any␣item.')
  raise
```

After successfully grabbing an item the robot should move to cooker #1 and place the item there. While moving to cooker #1 it should simultaneously check if the item has been lost, stop in such a case and wait for it to be put back into the gripper. Again one of the language expressions PyCRAM provides will help to implement this in just a few lines of code. First of all the goal positions are set and then the expression `failure_handling` is used. It provides the functionality of executing code and catch its exceptions with the option to retry the execution. Using this feature we send the command to move to cooker #1 and then inside a `while` loop we check if the gripper's position ever changes below `1` centimeter, which again in our demo indicates that the robot is not holding anything. In that case the `cancelling_movement` designator is performed and the robot will open its gripper. The user will then see a request to put the item back into the gripper. After doing so an error is raised to let the `failure_handling` expression catch the exception and restart the execution which will send the robot moving again if the item has been put back in. However, if the item has not been lost the next check inside the `while` loop is whether the robot reached its goal in which case the loop is left and `failure_handling` as well since the execution of the code has finished. And last the item is placed on cooker #1 using the helper function `place_item`.

```
    goal.position.x = -0.1
    goal.position.y = 1.9
    goal.orientation.z = 0.9999981068
    goal.orientation.w = 0.00194586635055

    look_goal.position.x = -0.6
    look_goal.position.y = 1.9
    look_goal.position.z = 1.0

    with failure_handling:
      try:
        if r_gripper_state.get_value() > 0.01:
          ProcessModule.perform(MotionDesignator([('type',
             'moving'), ('pose', goal), ('wait', False), ('
             frame', 'map')]))

        while True:
          p = pose.get_value().pose.position

          if r_gripper_state.get_value() < 0.01: # if lost
             item
            ProcessModule.perform(MotionDesignator([('type'
               , 'cancelling_movement')]))
            ProcessModule.perform(MotionDesignator([('type'
               , 'setting_gripper'), ('gripper', '1'), ('
               position', 1)]))
            input('Put␣the␣item␣into␣the␣right␣gripper␣and␣
               hit␣[Enter]␣to␣continue.')
            ProcessModule.perform(MotionDesignator([('type'
               , 'setting_gripper'), ('gripper', '1'), ('
               position', 0)]))
            raise
          elif math.hypot(goal.position.x-p.x, goal.
             position.y-p.y) < 0.1:
            break

          rospy.sleep(0.1)
      except:
        retry()

    place_item()
```

Altogether this is a simple example application that shows how the use of PyCRAM can ease tasks and how to use PyCRAM.

4.3 Setup

Author: Andy AUGSTEN

As PyCRAM is based on Python 3 while ROS and the robotics community work with a big extend using Python 2, it was necessary to set up a virtual environment for Python 3 and ROS usage.

In order to set up a virtual environment for Python and make ROS work with Python 3 a few packages need to be installed. Since we used Ubuntu as operating system the following shell commands did the job:

```
$ sudo apt-get install python-pip python3-pip libbullet-dev
    python-virtualenv
```

The *libbullet-dev* package is required for the ROS *geometry2* package that was used in this example application. While there are some ROS packages working out of the box with the virtual Python 3 environment there are also packages that do not. These packages need to be downloaded, compiled and sourced manually.

A new ROS workspace was set up to which the required packages were pulled.

```
$ cd $HOME
$ mkdir ros_catkin_ws
$ mkdir ros_catkin_ws/src
$ cd ros_catkin_ws/src
$ git clone https://github.com/ros/geometry
$ git clone https://github.com/ros/geometry2
$ git clone https://gitlab.informatik.uni-bremen.de/
   aaugsten/pycram.git
```

The next step is to finally create the required virtual environment for Python 3 and activate it.

```
$ cd $HOME/ros_catkin_ws
$ virtualenv -p /usr/bin/python3 venv
$ source venv/bin/activate
```

The next step is optional. The activation part needs to be done each time the virtual environment was deactivated or the machine was rebooted, hence, it is sourced on boot on our setup.

```
$ echo source $HOME/ros_catkin_ws/venv/bin/activate >>
   $HOME/.bashrc
```

Now that Python is operating inside the virtual environment some Python packages need to be installed. These packages will only be installed inside the virtual environment. Since the virtual environment is using Python 3 it will install the Python 3 compatible packages.

```
$ pip install -U rosdep rosinstall-generator wstool
   rosinstall ros-buildfarm rosdistro rospkg pyyaml
   catkin_pkg
```

Finally, the ROS workspace can be recompiled and sourced.

```
$ cd $HOME/ros_catkin_ws
$ catkin_make
$ source devel/setup.bash
```

Just like the virtual environment, this also needs to be resourced every time the system is booted. This is done automatically on system boot here. Again this is optional to prevent the need for resourcing the ROS workspace on every system boot.

```
$ echo $HOME/ros_catkin_ws/devel/setup.bash >> $HOME/.
   bashrc
```

Now ROS is using Python 3 as default Python interpreter and PyCRAM is ready for use.

Chapter 5

Conclusion

Nowadays more istitutes than ever before own a robot such as the PR2 which operate in domestic environments such as a household, hence planning and the development of control programs for navigation and manipulation is now even more important. For this purpose CRAM was invented and now with this thesis also PyCRAM, a similar implementation for the Python language.

The PyCRAM Plan Language is a domain specific language on top of Python. It uses the MacroPy library which provides a strong and powerful macro system. With fluents, which allow to wait for specific changes and which can be combined to networks in order to express complex conditions, and the macros it provides, it supports the user in developing highly concurrent control programs to monitor different sensors in parallel without having to set up complex threading structures and without having to care about synchronization. In addition, the provided language constructs come with an easy error handling concept which also allows to re-execute a block when an error occured.

When developing a control program we want it to be as general and flexible as possible and hence make decisions based on parameters such as the current location of the robot. To accomplish this there is the designator concept. Designators can be used to describe motions but also objects and locations as key-value pairs. They provide resolution algorithms to resolve given parameters. There are different types of designators which are identical in many of their properties but the algorithm used for resolving them may differ. One type of designator, the Motion Designator, has been implemented by us to resolve motions in our demo application.

One of the most important features that PyCRAM offers is to abstract from the actual hardware of the robot. For this purpose, plans are implemented using designators and these are then executed by process modules that communicate with the robot's hardware. This makes it possible to implement plans in a general way while implementing process modules corresponding to the different types of robots. For example, it is possible to define a plan for picking up an object and use it for a robot with only one arm but also for a robot with two arms. The process module implemented for a specific robot can decide how to interpret and execute a plan. Process modules also take care of synchronization. If we execute two motions using the same process module, it stores both motions in a queue and executes them one after the other. We usually define one process module for each component of the robot like one for the left arm and one for the right arm which allows us to execute actions on both arms asynchronously to each other. Theoretically it is also possible to implement two process modules for only one component and, thus, making it possible to execute two actions at the same time, however, this does not make sense in most case.

As alternative to Prolog we have defined the two functions `check_constraints` and `make_dictionary` in the `Designator` class. These can be used by designator resolvers to easily check properties and create a solution as well as by process modules to check properties again. This allows to keep resolvers short and clean. Resolvers are functions that expect a designator to be passed as parameter. Designator resolvers return a solution, and process module resolvers return a process module.

PyCRAM is split into several modules and the user only has to load whatever is needed. For example, if the user only wants to use fluents, then (s)he only has to load the `fluent` module. If one wants to use macros, however, the program must run from a bootstrap file.

Besides the implementation of PyCRAM, we also presented a demo application in our thesis. It demonstrates how the PR2 operates fetch and place tasks in a laboratory test kitchen. During execution, the robot tracks its goals with its gaze. It monitors if an object has slipped out of the gripper all the time in order to interrupt and continue when the object is given back. We used this application to verify and test our implementation and to demonstrate how to use PyCRAM and how it eases tasks. Besides the macro `pursue`, which in its implementation is nearly identical to `par` and of course has also been tested during the development of PyCRAM, all other expressions and concepts that our library has to offer have been used in our application.

5.1 Discussion

In the absence of higher programming languages for autonomous robots, PyCRAM fills the gap. With the help of PyCRAM it is possible to reduce the length of code needed for tasks such as picking and placing as presented in our demo application. We have shown how easy it is to write a concurrent control program with PyCRAM for such tasks and how to handle errors and work with retrying constructs.

Our research question was to find out whether it is possible to implement a domain specific language like CRAM in a more widely accepted programming language like Python. What makes Lisp very interesting for domain specific languages is its strong macro system, which other languages like C/C++ do not have, or at least they are not as powerful. One of the main reasons why we have chosen Python is that it is still one of the most popular programming languages for so many years now and it is also widely used by the robotics community. Python itself does not support macros at all, which made implementing a domain specific language not an easy task. A solution could have been to extend Python and develop a custom interpreter but it was our goal for PyCRAM to work with the official supported interpreters and we did not want it to look like a language standing for its own but like a language on top of another one to adopt its popularity. After lot of researching in order to find a way to use macros in Python and after trying quite a few libraries, we luckily found MacroPy, a Python library which adds the ability to code macros in Python 3 in a similar way to how it is done in Lisp.

Compared to macro definition in CRAM, macro-based implementation in PyCRAM looks more complex. It is not that easy to work with macros in Python, especially because there is no official support and thus no official documentation for it. Although there is a documentation for MacroPy, it is just a library and has its limits. The documentation was not always very clear and the library was not free of bugs. We actually ran into some troubles, had to trace it down and fork MacroPy to apply a fix before we integrated it into PyCRAM. But in the end it worked, so we can say it is possible to implement domain specific languages in Python as well.

While implementing a macro is not that easy, using it is much easier. For PyCRAM we only used block macros which feel very native using the `with ... as ...` construct. The only, but not so tragic, disadvantage is that one has to run the program from a bootstrap file which includes the program's main module because otherwise it would not go through the import hook and macros could not be replaced by their definition. Other concepts of PyCRAM like fluents, designators and process modules can be used directly from the console though.

During development we often tried to adopt how things are implemented in CRAM, so that we can later easily compare PyCRAM to the Lisp implementation. Sometimes, however, we implemented concepts in our own way because it seemed easier or because something felt like it is too much for our purpose, and because we wanted to stick to Python conventions. When implementing a language on top of Python it does not make sense to adopt something that feels wrong to a Python programmer.

In CRAM there is subclasses for value fluents and pulsed fluents but we decided to have only one class here. The only difference is that one can define a behavior to handle missed pulses which is only taken into consideration by the `whenever` macro. So what we did instead is to define a variable `_handle_missed` and give it a default value which simply gets overridden when a pulsed fluent is created by calling the `pulsed` function. Also, we did not define a subclass for fluent networks as they are just fluents themselves and we did not implement a function `funcall` to define own operators but instead one can simply pass a function, which gets evaluated when calling the `get_value` function, as value to the constructor.

The language expressions in PyCRAM and CRAM are the same but they differ in how they behave. It was not possible for us to make macros return a value but we were able to store it into a variable passed after the `as` statement which is pretty much the same as a return value. Unlike in Lisp, we cannot and do not want to interrupt threads in Python and hence the expressions for parallel execution do not do that in PyCRAM to stick to conventions that Python programmers are familiar with. This is also why there is only two possible return values for our expressions, these being `SUCCEEDED` and `FAILED`.

Designators are implemented in PyCRAM very similar to how they are implemented in CRAM, there is no big difference here. When it comes to resolution, however, we have added the two functions `check_constraints` and `make_dictionary` for that purpose, while in CRAM there is an approach very similar to Prolog interpretation. It would have been possible in Python, indeed there is a Python library to build a bridge between Python and Prolog, but we just did not need it to answer the research question nor did we need it for our demo application and we think that our approach is easier to understand for those who have never worked with Prolog before.

Process modules were only implemented for the purpose of abstracting from the robot's hardware and hence are kept very simple in PyCRAM. We have only adopted the functions `execute` and `perform` from the tutorials[1] from the official CRAM website as these were all that we needed.

All in all we have accomplished our goal to translate CRAM into Python and can answer the research question with it being possible to implement a domain specific language similar to CRAM in another language. It is not exactly the same but it does fulfill the same purpose. Since our implementation is only a small part of CRAM, there is still a lot of potential here though.

[1]`http://www.cram-system.org/tutorials`. Accessed 01/22/2019.

References

30,000ft Overview – MacroPy3 1.1.0 documentation. https://macropy3.readthedocs.io/en/latest/overview.html. Accessed 01/14/2019.

actionlib - ROS Wiki. http://wiki.ros.org/actionlib. Accessed 01/16/2019.

Barski, Conrad (2011). *Land of Lisp: learn to program in Lisp, one game at a time!* No starch press.

Dijkstra, Edsger W (2001). “Solution of a problem in concurrent programming control”. In: *Pioneers and Their Contributions to Software Engineering*. Springer, pp. 289–294.

Frederickson, Ben (2018). *Ranking Programming Languages by GitHub Users*. https://www.benfrederickson.com/ranking-programming-languages-by-github-users/. Accessed 01/14/2019.

Haoyi, Li (2013). *GitHub - lihaoyi/macropy: Macros in Python: quasiquotes, case classes, LINQ and more!* https://github.com/lihaoyi/macropy. Accessed 01/14/2019.

Hardware Specs | Willow Garage. http://www.willowgarage.com/pages/pr2/specs. Accessed 01/16/2019.

Hoare, Charles Antony Richard (1974). “Monitors: An operating system structuring concept”. In: *The origin of concurrent programming*. Springer, pp. 272–294.

Join the Community | Willow Garage. http://www.willowgarage.com/pages/pr2/pr2-community. Accessed 01/16/2019.

Lemaignan, Séverin, Anahita Hosseini, and Pierre Dillenbourg (2015). “PYROBOTS, a toolset for robot executive control”. In: *Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on*. IEEE, pp. 2848–2853.

Light, CM and PH Chappell (2000). “Development of a lightweight and adaptable multiple-axis hand prosthesis”. In: *Medical engineering & physics* 22.10, pp. 679–684.

McDermott, Drew (1991). *A reactive plan language*. Tech. rep. Citeseer.

Misra, Dipendra K et al. (2016). “Tell me dave: Context-sensitive grounding of natural language to manipulation instructions”. In: *The International Journal of Robotics Research* 35.1-3, pp. 281–300.

Mösenlechner, Lorenz (2016). “The Cognitive Robot Abstract Machine”. PhD thesis. Technische Universität München.

Nguyen, Hai et al. (2013). “Ros commander (rosco): Behavior creation for home robots”. In: *Robotics and Automation (ICRA), 2013 IEEE International Conference on*. IEEE, pp. 467–474.

Overview | Willow Garage. http://www.willowgarage.com/pages/pr2/overview. Accessed 01/16/2019.

Quigley, Morgan et al. (2009). “ROS: an open-source Robot Operating System”. In: *ICRA workshop on open source software*. Vol. 3. 3.2. Kobe, Japan, p. 5.

ROS.org | Is ROS For Me? http://www.ros.org/is-ros-for-me/. Accessed 01/16/2019.

rospy - ROS Wiki. http://wiki.ros.org/rospy. Accessed 01/16/2019.

Sanner, Michel F et al. (1999). “Python: a programming language for software integration and development”. In: *J Mol Graph Model* 17.1, pp. 57–61.

Thielscher, Michael (1998). *Introduction to the fluent calculus*. Citeseer.

Weinstein, Gregory S et al. (2007). “Transoral robotic surgery: supraglottic partial laryngectomy”. In: *Annals of Otology, Rhinology & Laryngology* 116.1, pp. 19–23.