Universität Bremen

Fachbereich 3: Mathematik und Informatik

Master's Thesis

PyCRAM - Enabling Robot Behavior Adaptation through Introspection, Prospection-based Reasoning and Action Reordering using Task Trees

PyCRAM - Ermöglichen von Roboterverhaltens-Anpassungen durch Introspektion, Projektions-basiertem Schlussfolgern und Aktions-Reorganisation mittels Task Trees

Christopher Pollok

Matriculation No. 400428

December 29, 2020

First Examiner:	Prof. Michael Beetz PhD
Second Examiner:	Dr. Rene Weller
Advisor:	Gayane Kazhoyan

Christopher Pollok

PyCRAM - Enabling Robot Behavior Adaptation through Introspection, Prospection-based Reasoning and Action Reordering using Task Trees

PyCRAM - Ermöglichen von Roboterverhaltens-Anpassungen durch Introspektion, Projektions-basiertem Schlussfolgern und Aktions-Reorganisation mittels Task Trees

Master’s Thesis, Fachbereich 3: Mathematik und Informatik

Universität Bremen, December 2020

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sämtliche wissentlich verwendeten Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet.

Bremen, den 29.12.2020

Christopher Pollok

Abstract

Robots deployed in households will need to adapt to changes in these dynamic environments. To this end, a robot ideally is able to inspect its past actions, simulate future actions without risking manipulating its physical environment, and reorganize its plan's actions. PyCRAM is a robot control framework, capable of doing generic and flexible planning. In this thesis, I present an extension to PyCRAM in the form of Task Trees to enable the aforementioned capabilities.

Task Trees are a representation of performed actions in a plan. A tree and its nodes can be inspected, modified and rearranged. Furthermore, a simulated Task Tree can be used to record actions performed in a Bullet world. Through this, Task Trees enable PyCRAM to provide a basis for behavior adaptation through introspection, prospection-based reasoning and action reorganization. I present an implementation of Task Trees with a convenient interface, that is easy to use in new and existing PyCRAM programs. I validate the implementation to demonstrate its functionality on simple examples and showcase its potential for significant plan optimization in a table setting task.

Contents

Chapter 1

Introduction

1.1 Motivation

Envisioning the robots of the future, people think of them as personal assistants, capable of doing just about anything a human can do in any environment the human needs them in. This includes everyday household activities like cleaning, cooking, or grocery shopping. These tasks and environments are highly dynamic, be it because of the interaction with human actors or changing scenery when getting into a new household. Robots need to be able to adapt to these environments autonomously.

Consider a robot that needs to set a table for breakfast. Assume that the robot has a predefined action plan for doing this kind of thing. Typically, this involves bringing milk, cereal, a bowl, and a spoon to the table. When the robot's plan is implemented, the programmer does not know what kinds of bowls, spoons, milk, or cereal the human owner will have in their kitchen. So the plan needs to be very generic. But even the best generic plan will not be able to account for every detail a robot might face in a new environment. Additionally, when something in the environment changes the robot also needs to adapt to these changes.

Let us assume that, typically, the cereal boxes of the user's favorite brand are best grasped with one gripper from the top. One day, there might be a new brand of cereal, which is much better suited to be grasped from the side or needs a different amount of grip force. These parameters are very important when interacting with the environment. Choosing them is not a trivial task. Ideally the robot could learn from past experiences to adapt over time. In this environment the robot could try out different sets of motion parameters in order to find the ones that generate collision-free motions.

When getting something out of the fridge or setting the objects on the table, the robot needs to be careful not to collide with the environment or knock something over. For this it situation, a prospection capability would be very useful for the robot, such that it could imagine performing its actions in a virtual simulation environment.

Consider the robot having the ability to imagine its actions in a virtual simulation. So it can try out different sets of parameters in order to find ones that don't cause any collisions.

Working alongside humans in a dynamic environment can lead to structural changes in said environment. For example, normally it might be a good idea to get the bowl and spoon at the same time, since they are located closely together in the kitchen. However, if the human reorganizes their kitchen this might not be true anymore. So the robot again needs to be able to adapt. This kind of adaptation might be done before starting the robot by providing the new structure of the kitchen in the form of background knowledge for the robot. Additionally it would be advantageous, if the robot could react to these problems at runtime and autonomously reorganize its actions to reflect the changes.

CRAM is a robot planning framework, that, among other things, provides the CRAM Plan Language (CPL). CPL is a high-level robot programming language enabling robots to adapt to dynamic environments. It employs underspecified descriptions of actions, objects and locations, called *Designators*. With these it provides a way to program robot plans that can be used not only in a single static environment, but be made general so they can be executed in a variety of dynamic environments. This is achieved by grounding the underspecified Designators at runtime, using environment-specific knowledge and on-demand resolution. Additionally, CRAM uses the concept of Task Trees to record the robot's actions during execution, in order to to inspect and reason about them both during and after execution. The recorded actions are represented by the Task Tree's nodes.

This enables the robot to find the most suited parameters for any previously executed action by introspection. When combining the use of Task Trees with a lightweight simulator like the BulletWorld [10], the robot is able to imagine its actions in the simulation and extract the best parameters from those simulations for execution in the real world. Finally, Task Trees can be altered by reorganizing actions, thus adapting the robot's behavior. For example, if the robot is tasked with retrieving two objects, the initial plan might be to drive to the first object's location, grab it and return it to the delivery point. Then to start the same process with the second object, even though the two objects might even be at the same location. In this case the robot's actions can be reorganized, so that it only drives once to the objects' location, picks up both of them and the returns to the delivery point.

These capabilities address each of the aforementioned problems. When encountering a new brand of cereal, the first attempt at grasping might fail, but future attempts can utilize introspection to determine the correct grasping parameters before failing. Furthermore, inspecting Task Trees of simulated actions allows to extract the best parameters before executing the action in the real world. Being able to reorganize the Task Tree's structure and re-executing it with the changes enables the robot to encounter structural problems in its plan and overcome them by reordering actions or adding new ones.

In order to make the CRAM framework and CPL, which are implemented in Common Lisp, more accessible, efforts are being made to reimplement them in Python under the name PyCRAM. In this thesis, I set out to further develop PyCRAM and extend it with the capabilities for using Designators to describe actions and Task Trees to record performed actions. The original Task Tree implementation in CRAM did not include ways to simply change a Task Tree and re-execute it. A Task Tree was strongly coupled to the plan's source code, which placed some constraints on the tree's structure and capabilities. With this new implementation I contribute ways to modify and execute Task Trees more freely.

1.2 Problem

In this thesis, I set out to design and develop a method to use Task Trees as a part of the PyCRAM framework.
To make CRAM more accessible it needs to be available in a programming language that is more accessible. To make PyCRAM a viable alternative to other robot control frameworks and CRAM itself, it needs to support the most powerful tools CRAM has to offer and make them available in a way that is easy to use. One of these tools are Task Trees and the capabilities they bring with them.
The three problems I focus on in this thesis are as follows:

(1) **Introspection**
The ability to retroactively inspect one's own actions is important for multiple reasons. In the case of robots in household environments it is almost certain that failures will occur. Even though they do not necessarily need to be catastrophic, it will always be preferable to prevent failures in the future. So giving robots the ability to introspect is beneficial.

(2) **Prospection-based Reasoning**
Manipulating a physical environment is an inherently risky scenario. Robots in these environments need ways to minimize risks. A very intuitive way to go about failure handling in general is trial and error. In principal it needs no additional implementation when it comes to the plans a robot needs to execute. However, in a real physical environment unrestricted trials can quickly lead to catastrophic outcomes. So instead of the real world, simulation software is used to simulate the physical environment and let the robot perform trial runs of its plans without risk. Such simulated executions are already possible with the current implementation of PyCRAM. However, it lacks a convenient way to do so and reason about the executed actions retrospectively.

(3) **Action Reorganization**
Being deployed in dynamic environments, robots need the ability to adapt to changes in said environment. Generic plans can explode in complexity if developers try to account for

every possible scenario. Robot control systems need a mechanism through which they can alter their plans to adjust to the changing requirements of a dynamic environment.

Task Trees are a promising solution to these problems. The nodes being a representation of the executed actions of the robot already provides the ability to analyze those actions and learn from them. Coupling the already present integration of the Bullet simulation engine and the recording aspect of Task Trees provides a convenient way to simulate a robot's actions and reason about the results of those simulations. Lastly the Task Tree's modifiability and subsequent re-execution make it possible to reorganize actions in order to adapt plans in dynamic contexts.

I do not aim to outright solve these problems in a all-encompassing fashion (especially problem (2)). Rather, my work is intended to give developers a basis on which to build more specialized solutions to these problems. A more thorough discussion on possible extension is given in section 6.3.

Additionally, from the usability viewpoint, I aim to create a Task Tree structure, in which changes to it do not impact the usability of its interface. Re-executing a tree or simple changes like adding, moving or removing actions from a plan should ideally be possible by just having a reference to that action's node in the tree and calling some function on the object.

1.3 Contribution

With this thesis I present an approach to generating a task logging structure that is suitable for both reasoning about and re-execution. As a practical contribution, I implement an extension to the PyCRAM planning framework. This extension contains an implementation of the presented approach as well as a library of functions to conveniently use the system for introspection, prospection-based reasoning and action reorganization of robot plans in a simulated environment. I demonstrate its utility in those areas by showcasing examples.

1.4 Thesis Structure

This section provides a brief overview of this thesis' contents. Generally, this thesis is intended to be read sequentially.

Chapter 2 reviews existing works on robot action logging and plan optimization.

Chapter 3 introduces the concept of Action Designators and Task Trees and explains them in the context of the original CRAM framework.

Chapter 4 presents the implemented extension of PyCRAM. It provides an introduction from a user's perspective and also offers insights for developers.

Chapter 5 explores three use cases for Task Trees in general and showcases the utility of the implemented system in example scenarios. It concludes with a showcase of all the system's features used on a single plan for optimization.

Chapter 6 reviews the implemented system by reflecting on the results of the previous chapter, and draws a conclusion regarding the initial problem description.

Chapter 2

Related Work

In this chapter, I give a brief overview of other works in the realm of robot behavior optimization and action logging for analysis. Some of the mentioned works describe parts of larger plan control systems. For the sake of brevity, I will not describe all of these systems in detail and rather focus on the part of interest in the context of this thesis.

Plan Repair in CHEF

Hammond [9] presents a plan repair system as part of CHEF, a case-based planner for the domain of Szechwan cooking. CHEF employs an episodic memory of recipes for dishes it can generate plans for. When a new dish is requested, this library of recipes is used to generate combinations of plans to achieve the desired result (the requested dish) or to get as close as possible. However, not all plans can just be combined like this and would lead to failures. CHEF has the ability to detect and explain those failures causally. For this, it employs a deep domain model describing failures and the steps causing them. To fix faulty plans, the system uses a set of repair rules indexed by the causal explanations of failures they fix. It also makes sure that the repair strategies it executes do not interfere with the goal that was originally intended when the failure occurred. Finally, CHEF does not need to choose a single repair strategy based on a priori criteria, but can simulate multiple strategies in parallel and choose the best result afterwards.
Repair strategies are indexed by structures called ”planning TOPs”, based on the idea of understanding TOPs first suggested by Schank [18]. These are memory structures consisting of the description of a planning problem and the set of repair strategies able to solve the problem. It is important to note, that CHEF's TOPS are not only defined by the steps that led to the problem, but also the original goal. One example of such a TOP is SIDE-EFFECT:BLOCKED-PRECONDITION, describing a situation, where the side effect of one step breaks a precondition to a later step. Stored under this TOP are different general repair strategies. Some examples are ALTER-PLAN:PRECONDITION, describing the strategy to

find a way where the precondition is not needed anymore, or ALTER-PLAN:SIDE-EFFECT, where the plan is changed to not cause the side effect leading to the problem in the first place. CHEF uses a total of 16 TOPs to index two to six repair strategies each. There is overlap between the sets of repair strategies for different TOPs, if their corresponding problems are similar.
To detect failures CHEF uses a simulator capable of running completed plans. Through this it can recognize failures causing the simulation to stop because a step could not be completed, or those caused by a mismatch between the end state and the expected outcome. It goes on to construct a causal explanation of the failure through back chaining the causal links created during the simulated plan execution. A discrimination network is used to index the planning TOPs by the explanation features. Repair strategies define a general framework, which can be filled with specific information of a given problem scenario to turn it into a specific change. Once a TOP has been chosen, its strategies are all tested and the best working repair is selected based on a set of heuristics.

Autonomous Behavior Adaptation with UM-PRS

Lee et al. [12] present a C++ implementation of a Procedural Reasoning System (PRS) they call UM-PRS. They chose to implement their robot control program for military reconnaissance scenario in a PRS framework, because of its integration of goal-directed reasoning and reactive behavior. Their system takes a high-level mission plan and an annotated map of the deployment area as input. The annotations instruct the robot to do specific tasks at specific locations, while the mission plan describes the robot's major goals declaratively. However, the map is incomplete in the sense, that it lacks detail needed to safely maneuver unpredictable or dynamic environments. The UM-PRS consists of a World Model database, a set of goals, a set of plans called Knowledge Areas (KAs) and an intention structure, acting as the run-time stack of the system. KAs have a body consisting of actions, either primitive or representing subgoals. An interpreter controls the execution of the system. It monitors the beliefs in the World Model and selects the KAs to put on the intention structure. The interpreter executes the currently active KA's actions one by one. If an action changes the World Model or the set of goals, the KAs are reevaluated and a new KA may be chosen. This enables the system to react to changes in the environment or switch to more important goals, when the need arises. The PRS by Lee et al. is capable of autonomously adapting to a dynamic environment. However, for every situation that could arise, a corresponding procedure needs to be implemented.

Reactive Action Packages

The original Reactive Action Packages (RAPs) planner by Firby [5] was purely reactive, as it made no predictions about the future state of the system and did no long-term planning. A RAP is an autonomous process pursuing a planning goal until it has been achieved. If the planner has multiple goals, there are multiple RAPs each pursuing one of them. Each RAP only acts on the current believed world state, hence the planner being purely reactive. The planner described by Firby employs a RAP interpreter managing a RAP execution queue. Each RAP holds a number of methods for achieving its goal. The interpreter selects one of the methods based on the system's current belief state. When chosen by the interpreter, the RAP will first check for goal completion and in the negative case choose one of those methods to perform in turn. A method consists of a network of tasks, called a task net, which are either primitive commands to perform on the robot, or a new sub-goal invoking another RAP. The RAP is placed back on the execution queue until its chosen method concludes or fails. If no method can be chosen based on the current world model, the RAP is deemed failed.
Behavior adaptation in this original version was implicitly integrated in the choosing of appropriate task nets when activating a RAP. However, since the RAPs could only ever act on the current state, no optimizations could be made. This includes RAPs interfering with each other's progress, so that the overall execution would be hindered because the RAPs did not work together.
The system was later extended to include sensor and memory mechanisms [8], to interface with a behavior-based control system [6] and to allow for simultaneous execution of subtasks [7].

Action Logging and Replay in PYROBOTS

PYROBOTS [13] is a python robot control software toolkit. It presents itself as a lightweight embedded domain-specific language (not unlike CRAM and PyCRAM). Core concepts include a robot instance as an encapsulation of its low-level controllers and state, asynchronous actions, monitored events and resource management (e.g. actuators). Actions are hierarchical and can be canceled, which leads to a possible recovery action. Events can be monitored to invoke callbacks once or always. Finally, PYROBOTS provides a resource locking mechanism to ensure exclusive usage of sensors and actuators for actions.
PYROBOTS provides logging mechanisms, even though only for debugging purposes and not for autonomous behavior adaptation or optimization. Actions and resource usage are logged over time and can be inspected at runtime through console output or as a file afterwards. Additionally, the generated logs allow offline replay.

Behavior Adaptation by Trial and Error

Outside the realm of plan-based optimization Cully et al. [3] have created a system enabling robots to adjust their behavior through trial-and-error in case of damage. To achieve this their system creates a map of the behavior-performance space by simulating a vast amount of behaviors and predicting their respective performance. This step has to be done only once for each robot design and allows for rapid adaptation afterwards. Designers only need to describe the behavior (e.g. how much to use each limb when walking) and performance (e.g. forward speed) dimensions, the map can then be automatically generated. When a robot needs to adjust its behavior, it starts with low confidence values for each point in the map. While testing out different behaviors it rates their actual performance and updates its confidence in the performance value. Neighboring behaviors are also affected by this update. The process ends, when the robot finds that it has sufficiently tested behaviors and chooses the one with the best performance.
Cully et al. call this approach "Intelligent Trial and Error". They present two novel algorithms for initially creating the map (MAP-Elites [16]) and the adaptation step (M-BOA).
They show in experiments, that their algorithms allow robots to forego faulty behaviors, that don't work anymore because of differing types of damage, in favor of behaviors that do achieve the intended results. Aside from damage recovery, their experiments with an undamaged robot also showed, that it was able to find walking gaits even faster than their reference gait. Showing that it could also be used for automatic behavior production.
Cully et al. present a very powerful method for robots to adapt to changes in their own anatomy and/or their environment.

Past Efforts in CRAM

CRAM [14] early on included the capability to reflect on itself in order to answer questions about interal processes and planned actions. It always utilized a Task Tree structure to record and hold actions performed during plan execution [15, 2]. A more detailed description of the original Task Tree implementation is given in chapter 3.
To benefit from previous experiences CRAM was later extended to be able to record and query episodic memories[20, 19]. The Task Tree was just one part of these memories, beside recorded sensor data and the system's belief state over time.
Rule-based Task Tree transformation for plan adaptation was first implemented in simulation by Müller et al. [17] and later used by Kazhoyan et al. [11] on a real robot. Kazhoyan and Beetz [10] also presented a method to project a robot's actions through simulation during plan execution.
Recently, work has begun to port the CRAM framework from Common Lisp to Python in

the form of PyCRAM [1]. Originally only including the core CPL constructs and support for Designators and Process Modules, it was later extended for use with the Bullet simulation engine [4]. In this thesis, I extend PyCRAM with an implementation of Action Designators, Task Trees and the capabilities for introspection, prospection and action reorganization.

Chapter 3

Foundations

The CRAM framework (written in Common Lisp) already provides an implementation of Task Trees. In order to differentiate my new approach from the original, this section will outline how Task Trees were implemented in CRAM and what their limitations are. Additionally, a quick overview of the state of PyCRAM before this thesis' extension is given to distinguish my contribution from the already existing framework.

3.1 CRAM Task Trees

CRAM uses a structure called `Task` to represent a plan function. Tasks can be executed and are stored in `TaskTreeNode`s, which make up the CRAM Task Tree. Being implemented in Lisp, CRAM can and does make extensive use of macros to provide its Task Tree functionality. In Lisp the normal way to define a function is by using the `defun` macro (similar to Python's `def` keyword). CRAM uses its own special macro to define plan functions (`def-cram-function`, or `def-top-level-cram-function` for the top-most plan). The macros wrap the defined function in a so-called replaceable function, that allows for later modification of the Task Tree. A new Task Tree is only created in the top-level variant, while the other variant will not be able to be executed when no Task Tree has been created at an earlier point. Replaceable functions serve as a kind of container for the actual function. They don't really get executed, instead, when they are reached a `Task` with the actual function is created on the fly and gets executed right away.
The Task Tree is created when the code is executed for the first time (during runtime). During execution of the code, the `TaskTreeNode`s are created and put into a hierarchy. After initial creation, the Task Tree's structure is fixed. Nodes can not be added, moved or deleted. It is however possible to alter node's contents. When nodes are executed, actually the task stored in them is executed (or rather the replaceable function holding the task). This enables code replacements by changing the content of the nodes. If a node is not supposed to be executed anymore, one can also remove its content, so that no code is executed when it is reached. Even

though new nodes cannot be added, the replacement code can be arbitrarily complex, so new tasks can be added. But the Task Tree is not updated to reflect the new task hierarchy. So when replacing with code that would normally generate `TaskTreeNode`s, they would not be created. Of course this replacement mechanism also enables the moving or switching of tasks from one node to another.
After a Task Tree for a specific top-level plan has been created (identified by its name), execution of that plan will execute the Task Tree instead. It is not possible to execute the Task Tree directly.

3.2 PyCRAM

PyCRAM does not yet present a full port of the original CRAM to Python. It however already implements many core features and concepts of CRAM like Fluents, Designators, Process Modules, the CRAM plan language and simulation with Bullet. A quick overview of the features used or built upon in this thesis is given here.

3.2.1 Designators

Designators are (initially) underspecified descriptions of motions, objects, locations or actions. They can be used to describe these things in a way that does not require knowledge of every detail that is required for execution of corresponding plans. In preparing to execute the plan's actions the Designators can then be resolved to hold more concrete data by different systems for the different types of Designators, such as semi-random sampling for locations or querying a knowledge base for grasping information of objects.
PyCRAM had an implementation of the base Designator class and one class inheriting from it for Motion Designators.
For this thesis, I needed to implement Action, Object and Location Designators. I used the base Designator class for Object and Location Designators. But for Action Designators, I did not inherit from the base class, because its resolution system did not fit the purposes of my Action Designators.
In CRAM all Designator types are resolved through calling the `reference` function and using Prolog queries in the middle layer, but each type has it's own mechanisms for querying further knowledge and controlling flow on the low-level.

3.2.2 Process Modules

Process Modules provide the execution system for Motion Designators in CRAM and PyCRAM. They are robot-specific modules of the overall planning system, responsible for activating the specific robot's motor functions to execute the described motion. Resolution is actually very minimal since Motion Designators most often contain all necessary data to execute the motion. Process modules are meant to be implemented on a per-robot basis and be loaded as necessary for a specific robot.
This thesis does not expand upon this functionality, but uses process modules to call the Bullet simulators functions to move the robot and simulate visual perception.

3.2.3 Bullet Simulator Integration

The Bullet simulation engine is used to simulate the robot's behavior in a no-risk environment. This can be used during development for rapid testing of new plans. In production code it lets the robot simulate its actions before attempting them in the real world, thus minimizing the risk of failure. Bullet employs a world state mechanism that allows for easy saving and resetting of the simulated environment. The integration of Bullet into PyCRAM consists of an API implementation to interact with the inner workings of Bullet, corresponding process modules and additional features on top of that like attachment handling and reasoning mechanisms.

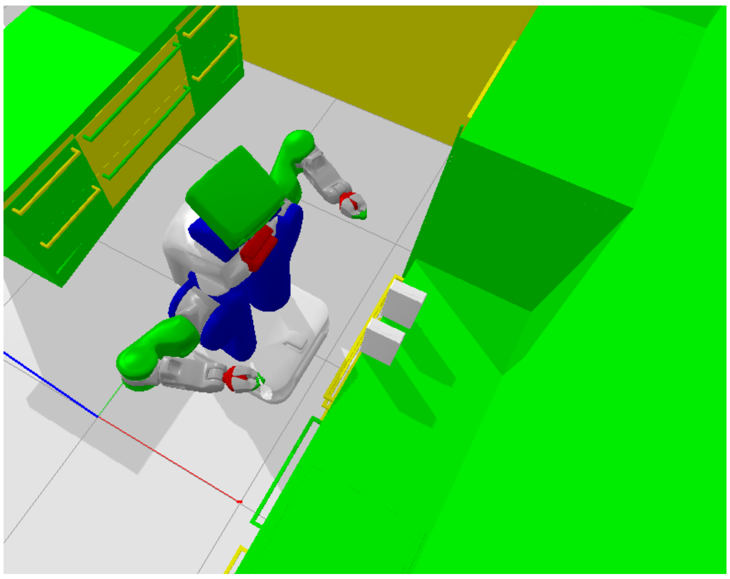


Figure 3.1 Screenshot of the PyBullet world.

Chapter 4

Methods

4.1 PyCRAM Task Trees from a User's Perspective

In this chapter I highlight the additions made to the PyCRAM framework in the form of Task Trees and action designators.

4.1.1 Action Designators

The first addition I made to the existing framework were action designators. PyCRAM already provides a base designator class, but the action designators in my implementation are not dependent on that. As mentioned above, the action designators use a separate resolution system and otherwise there was no need to use the features of the base designator class.
First, I will describe the control flow of action designators. Action designators are represented by the `ActionDesignator` class, with each instance holding an `ActionDesignatorDescription` object. To execute a plan associated with an Action Designator the user has to create a new `ActionDesignator`, pass it an `ActionDesignatorDescription` and then perform the `ActionDesignator`. Like this:

```
ActionDesignator(TransportObjectDescription(...)).perform()
```

The .perform() will also call the `ground` function of the `ActionDesignator`'s `ActionDesignatorDescription` and eventually call the grounded function. The description's `ground` is implemented in a Designator resolution module that has to be imported beforehand to take effect. Inside the module, the description's parameters are used to determine what plan function the action description resolves to. Missing parameters will be inferred by different means, depending on the methods available in each case (this is described in more detail in section 4.1.6). The inferred parameters are stored in the description and the plan function with its parameters is wrapped into an anonymous function, to be called when the Designator is performed.

Object and Location Designators

Unlike actions, Object and Location Designators were implemented as simple sub-classes of the base Designator class. This was done to differentiate them from Motion Designators. At this time there is no sophisticated resolution system for these types of designators in place. However, the plans that were implemented as part of this thesis make heavy use of them and use simple generator functions, e.g. to provide navigation goals based on Object Designator type properties.

Plans

Plans in the context of PyCRAM are functions that use Action Designators to perform actions, i.e. call other plans. Through this plans can form a hierarchy, where higher plans can be very abstract, e.g. ”set a table”, while lower plans are very specific, e.g. ”open gripper”. This hierarchy allows code of low-level plans to be reused over many different contexts.

4.1.2 The Idea of Task Trees

A Task Tree is a representation of actions performed in the execution of a plan. While the plan is executed, the performed actions are recorded and stored in a tree structure. This is done for multiple reasons.

1. After the execution is done (or even possibly during it) the recorded actions can be inspected. This again can have different purposes, for example looking up parameters for an action that was already performed once or just seeing if an action was successful in the past.
2. Task Trees of simulated actions can be created in order to retrieve valuable information from them, similarly to how the Task Trees of real-world execution would be used
3. Task Trees can be re-executed, which saves the robot the work of inferring action parameter values at runtime.
4. Task Trees can be altered before re-execution by changing parameters or reordering actions. This can lead to better performance.

4.1.3 Understanding the Task Tree's Structure

Task trees consist of `TaskTreeNode` objects, just called *nodes* from now on. A node can have a single parent and multiple children, all of these are also nodes. Nodes without parents are roots and represent the highest abstraction of a plan represented by a Task Tree. A single tree has only one root. A node's children are stored in a list, with their sequence representing

their execution order. Additionally each node is root to it's own separate (initially empty) tree of "execution siblings". Practically, each node has a slot for one previous and one next execution sibling. These can hold another node, which of course has the same slots and thus a tree of execution siblings is created. When re-executing a Task Tree, the previous execution sibling is executed before the node itself, while the next execution sibling is executed after. This results in a node's execution sibling tree being executed in IN-order.
Nodes also have a path, a status, start and end time as well as a `Code` object. A `Code` object is a representation of a function along with it's arguments. Nodes store this information to call the appropriate functions with the right arguments during execution. A path is a string of the node's parent's path joined with the node's code's function name, followed by the index of that function in the context of it's parent, separated by a slash (/). The root node's path is just its code's function name. Examples of these follow in the next section. The node's status can be either CREATED, RUNNING, SUCCEEDED or FAILED. The status can be used for introspection purposes later. Figure 4.1 depicts a simplified class diagram of `TaskTreeNode` and its related classes. In figure 4.2 a simplified notation is used to show a `TaskTreeNode`'s structure.

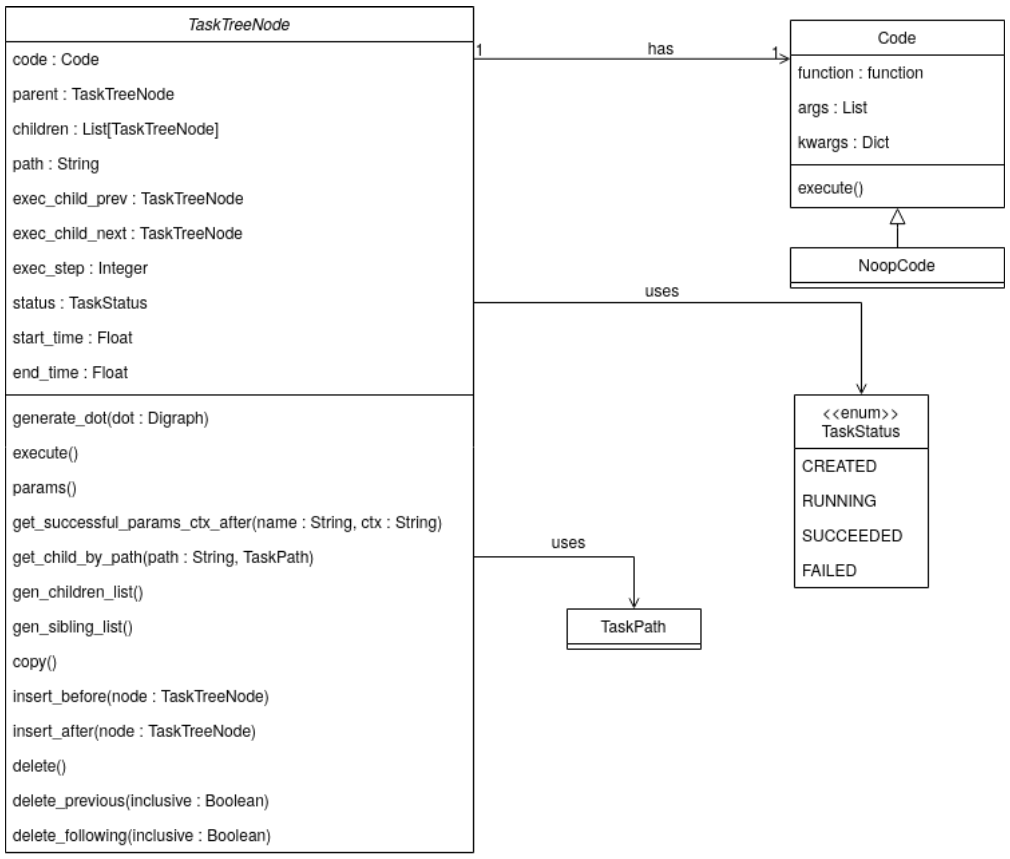

TaskTreeNode
code : Code
parent : TaskTreeNode
children : List[TaskTreeNode]
path : String
exec_child_prev : TaskTreeNode
exec_child_next : TaskTreeNode
exec_step : Integer
status : TaskStatus
start_time : Float
end_time : Float
generate_dot(dot : Digraph)
execute()
params()
get_successful_params_ctx_after(name : String, ctx : String)
get_child_by_path(path : String, TaskPath)
gen_children_list()
gen_sibling_list()
copy()
insert_before(node : TaskTreeNode)
insert_after(node : TaskTreeNode)
delete()
delete_previous(inclusive : Boolean)
delete_following(inclusive : Boolean)
1
has
1
Code
function : function
args : List
kwargs : Dict
execute()
NoopCode
uses
<<enum>>
TaskStatus
CREATED
RUNNING
SUCCEEDED
FAILED
uses
TaskPath

Figure 4.1 Class Diagram: TaskTreeNode

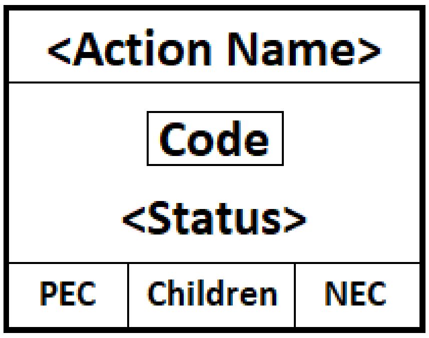


Figure 4.2 Simplified structure of a `TaskTreeNode`.
The node is depicted with a name, a stand-in for it's `Code` object and slots for its previous and next execution siblings (PEC and NEC) and its list of children.

4.1.4 Using the Task Tree's Capabilities

4.1.4.1 `TaskPaths`

First, I will to introduce the `TaskPath` class. In order to easily retrieve specific nodes from the Task Tree, each node has a unique path assigned to it. Basically, a `TaskPath` object is a wrapper around a path string, that ensures they are well formed. They are used to retrieve arbitrary nodes from a Task Tree based on it's path. For this `TaskPath` objects can be either used directly or just be relied on working in the background, when using simple strings as parameters instead.
Generally, a `TaskPath` will be used to describe paths from the Task Tree's root node. But it can be used from any node in the tree. The most basic path is a single node's function name. That would be the first occurrence of a node with the named function in the first layer under the root. Adding an index can be used to retrieve a later occurrence. This only works if there are enough instances of the named function. To access deeper layers of the tree another function name (and index) can be appended.
The structure of `TaskPath`s in Backus-Naur form is as follows:
<**TaskPath**> ::= [<**TaskPath**>/]<**FunctionName**>[.<**Index**>]

Examples:
If you know information about the node you are trying to access, it is easy to formulate the path for that node. For example, if you want to find the node in the Task Tree that corresponds to the `set_table` action's first `pick_up` subaction, you would describe it like this: "**set_table/pick_up.0**".
Retrieving the second `set_table` action's `transport` subaction's third `navigate` subaction would take this path: "**set_table.1/transport/navigate.3**".

Paths can be arbitrarily complex or simple and contain even just a single action's name: "**transport**".

4.1.4.2 Introspection

Introspection follows a simple pattern: First you have to get some node you are interested in and then inspect its attributes. There are two main ways to retrieve a node. The default way is to use the `get_child_by_path` function of `TaskTreeNode`. It works with either a `TaskPath` object or a string. However, you can also use custom functions that retrieve nodes based on other criteria. Currently there is only one function of this kind: `get_successful_params_-ctx_after`. It takes two function names, the second giving the context for the first one. The goal is to retrieve a node with the first function, after which a node with the second function was successful.
So, for example, find a navigation node, after which a pickup action was successful. The assumption being, that the navigation's parameters led to the success of the pickup action. This would most often be used in conjunction with prospection, which will be explained in the next section.
The inspection of nodes can be done through normal means of accessing attributes on any Python object. Though most interesting will be the `function`, `args` and `kwargs` attributes of a node's `code` object and its `status` attribute.

4.1.4.3 Prospection-based Reasoning

In order to reason about simulated actions, there needs to be a way to simulate actions without impacting the real world (or primary simulation) state and Task Tree. This is handled by the class `SimulatedTaskTree`. It functions as an encapsulation mechanism for performing actions in a simulated environment separate from the main simulation of the robot's belief state and Task Tree. The recommended usage is in a with-statement, like so:

```
with SimulatedTaskTree() as st:
  ... doing something ...
  params = st.get_successful_params_ctx_after(...)
```

Upon entering the with-statement, the currently active Task Tree is suspended and a temporary Task Tree is created. Any actions performed inside the with-block are recorded to that new tree. When the with-block ends, this temporary tree is discarded. This can be used to try out any action and see if it succeeds, and if it does perform it for real afterwards. It can also be used to repeat a single action multiple times with different parameters, extract the successful run's parameters and use it to perform the action with the correct parameters right away.

4.1.4.4 Action Reorganization

After a Task Tree has been recorded it can be re-executed. This alone can already be useful, when dealing with a similar scenario, where a previously executed plan can be reused. However, when dealing with a dynamic scenario or changing challenges, it can be useful to reuse an existing plan, but also be able to tweak it. Action Reorganization can be used to modify an existing Task Tree to meet new requirements. For example, the previously executed plan has the robot collect a bunch of items one by one. Two of those items could easily be fetched at the same time without having a negative impact on the rest of the plan. Situations like this can occur because not everything can be perfectly planned ahead in a generic plan. In those cases a simple reordering of actions for picking up and deleting now obsolete navigation actions can yield a much more efficient plan.
In the following paragraphs I will highlight the different methods of reorganizing or changing a Task Tree. Figure 4.3 shows an example of a Task Tree.

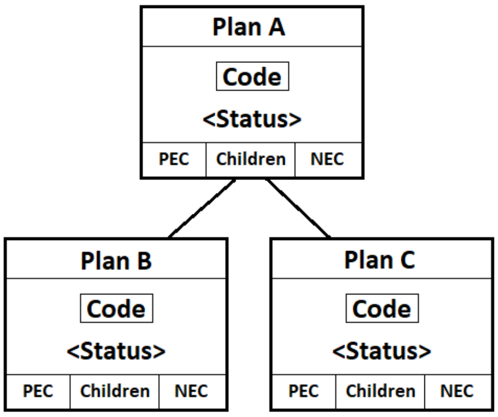


Figure 4.3 Example of a Task Tree using a simplified notation.
Plan B and C are connected to the Children slot of Plan A. There were no modifications done to this tree, so no PEC or NEC slots are filled.

Insertion

The most basic way to modify a Task Tree is to insert new or copied nodes before or after existing ones (inserting non-copied nodes that are already in the tree is technically possible, but will lead to unexpected behavior). The functions `insert_before` and `insert_after` can be used on any node in the Task Tree and will insert the given node as an execution sibling. Figure 4.4 shows two examples of a new node added to figure 4.3's Task Tree.

```
node_copy = tt.get_child_by_path("set_table.1/navigate.3").copy()
insert_point = tt.get_child_by_path("set_table.0/place.1")
insert_point.insert_after(node_copy)
```

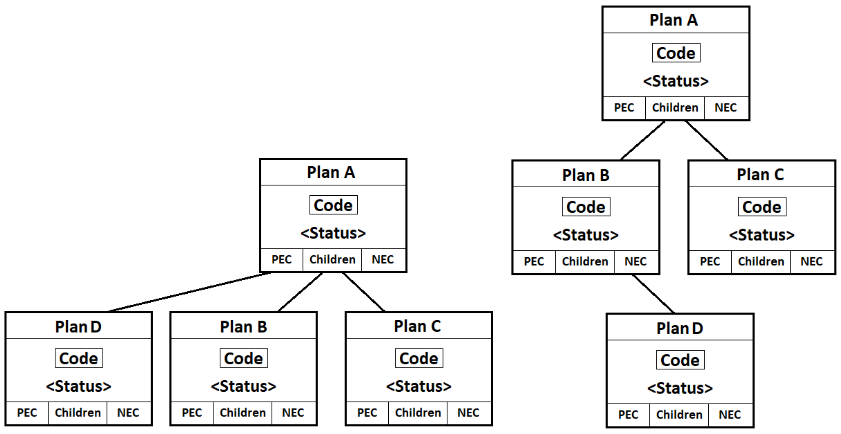


Figure 4.4 Examples of adding a new node to figure 4.3's Task Tree.
The new Plan D node is connected to a PEC or NEC slot because it was added after the initial creation of the Task Tree.

Deletion

When actions become obsolete through optimizations, they should be deleted. This is pretty simple in PyCRAM: Just call the node's `delete` function. A normal node (those created during the first execution) will not actually be deleted, but its code object will be replaced with a `NoopCode` object. Nodes with a `NoopCode` do not execute any function. When a tree is re-executed, all noop-nodes are skipped and when querying nodes through the provided functions, they will not be found. If the deleted node is an execution sibling however, it will be removed from the execution tree and its execution sibling children (if any) will be brought up, maintaining the original execution order. Figure 4.5 shows the two processes side by side.

```
tt.get_child_by_path("set_table.1/navigate.3").delete()
```

There are two additional convenience functions to make mass deletion easier. `delete_previous` and `delete_following` can be called to, as their names suggest, delete all of the previous or following siblings of a node under its parent. An optional parameter can be used to include the node itself in the deletion process.

```
tt.get_child_by_path("set_table.1/navigate.3").delete_following()
```

Moving Nodes

It is also possible to directly move a node to another location inside the tree without having to manually copy and delete it. PyCRAM provides the function `move_node_to_path`, which takes a node, a path and a placement indicator as parameters. The last one being either `-1`, `0` or `1`. `-1` and `1` meaning to place the node before or after the node at the path, respectively. And `0` meaning to replace the node at the given path with the new node.

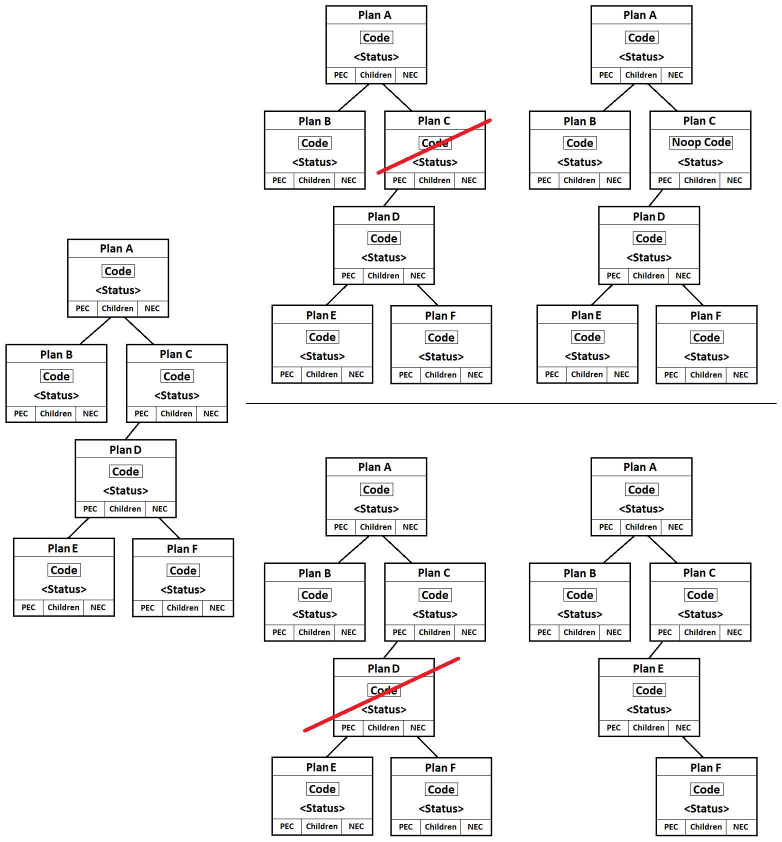


Figure 4.5 Task Tree examples for node deletion.
Top: Deleting a normal node. Bottom: Deleting an execution sibling.
When deleting a normal node, its `Code` object is replaced with a `NoopCode` object.
When deleting an execution sibling, it will be removed from the tree and its children will be brought up.

```
node_to_move = tt.get_child_by_path("set_table.1/navigate.3")
move_node_to_path(node_to_move, "set_table.0/place.1", 1)
```

The moved node is copied and inserted at the destination, while the original node is deleted. So any reference to the old node are invalidated when it is moved. The original node's children and possible modifications made to them will be lost. PyCRAM does not support unexecuted nodes to have children. Because they have been moved, the information gathered, when they were first executed, would not represent the new situation anymore. Figure 4.6 shows a move of a normal node to the previous execution sibling slot of another.

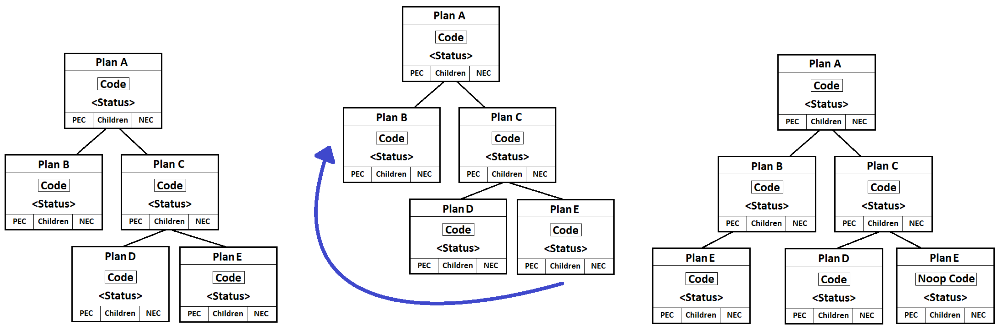


Figure 4.6 Task Tree example for moving a node.
When moving a node, the original node is deleted and a new one is put into a execution sibling slot of the target node.

4.1.5 Visualization

The `TaskTreeNode` class provides a `generate_dot` function, which creates a graphical representation of itself using the `graphviz` Python package. Figure 4.7 shows an example. Each node in the tree represents a node in the corresponding Task Tree. The nodes are labeled with the last section of the node's path, their plan function with its parameters, a status and start and end times. The function is recursive and gets called for each child in the order they are executed. In the end the generated tree is an accurate representation of the original Task Tree.

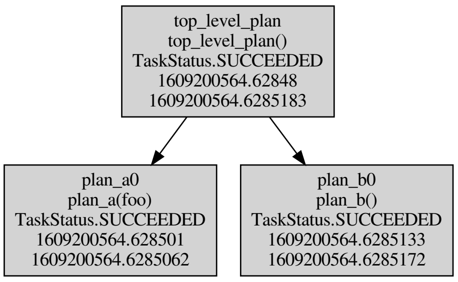


Figure 4.7 `TaskTreeNode` Visualization Example

4.1.6 Understanding the Architecture of PyCRAM-based Plan Control Programs

In this section, I propose an architecture for plan control programs utilizing Action Designators and the features offered by Task Trees. The section does not dive into the inner workings of PyCRAM, but provides a guideline for structuring a program using PyCRAM. A more detailed explanation on the design of PyCRAM is given in section 4.2.

The `pycram` package

The `pycram` package and its modules do not provide ready to use plans. Instead it provides the building blocks to create a plan control program. These include the `designator`, `action_designator`, `language`, `process_modules` and `task` modules as the most important ones. The `language` module houses PyCRAM's implementation of the CPL macros, which are not used in the context of this thesis, so its usage is not further explained here. The `designator` module provides the base `Designator` class and the object, location and motion subclasses. The former two are used as parameters for plans, while the latter is the lowest level of abstraction to actions (or motions) performed by the robot. Action Designators, as mentioned before, are not subclassed from the base Designators and are found in their own module `action_designator`. Process modules are the robot-specific interfaces, which are called when resolving Motion Designators. The base class is provided by the `process_module` module. Finally, the `task` module houses the Task Tree features and provides the API to use them, such as the `with_tree` decorator.
Aside from all of that, PyCRAM also provides a BulletWorld interface in the `bullet_world` and `bullet_world_reasoning` modules. This simulation can be used to develop a plan control program without the need of an actual robot.
When creating a plan control program you would not develop extensions inside the pycram package, but rather create a new package importing those modules. In the following paragraphs all parts needed for such a package are explained.

Plans and Action Designator Description Modules

When thinking about what a robot should be able to do, we think about which actions it needs to perform. In PyCRAM actions are represented by two components: a plan and an Action Designator description.
Plans are essentially functions that perform Motion or Action Designators. A plan control program will need a collection of those functions, preferably in their own module. The functions have to be assigned the `with_tree` decorator, to make use of the Task Tree functionality. When the plan is supposed to make the robot execute a motion, it will perform Motion Designators. Action Designators are performed in order to call other plans.

```
# Low-level Plan performing a motion
@with_tree
def open_gripper(gripper):
  print("Opening gripper {}".format(gripper))
  ProcessModule.perform(MotionDesignator([('type', 'opening-gripper')↩
      , ('gripper', gripper)]))

# High-level plan performing actions
@with_tree
def example_plan(gripper):
    print("Showing an example with gripper {}".format(gripper))
    ActionDesignator(OpenGripperDescription(Arms.Left)).perform()
```

Each action type has its own `ActionDesignatorDescription` subclass. PyCRAM's `action_-designator` module provides some basic descriptions that any plan control program can use. If your program needs more or different descriptions, it is easy to create new subclasses. Any description just needs a constructor, which takes the information needed for the action as parameters, e.g. for moving a gripper: which arm's gripper should be moved. In the constructor body, you need to create an attribute for each parameter and set it to the parameter's value. During the resolution process (also called "grounding") a description might generate more information by using the knowledge base, that need to be saved for later inspection, it is recommended to also create these attributes in the constructor already.

```
class PickUpDescription(ActionDesignatorDescription):
  def __init__(self, object_designator, arm=None, grasp=None):
      self.object_designator = object_designator
      self.arm = arm
      self.grasp = grasp

      # Grounding attributes
      self.gripper_opening = None
      self.effort = None
      self.left_trajectory_poses = []
      self.right_trajectory_poses = []
```

Plans and action descriptions, or rather their resolution, have a symbiotic relationship. Plans use descriptions to perform other actions. Descriptions, when grounded, hold a plan's function for execution. They come together through an extra module for Designator resolution, explained in the next paragraph.
On a side note: Action Designator descriptions were implemented as classes, so that plan developers can have completion hints for the different parameters of a description. The resolution systems in CRAM and PyCRAM have been more or less completely string-based until now. This leads to a lot of development time being wasted looking up the correct keyword to

use when creating a Designator, without being able to just auto-complete it.

The `designator_resolution` module

To keep the program organized descriptions and plans should be stored in separate modules[1]. In order to avoid cyclic imports, there needs to be an extra module to bring plans and descriptions together. This module will be responsible for resolving Action Designators and choosing the appropriate plan functions.
A `DesignatorDescription` subclass will already have a `ground` function defined in its Action Designator description module. However, since this module cannot import the plans module, it is only defined as returning `self`. Grounding not only needs to fill the unset attributes mentioned in the previous section, but also needs to fill the description's `function` attribute with the right plan function and its parameters wrapped in an anonymous function. `ground` functions needing access to the plan functions and plans needing access to description classes is another (and the main) reason why a separate Designator resolution module is necessary. In this separate module the description classes' `ground` functions can be implemented outside of their class scope:

```python
from action_designator_descriptions import OpenGripperDescription
from plans import open_gripper

def ground_open_gripper(self):
  self.function = lambda : open_gripper(self.gripper)
  return super(OpenGripperDescription, self).ground()

OpenGripperDescription.ground = ground_open_gripper
```

Somewhere at the start of your main program code, at the latest before any Action Designators are resolved, you need to import the `designator_resolution` module. You will not be using anything of the module directly, but in it the Designator resolution is set up. And importing it will run that setup code.

run.py

To execute a program using PyCRAM a run.py file has to be used as the main entry point. In this file there should generally be just two imports and nothing else:

```python
import macropy.activate
import main
```

[1]In bigger programs, there might be even multiple modules of descriptions and/or plans, to separate high- and low-level or other groups of action types.

This is necessary because of PyCRAM's usage of macropy. For macros to work the `macropy.activate` module needs to be imported alongside your actual program code (in this case the `main` module). The run.py can then be executed like any other Python script.

4.2 PyCRAM Task Trees from a Developer's Perspective

This section will go into further detail on the design and technicalities of the Task Tree implementation.

4.2.1 Designing the Task Tree feature

One of the earliest decisions to be made was how developers should be able to use the Task Tree feature of PyCRAM. In analogy to the original implementation, it should not inconvenience the developer in writing plans. But each plan function call also has to take care of managing the action's place in the Task Tree. This involves creating a `TaskTreeNode` as a child of the current node before each call, placing the function's code inside the node, and then executing the function. Afterwards the current node has to be reset to the node's parent. So for each plan function call there has to be code injected before and after the actual plan function.
In Lisp, CRAM could make use of macros to easily achieve this. A macro can make arbitrary modifications to its body. To use the Task Tree features CRAM developers only need to substitute the standard `defun` macro with `def-cram-function` to define plan functions. In order to define a top-level plan function another macro has to be used, but that is just a tiny inconvenience. So CRAM developers don't have to think about the Task Tree feature's underlying mechanisms.
For PyCRAM, I wanted a comparable or better level of convenience. The existing PyCRAM implementation already uses the macropy library to implement CPL constructs as macros, but I wanted to look into native Python solutions first. Decorators are a very powerful tool in Python, capable of replacing a decorated function with another function. A decorator takes the original function as input and returns a new function to be called instead. This mechanism could be used to achieve the effects described above. The decorated plan function would get replaced with a function called `handle_tree`, that handles the Task Tree and calls the original function at the appropriate time.
But there is an important feature of Task Trees, that I set out to improve upon. Task Trees should be as modifiable as possible. In the original CRAM, the Task Tree's structure is very rigid and after an initial execution only the node's contents could be changed. This required a complicated way of thinking, for example when wanting to simply add another action after one that was already present in the tree. The original and the new action needed to be

combined into one action, that would then replace the original one. In principal, any arbitrary modification should be possible with this strategy, but would require complex considerations, the more involved the changes were.

PyCRAMs Task Trees should support an addition of a new action by simply adding a new node to the tree at an arbitrary place. The initial most obvious solution to me was to change the plan functions' code at runtime, whenever an action was to be added or removed. The macropy library uses Python's Abstract Syntax Tree to achieve code changes at runtime. But with this comes an array of complicated problems: Where to add the new function calls, how to find existing child functions and differentiate them from the newly added ones, what happens on re-execution, where not all the tree handling code should be executed again, etc. So the approach of directly changing the plans' code proved to be very complicated and as it turns out, also unnecessary.

The approach I chose for this thesis is pretty simple. It does use the `handle_tree` function proposed above, but with a slight change. Instead of directly calling the plan function after handling the node creation and setting it up as a child of the current node, it tells the current node to execute its next child (which was just added). While executing, the tree's nodes each keep track at which index of their children they currently are. Each node asserts itself as the currently active node globally, when starting execution and relinquishes the role when it is done executing. So each child's `handle_tree` call then again has the right current node at hand.

This gives two main benefits. Firstly, the nodes are identifiable via their index in their parents, so the `handle_tree` function can just tell the parent to execute the next child. This is necessary, because `handle_tree` has no way of knowing which child should be executed next. `TaskTreeNode`s keep track of the child they are supposed to execute via a counter variable `exec_step`.

Secondly, the nodes themselves now handle the execution of their plan function. This makes it easy to let them execute arbitrary code before or after that. For this purpose each node has what I call *execution siblings*. Each node can have a previous and a next execution sibling, which is, again, a normal `TaskTreeNode`. This creates a tree structure of execution siblings in each node. When executing, a node first executes its previous execution sibling, then its own plan function and finally the next execution sibling. Recursively, the execution siblings do the same of course. So the execution sibling tree gets traversed in Inorder. The execution sibling tree is separate from the overall Task Tree. A node does not directly know its children's execution siblings, but they are still executed, when those children are executed. This enables users of the Task Tree feature to easily inject new nodes at almost any point in the original plan. The only caveat being, that it has to be directly before or after an existing node.

Removing nodes should be equally simple. This design allows for easy deletion of nodes in the execution sibling trees as long as the execution siblings of the deleted node are brought up and reconnected accordingly. However, removing normal nodes from the initially created

tree seems to be a problem at first, because removing a child would throw off the parent's `exec_step` counter. But not actually removing deleted nodes, just replacing their `Code` object with an `NoopCode` object, i.e. modifying them to not execute any code, solves this problem as well. Note, that deleted nodes' execution siblings are still executed normally.
Also, a Task Tree should be easily executable after its initial creation. This was another reason why it made sense to let the nodes handle the execution of their functions instead of the decorator function. This way, one can just call the root's `execute` function and all the processes described above work in the same way. The only difference being that the `handle_tree` function needed to distinguish between initial Task Tree creation and later re-execution. This is explained in more detail in the next section.
In summary, using a decorator function in combination with execution sibling trees provided all the necessary flexibility to add and remove nodes, while retaining a rigid enough structure to be able to identify the correct child nodes for example. For these reasons I chose it as the right approach for this thesis.
One problem remains unsolved with this approach however. Failure handling is extremely important in robot control programs, because acting in the real world constantly results in failures. There are many different ways to go about handling failures, but basically an action's execution would be attempted and if it fails, that failure would be handled. Now, with the ability to replace and move actions in the plan there comes a problem. If some failure handled action A is replaced by another action B, the handling is now around B and would in many cases not make sense anymore. Furthermore, often not only a single action is failure handled, but multiple, for example because they form the failure-prone sub-part of an action. If one of those actions would be moved outside of the failure handled part, its failures are not handled anymore. Instead of putting restrictions on when and how certain modifications can be made to the tree, I decided to put a restriction on the design of plans that can be used with Task Trees: Every action that needs failure handling should do so itself. For failure handling that encompasses multiple actions, a new action with those as sub-actions should be created. This way, failure-handled actions are either moved all at once, or the developer probably knows what they are doing, when they modify actions in a failure-handled block.

4.2.2 The `with_tree` decorator

Arguably, the most important part of this thesis to get right was the decorator function used to interact with the Task Tree feature. Developers of PyCRAM plans can program their plans just like before. When the use of Task Trees is desired, they only need to add the `with_tree` decorator to their functions, like so:

```python
@with_tree
def do_something():
    ...
```

Nothing else has to be done in order for actions executing that plan function to end up in the Task Tree.

In the remainder of this section, I show the whole decorator function and explain it in more detail. Code listing 4.1 contains the source code. The decorator takes in the original function as a parameter, defines a `handle_tree` function, which ultimately puts the original function in the tree, and returns it as the new function. `TASK_TREE` and `CURRENT_TASK_TREE_NODE` reference the root and currently active `TaskTreeNode` object respectively. The function distinguishes between top-level and normal plans.

When a plan is executed, while there is no active Task Tree, it automatically becomes a top-level plan. Then the initial setup of the global variables is done and the newly created root node is executed. When there is an active Task Tree, the plan function is put inside a `TaskTreeNode`, which is added as a child to the currently active node and then executed. But a new node is only created and added to the tree if the tree is being executed for the first time, i.e. is in the process of being built. This is checked via the `exec_step` attribute and the number of children. If the node has enough children to execute the next at the `exec_step` index, the tree is of course already built. In all cases the function is finally executed and the result returned.

```python
def with_tree(fun):
  def handle_tree(*args, **kwargs):
      global TASK_TREE
      global CURRENT_TASK_TREE_NODE
      code = Code(fun, args, kwargs)
      if CURRENT_TASK_TREE_NODE is None:
          TASK_TREE = TaskTreeNode(code, None, fun.__name__)
          CURRENT_TASK_TREE_NODE = TASK_TREE
          result = CURRENT_TASK_TREE_NODE.execute()
      else:
          if len(CURRENT_TASK_TREE_NODE.children) <= ↩
              CURRENT_TASK_TREE_NODE.exec_step:
              new_node = TaskTreeNode(code, CURRENT_TASK_TREE_NODE,
                                      '/'.join([CURRENT_TASK_TREE_NODE.↩
                                          path, fun.__name__]))
              CURRENT_TASK_TREE_NODE.add_child(new_node)
          result = CURRENT_TASK_TREE_NODE.execute_child()
      return result
  return handle_tree
```

Code Listing 4.1 Source code of `with_tree` decorator

Chapter 5

Evaluation

This chapter evaluates the capabilities of the Task Tree implementation by showcasing their usage and visualizing the results.

5.1 Task Tree Capabilities

To evaluate the functionality of the basic capabilities of Task Trees, I created a simple demonstration plan function, on which the functions are to be executed. The result was then inspected afterwards. The demonstration plan consists of a top-level plan, calling two lower-level plans, one of which again calls two other plans. Code listing 5.1 contains the source code for these plans.
When executed the top-level plan produces this output and tree:

```
Top-level Plan: Executed.
Plan A: Executed with param: foo.
Plan B: Executed.
Plan C: Executed.
Plan B: Log between plan calls.
Plan C: Executed.
```

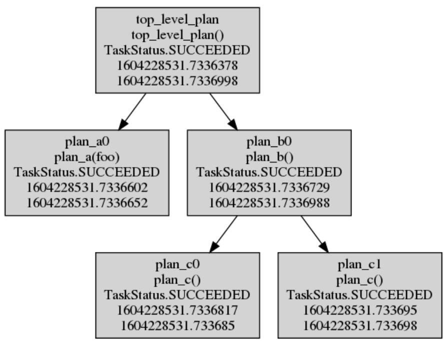


Figure 5.1 Evaluation base plan.

```python
@with_tree
def top_level_plan():
    print("Top-level Plan: Executed.")
    plan_a("foo")
    plan_b()

@with_tree
def plan_a(param):
    print("Plan A: Executed with param: {}.".format(param))

@with_tree
def plan_b():
    print("Plan B: Executed.")
    plan_c()
    print("Plan B: Log between plan calls.")
    plan_c()

@with_tree
def plan_c():
    print("Plan C: Executed.")
```

Code Listing 5.1 Source code of demonstration plans for evaluation.

5.1.1 Inserting nodes

For insertion an additional plan is needed which can be used as the code for a new node. For this evaluation first this new plan is inserted after plan A, then another plan B node is added before plan A. For each insertion the tree before and after re-execution will be shown alongside the console output.

Inserting a new node without subactions

The new node inserted here is a node containing a `Code` object with the `plan_d()` function. Plan D has no subactions and thus will not add any other nodes via side effects.

```python
@with_tree
def plan_d():
    print("Plan D: Executed.")
```

```
Top-level Plan: Executed.
Plan A: Executed with param: foo.
Plan D: Executed.
Plan B: Executed.
Plan C: Executed.
Plan B: Log between plan calls.
Plan C: Executed.
```

As can be seen in the output the newly added node is executed right after plan A. Apart from that this also shows the insertion caused no additional side effects or problems for the

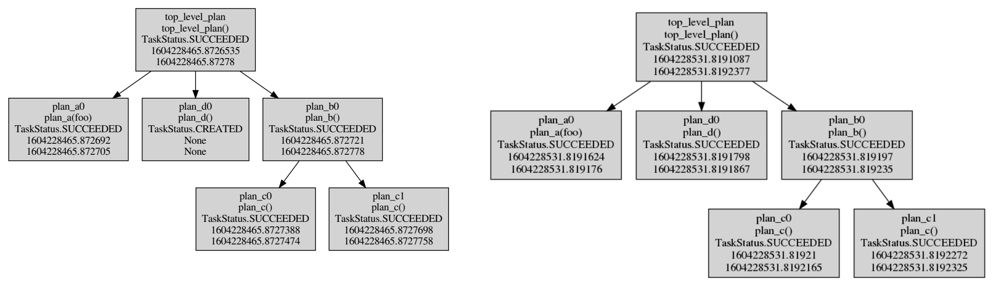

top_level_plan top_level_plan() TaskStatus.SUCCEEDED 1604228465.8726535 1604228465.87278
plan_a0 plan_a(foo) TaskStatus.SUCCEEDED 1604228465.872692 1604228465.872705
plan_d0 plan_d() TaskStatus.CREATED None None
plan_b0 plan_b() TaskStatus.SUCCEEDED 1604228465.872721 1604228465.872778
plan_c0 plan_c() TaskStatus.SUCCEEDED 1604228465.8727388 1604228465.8727474
plan_c1 plan_c() TaskStatus.SUCCEEDED 1604228465.8727698 1604228465.8727758
top_level_plan top_level_plan() TaskStatus.SUCCEEDED 1604228531.8191087 1604228531.8192377
plan_a0 plan_a(foo) TaskStatus.SUCCEEDED 1604228531.8191624 1604228531.819176
plan_d0 plan_d() TaskStatus.SUCCEEDED 1604228531.8191798 1604228531.8191867
plan_b0 plan_b() TaskStatus.SUCCEEDED 1604228531.819197 1604228531.819235
plan_c0 plan_c() TaskStatus.SUCCEEDED 1604228531.81921 1604228531.8192165
plan_c1 plan_c() TaskStatus.SUCCEEDED 1604228531.8192272 1604228531.8192325

Figure 5.2 Inserting a new node.
Left: Task Tree after inserting new Plan D node. Right: Task Tree after re-execution

re-execution of the Task Tree.

Inserting a node with subactions

Here a Plan B node is added to the initial tree. Plan B has two Plan C subactions and will cause those to be added as well, when the tree is executed.

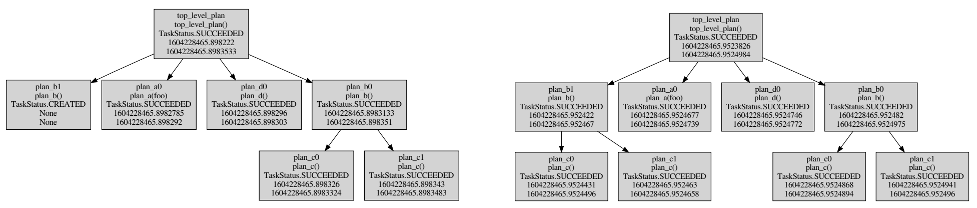

top_level_plan top_level_plan() TaskStatus.SUCCEEDED 1604228465.898222 1604228465.8983533
plan_b1 plan_b() TaskStatus.CREATED None None
plan_a0 plan_a(foo) TaskStatus.SUCCEEDED 1604228465.8982785 1604228465.898292
plan_d0 plan_d() TaskStatus.SUCCEEDED 1604228465.898296 1604228465.898303
plan_b0 plan_b() TaskStatus.SUCCEEDED 1604228465.8983133 1604228465.898351
plan_c0 plan_c() TaskStatus.SUCCEEDED 1604228465.898326 1604228465.8983324
plan_c1 plan_c() TaskStatus.SUCCEEDED 1604228465.898343 1604228465.8983483
top_level_plan top_level_plan() TaskStatus.SUCCEEDED 1604228465.9523826 1604228465.9524984
plan_b1 plan_b() TaskStatus.SUCCEEDED 1604228465.952422 1604228465.952467
plan_a0 plan_a(foo) TaskStatus.SUCCEEDED 1604228465.9524677 1604228465.9524739
plan_d0 plan_d() TaskStatus.SUCCEEDED 1604228465.9524746 1604228465.9524772
plan_b0 plan_b() TaskStatus.SUCCEEDED 1604228465.952482 1604228465.9524975
plan_c0 plan_c() TaskStatus.SUCCEEDED 1604228465.9524431 1604228465.9524496
plan_c1 plan_c() TaskStatus.SUCCEEDED 1604228465.952463 1604228465.9524658
plan_c0 plan_c() TaskStatus.SUCCEEDED 1604228465.9524868 1604228465.9524894
plan_c1 plan_c() TaskStatus.SUCCEEDED 1604228465.9524941 1604228465.952496

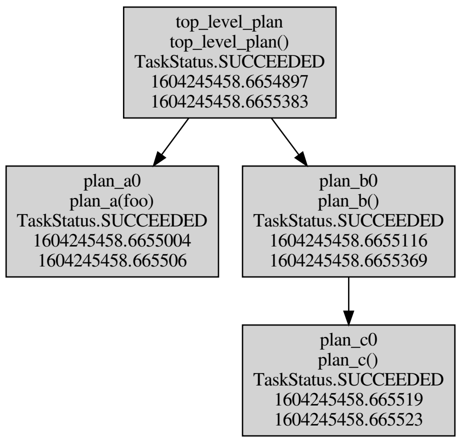
Figure 5.3 Inserting a new node with sub actions.
Left: Task Tree after inserting a new plan B node. Right: Task Tree after re-execution.

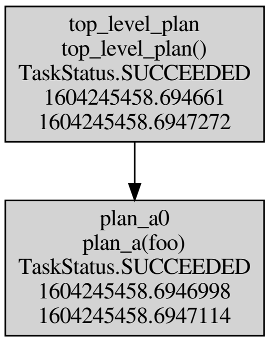
```
Top-level Plan: Executed.
Plan B: Executed.
Plan C: Executed.
Plan B: Log between plan calls.
Plan C: Executed.
Plan A: Executed with param: foo.
Plan D: Executed.
Plan B: Executed.
Plan C: Executed.
Plan B: Log between plan calls.
Plan C: Executed.
```

This case is more interesting. The before image shows the new plan B not executed yet, and thus having no children in the tree. During execution the children are added, as though the new node was part of the tree all along. From here on out the new nodes are equally accessible as the original ones and the tree as a whole is not in any way limited to further modifications.

5.1.2 Removing nodes

Deleting a node from the tree results in that node not showing up in the generated visuals and not being executed, when the corresponding function call is reached in the parent plan. This can be seen in the left example of figure 5.4, where a leaf node has been deleted. On the right side the node with plan B has been deleted. Deleting an inner node of course removes the children of that node.



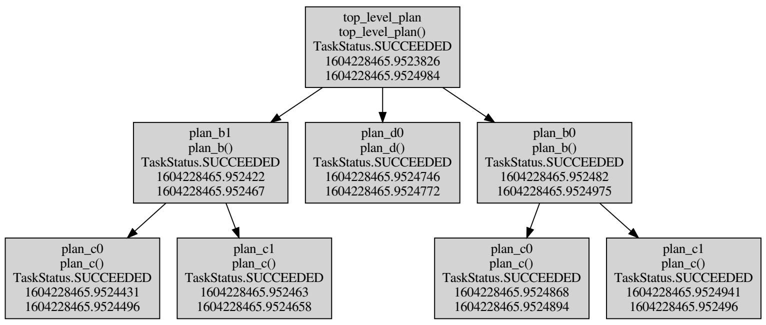


```
Top-level Plan: Executed.
Plan A: Executed with param: foo.
Plan B: Executed.
Plan C: Executed.
Plan B: Log between plan calls.
```

```
Top-level Plan: Executed.
Plan A: Executed with param: foo.
```

Figure 5.4 Deletion of a single node.
Left: Deleting the second Plan C. Right: Deleting Plan B.

When inserting a node before or after an existing node, the new node is (at least programatically) not a direct child of the node's parent. It is attached to the node it is inserted in reference to as a execution sibling. When the original node is deleted, the inserted nodes have to remain in place and should not be removed as well, as they are not children of that original node. This is achieved by not completely removing a deleted node from the tree, but rather emptying it and turning it into a noop node, as explained in chapter 4.
Figure 5.5 shows the tree from figure 5.3 after deleting plan A. The inserted nodes of plan B and D are not affected.



```
Top-level Plan: Executed.
Plan B: Executed.
Plan C: Executed.
Plan B: Log between plan calls.
Plan C: Executed.
Plan D: Executed.
Plan B: Executed.
Plan C: Executed.
Plan B: Log between plan calls.
Plan C: Executed.
```

Figure 5.5 Deleting a node with execution siblings.

5.1.3 Moving nodes

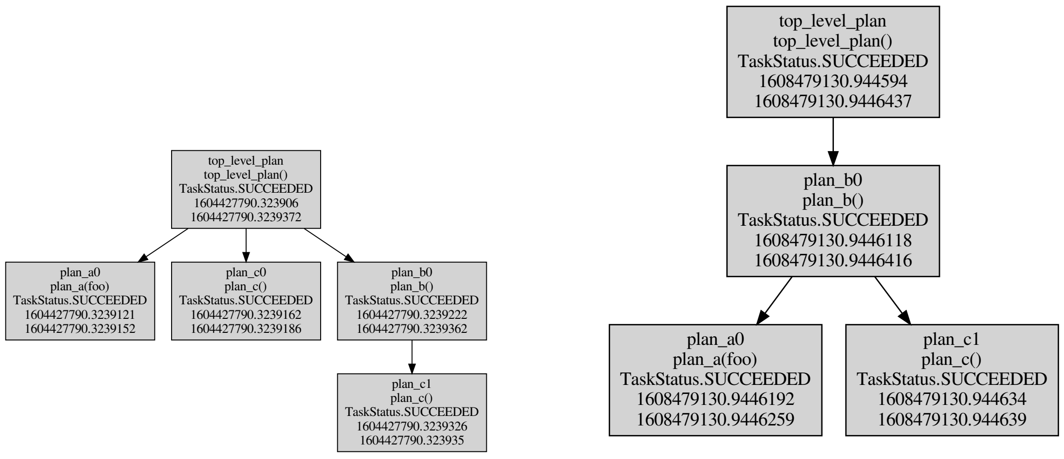
Moving a node from one point to another in the tree should remove the node at the original position and insert it at the specified location. In principal the moving of nodes is just a convenient combination of copying, removing and inserting nodes. But for completion's sake figure 5.6 shows the move of a plan C node to directly after plan A on the left and plan A being moved to replace the first plan C on the right. Both show the tree after re-execution, with the console output below to showcase the changed order actually took effect.

5.1.4 Changing Parameters

In all of the previous examples it can be seen that plan A is called with a single parameter with the value `foo`. In this last showcase the value of this parameter will be changed to `bar`. As much is visible in figure 5.7, where the tree and the output show the changed parameter.

5.2 Integration

In this section I showcase the three capabilities illustrated in the Introduction, using the implemented Task Tree functionality. First the capabilities introspection, prospection-based



```
Top-level Plan: Executed.
Plan A: Executed with param: foo.
Plan C: Executed.
Plan B: Executed.
Plan B: Log between plan calls.
Plan C: Executed.
```

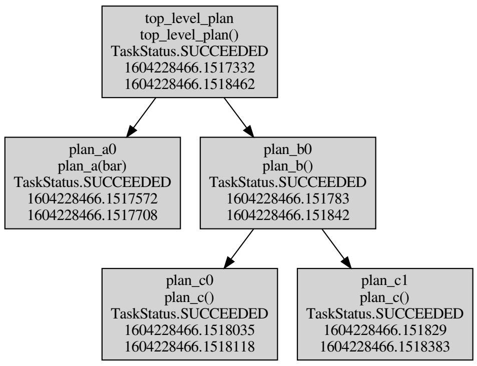
```
Top-level Plan: Executed.
Plan B: Executed.
Plan A: Executed with param: foo.
Plan B: Log between plan calls.
Plan C: Executed.
```

Figure 5.6 Moving nodes.
Left: Moving a node to be inserted after another. Right: Moving a node to replace another.

reasoning and action reorganization are examined on their own. Finally a comprehensive demonstration scenario is described, in which all three capabilities find application. All of the following sections will describe a self-contained scenario and a corresponding plan, propose improvements to the original plan and evaluate the results of these improvements. Each plan is executed in a simulated environment using PyCRAM's BulletWorld integration. Before each execution the original state of the world is restored, so that each plan execution is done under the exact same conditions. Figure 5.8 shows the simulated environment, in which the plans have been executed.

5.2.1 Introspection

Consider a robot tasked to retrieve a bowl from a drawer in a kitchen. The robot knows the kitchen and the drawer's location in it. For this evaluation, we also assume the opening distance for the drawer is given as well. So for execution of its task, the robot still needs to find a suitable position to navigate to and be able to open and look inside the drawer. The actions of interest for this evaluation are the *navigation* and *detect* actions. The first of course navigates the robot to the desired position, while the latter is used to instruct the vision



```
Top-level Plan: Executed.
Plan A: Executed with param: bar.
Plan B: Executed.
Plan C: Executed.
Plan B: Log between plan calls.
Plan C: Executed.
```

Figure 5.7 Changing a parameter inside a Task Tree node.

system of the robot to check for the desired object in the view. Of secondary interest is also the *look_at* action, that is performed before the *detect* action in order to make the robot move its neck so that its gaze is directed at the likely position of the object (in this case the inside of the drawer). After the first *navigation*, the robot will always be able to open the drawer and after it has detected the object, it will perform a *pick_up* action and park its arms.
In this scenario the robot has to perform the actions up to the detection, in order to see if it works. In real applications the robot needs to select a target location from many possible candidates. This can result in many retries and wasted time, if the choice is not suitable for detecting the target object. Figure 5.9 shows the Task Tree of such an execution. The red framed part is just the failed tries of the *detect* action and the navigation before it. The green frame shows the final *navigate* and *detect* actions, which are the only actions of their sort actually contributing to the desired outcome of the whole plan. For this evaluation the new navigation poses for each try are not sampled from a distribution, but generated by a function. For every run the generated poses are the same and will always result in the three failed attempts seen in figure 5.9. This is of course an arbitrary number but it is most likely lower than a real sampling would result in.
This is very impractical. Each new navigation can lead to lengthy execution times, depending on the robot. Most robots will need some (if even just a little) time to start and stop their motors for each motion. With two unnecessary motions per retry this can quickly add up. And this is also just for the single action of retrieving a bowl.
The optimization for this scenario becomes possible after a previously executed plan with a

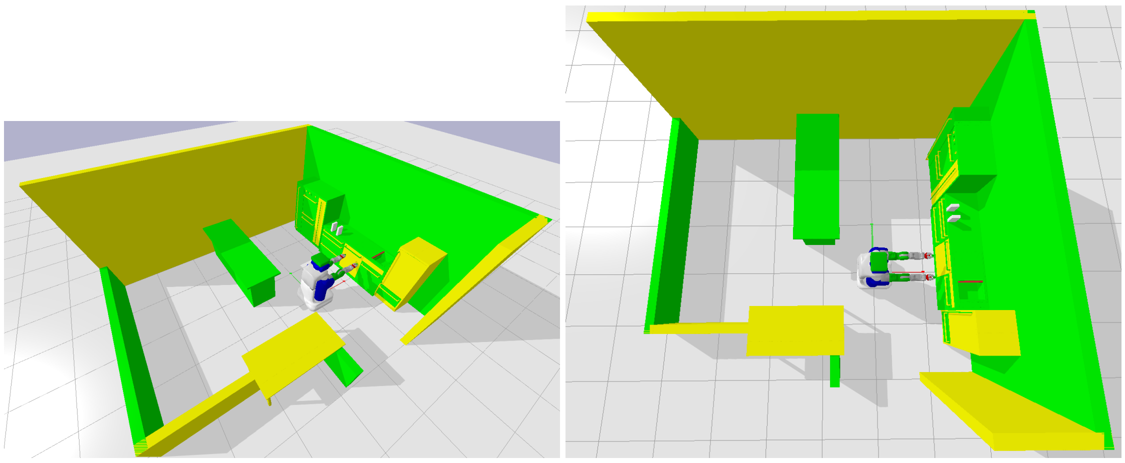


Figure 5.8 Simulated kitchen environment.

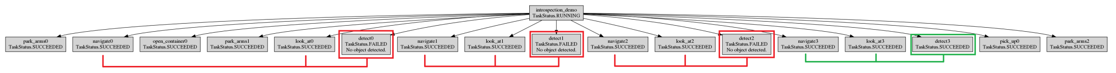


Figure 5.9 Task Tree with many actually performed retries of a single action.

similar scenario becomes available for introspection. In this case the Task Tree of figure 5.9 is saved for the same plan to reuse the information contained within. After detecting the bowl in the previous execution, the robot can inspect the Task Tree to retrieve what parameter it used for the *navigate* action and use it again. The resulting Task Tree can be seen in figure 5.10. As before the green frame shows the detect action that succeeded. Since it could reuse the parameters used in the previous bowl retrieval, only a single *navigate* action was necessary to achieve the same results.

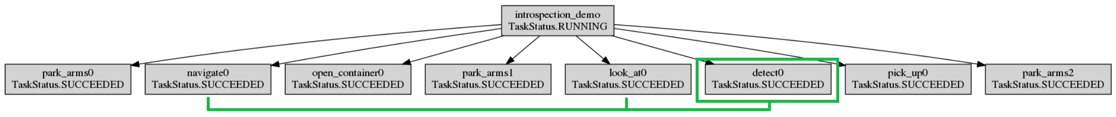


Figure 5.10 Task Tree optimized by the use of introspection.

Table 5.1 compares the number of actions performed in 5.9 and 5.10, 17 and eight. It can be seen that introspection alone already can lead to substantial improvements in a plan. In this demonstration the *detect* action fails three times before it can successfully find a bowl. Each failure results in three new actions needed to achieve the goal. These extra nine actions in the unoptimized plan are practically useless and just add execution time and potential risk. Using introspection for analysis after the fact is very simple, but it does require some extra work to be useful at runtime. In this evaluation, the Task Tree of a successful execution was just saved in a global variable to be reused later. In a production environment with many

	# of Actions
Without Introspection	17
With Introspection	8

Table 5.1 Number of actions in original and optimized execution using introspection.

different plans, that are used in differing contexts, this is not at all practical. But one can imagine a library of serialized Task Trees, that are stored based on applicable contexts. But such a library is out of scope for this thesis.
On its own introspection can provide most of its value for analysis purposes. To look up parameters used and actions performed, information that can be easily lost in a big framework of plans. This thesis extends PyCRAM with the capability to inspect the last executed plan, which can provide insight into this data.

5.2.2 Prospection-based reasoning

Now, similar to the plan in the previous section, consider a robot tasked to retrieve another object from a drawer. This time a spoon. In this scenario the robot already knows a good place to navigate to, but has to find a suitable opening distance for the drawer. Also the robot does not have prior experience with this drawer, so it can't look up the necessary information. In order to find the right parameter, the robot needs to test different opening distances and choose one that works. But as discussed in the previous section, trying out actions for real takes time and can also lead to damage if something dangerous is performed.
So in order to reduce time and risk, prospection can be used to let the robot perform the actions in simulation. To then use simple reasoning to retrieve the best set of parameters. In this scenario any set of parameters, that leads to a successful pickup of the spoon, will suffice. The robot tries to perform the actions depicted in figure 5.11. The opening distance parameter for the open_container is again taken from a simple generator. If it can't see the spoon it will fail on the *detect* action and try the next parameter. When the simulation succeeds, the used parameters are extracted via the `get_successful_params_ctx_after` function described in section 4.1.4.2. Figure 5.12 shows the Task Tree after the successful pickup of the spoon, using the extracted parameters from the prospection.

The demonstration plan needs four simulations to find a parameter set that results in a successful pickup of the spoon. The unsuccessful executions each perform four actions, the successful one performs five. Table 5.2 compares the number of actions performed in the real world with and without use of prospection. In total 17 actions are simulated and afterwards five are performed in the main simulation. Without prospection the 17 actions would have to

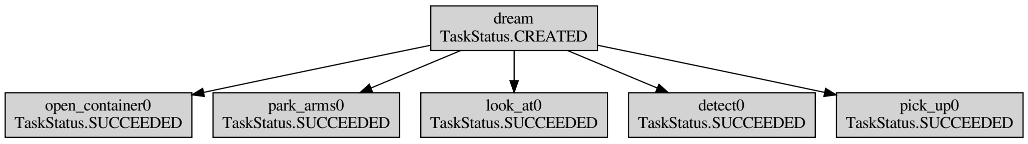


Figure 5.11 Task Tree of simulated actions in prospection demo plan with reduced labels.

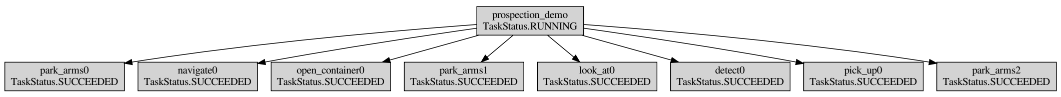


Figure 5.12 Task Tree of prospection demo plan with reduced labels.

be performed to try out the different parameters and eventually pick up the spoon. So there would be twelve actions basically only contributing to execution time and risk.
Prospection-based reasoning enables plan developers to further generalize their plans, because it can account for irregularities in the environment or differences between similar contexts. Not every container can be opened the same way, not every object can be seen from the same point. So having the possibility to quickly iterate through multiple sets of parameters and try them out is a very powerful feature of Task Trees in combination with the Bullet simulation engine.

5.2.3 Action Reorganization

For this scenario, we look at a more complex task. The plan involves transporting two objects from one point to another. The target objects are a milk carton and a box of cereal. The milk is located in a fridge on one side of the kitchen and the cereal on a counter next to the fridge. They are supposed to be transported to a table on the other side of the kitchen.
There is a generic plan for transporting a single object. The plan in this scenario uses an action corresponding to that plan to transport the objects. So each object is transported on its own. A single transport plan consists of a navigation, an optional container opening, a

	# of Actions
Without Prospection	22
With Prospection	5

Table 5.2 Number of actions in original and optimized execution using prospection.

pickup, an optional container closing, another navigation and finally a placing action. This equates to four navigation actions in total. At least three being long distance from one side of the kitchen to the other.

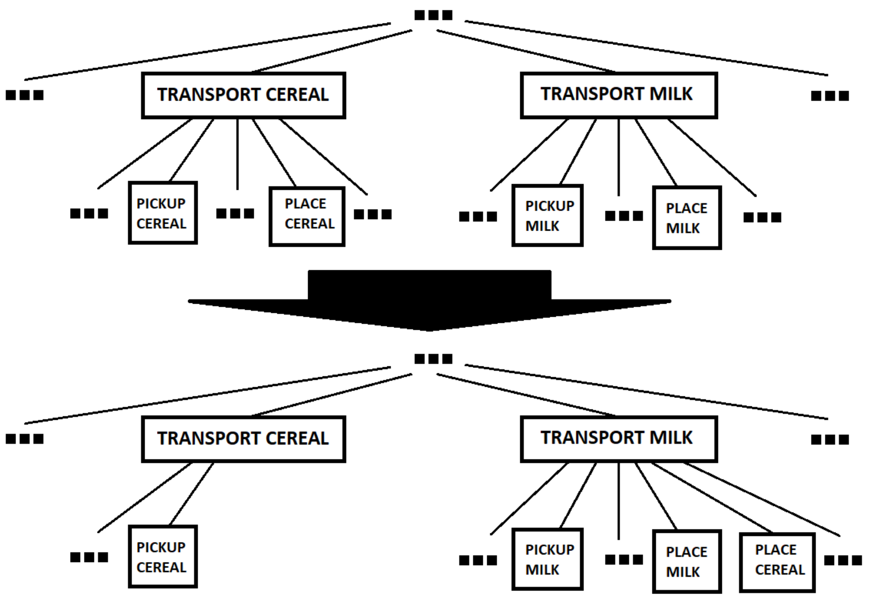


Figure 5.13 Task Tree for transporting two objects transformed to transport them at once. The `place` action of the first object is delayed until after the second `place` action.

Simple action reorganization can achieve needing only one initial, one short and one long navigation. Using an already performed Task Tree of this scenario as the base, a simple function can reorganize the actions as follows: Move the first `place` action after the second and remove all actions in the first `transport` after and including the second `navigate`. This way the first `transport` ends after picking up the milk and closing the fridge and the second `transport` includes an additional `place` action. Because the robot did not move to the other side of the kitchen after picking up the milk, the navigation to pick up the cereal is also shorter. Figure 5.13 shows a simplified depiction of this reorganization.

Position	**X**	**Y**
A (Start)	0m	0m
B (Milk)	0.5m	-0.4m
C (Cereal)	0.6m	0.9m
D (Destination)	-1.8m	1m

	Path	**Distance**
Original	A, B, D, C, D	8.14m
Reorganized	A, B, C, D	4.35m
Distance saved:		**3.79m** (46.56%)

Table 5.3 Positions and distances traveled in Action Reorganization evaluation plan.

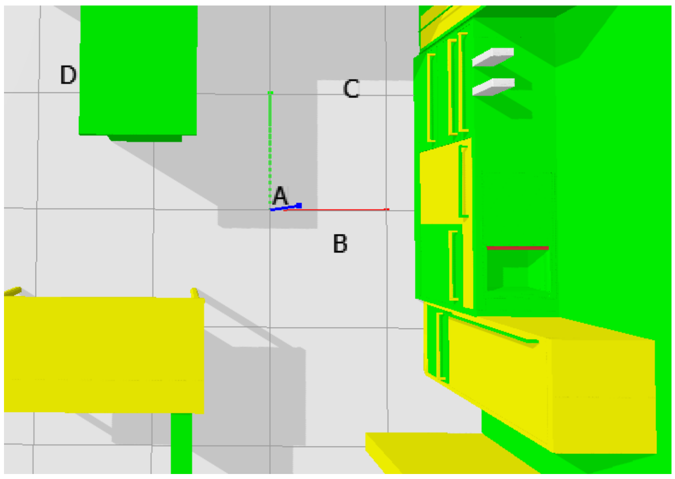


Figure 5.14 Locations visited during action reorganization plan.

Since all the actions are performed in simulation, it is hard to measure the actual impact the reorganization has. But as a very broad approximation, we can observe the decrease in the euclidean distances traveled by the robot (keep in mind, that the simulation just teleports the robot to its destination, so the actual distances traveled would be higher). Figure 5.14 and tables 5.3 show the positions and distances for the plan before and after reorganization. Even though the optimized plan uses only one navigation less, the total distance traveled was decreased by 46.56%. This is just a small sample, but it already shows the potential effect action reorganization can have. More complicated and less generic optimizations could be made in different contexts. For example, if a robot is tasked with fetching a greater number of small objects from a single location, it would be feasible to have it get another object, e.g. a tray, to collect the objects on, before navigating to the destination. This could improve the robot's navigation times even more than what was observed here.

5.2.4 Robot Behavior Adaption by combining the Task Tree capabilities

Finally, we take a look at a comprehensive scenario, where all of the aforementioned optimizations are applied in conjunction. The robot is tasked with setting a table. It has a generic plan for setting a table for breakfast for a single person, called `set_table`. That plan involves transportation of the following items: a bowl, a spoon, a milk carton and a box of cereal. As before, the spoons and bowls are stored in drawers, the milk is in the fridge and the cereal is located on the counter top. The plan involves failure handling for placement of the objects, so that when the first seat a the table is taken (i.e. there are already objects placed at that seat), the next seat is chosen as the destination. For the transport of milk and cereal the execution is aborted, when the prospection finds objects already placed at the destination.

Another plan, `set_table_for_two`, uses `set_table` twice to instruct the robot to set the table for two people.
The overall demonstration for this evaluation calls `set_table_for_two` once, resets the simulation and calls it again. The second call uses the Task Tree from the first execution to optimize by using introspection. Prospection-based reasoning is used in both calls, because otherwise the execution would not be possible without for example risk of collisions happening. After the second execution the world is reset again and the Task Tree is optimized using action reorganization. Finally the optimized Task Tree is executed. A video of this procedure inside the simulated kitchen environment is available online[1].
Introspection is used to retrieve parameters for the navigation and container opening distance of the spoon and bowl fetching. Prospection-based reasoning allows the robot to simulate the placing of the bowl and spoon. Also the whole of the transport actions of milk and cereal are done in simulation first and only if they succeed the same parameters are used to perform the actions for real. It would technically be possible to not use prospection right away, but that would require manually resetting the robot and world to a state where the action can be retried. One would have to make sure that no object was hit and moved out of place as well. So while possible, it would not be very practical for this demonstration.
Table 5.4 shows how many real actions are attempted in the original and optimized versions of `set_table_for_two`. The optimizations considered here are only through introspection and prospection-based reasoning. Note, that not all of the numbers under "Original" could be counted when executing the demonstration plans. For the actions which use prospection right away, the numbers were calculated based on the number of failed simulations. Refer to footnote *a* for more details. These numbers give a good impression of what kinds of improvements can be made. In the optimized plans, only the necessary actions (i.e. those that actually succeed and achieve their goal) are performed outside of the simulation. This saves execution time and reduces risk.
The reorganization of actions for this demonstration is a bit more complex than the previous example. But it is done with the same goal in mind: reducing the number and distances of navigation actions. To achieve this the fetching and delivering of objects of the same type was combined. The pickup and place actions for bowl and spoon from the second `set_table` were moved to right after their counterparts in the first plan, with the gripper parameter changed to the other arm of course. For the placing actions, the navigation to the other side of the table was included in the move as well. Since both grippers are full after getting two objects from a drawer, the closing of each drawer is delayed until the robot has placed the objects again. This might not strictly be necessary, since the robot could also push the drawers close, but it is included here anyway. Figure 5.15 shows the different locations in the kitchen visited by the robot during the `set_table_for_two` plan. In table 5.5 the paths taken in the original and optimized plan are compared.

[1] *https://youtu.be/bUSdWfMjyrI*

Action	**Original**	**Optimized**
Fetch Bowl	8	2
Deliver Bowl	5[a]	2
Fetch Spoon	5[a]	2
Deliver Spoon	5[a]	2
Transport Milk	3[a]	1[b]
Transport Cereal	3[a]	1[b]

[a]The unoptimized numbers for these actions are based on the number of failed simulated attempts in the demonstration plans. The premise being, without prospection each failed simulated action would have to be taken still, but without simulation. So this table lists how many real attempts it would have taken without the use of prospection.
[b]In addition to the real action taken, a simulated transport of milk and cereal is attempted, but no additional real action is taken.

Table 5.4 Comparison of real action attempts needed in `set_table_for_two` with and without optimization

As with the evaluation of action reorganization by itself (section 5.2.3), the distance saved equals to roughly half of the original distance traveled. It is worth noting however, that these numbers do not represent actual distances that would be traveled by an actual robot, because all of these evaluations are done in a simulated environment. The number of navigation actions is also reduced by a third. Not as significantly, but as noted before, the number of motions executed on the robot can also have an impact on overall execution time.

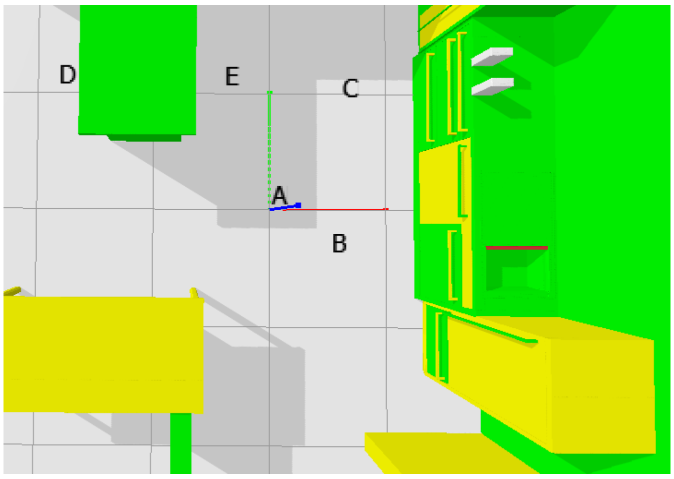


Figure 5.15 Locations visited during `set_table_for_two`.

Position	**X**	**Y**
A	0m	0m
B	0.5m	-0.4m
C	0.6m	0.9m
D	-1.8m	1m
E	-0.4m	1m

	Path	**Distance**
Original	A, C, D, C, D, B, D, C, D, C, D, E, C, D, E	31.45m
Reorganized	A, C, D, E, C, D, E, B, C, D	15.49m
Distance saved:		**15.96m** (50.75%)

Table 5.5 Comparison of original and optimized navigation paths in `set_table_for_two`.

Chapter 6

Conclusion

6.1 Summary

I set out to develop a easy-to-use but powerful Task Tree implementation as a basis to enable behavior adaptation in robot plans. I argue for this approach based on three key capabilities: introspection, prospection-based reasoning and action reorganization. Task trees inherently support or enhance those capabilities in a robot control system.
First I implemented Action Designators and a corresponding resolution system to represent and execute actions. I then presented the `with_tree` decorator as the key feature of PyCRAM's Task Tree extension. It enables any function to be recorded in a global Task Tree. Decorating plan functions with `with_tree` and using Action Designators to let plans call each other enables PyCRAM to build hierarchical generic plans capable of performing household activities like setting a table. Additionally an array of convenience functions is provided for simulating actions, extracting information or modifying Task Trees.
In evaluating I demonstrated the functionality of the basic operations on a Task Tree on one hand. And showcased the advantage of using knowledge acquired through the use of Task Trees on the other. Using Task Trees in context of the key capabilities lead to significant performance improvements and risk reduction in all showcases.

6.2 Discussion

I view the PyCRAM extension developed in this thesis, as a good basis to build upon. The evaluation has shown the practical benefit of using Task Trees over not using them. From developing the plans for the evaluation, I find the usability to plan developers to be very good. It does not posit any real overhead for development of the actual plans, but provides value for analyzing and optimizing plan execution. The interface to inspect and modify Task Trees is straight-forward and easy to grasp. Only when doing very involved action reorganizations, like in the final evaluation, I had to really think about how to achieve the desired transformation without impacting the parts I did not want to interfere with. But I think this is pretty much unavoidable because of the high causality present in most plans. One other thing I would have liked to be even more easy to use would be the simulation of actions. Or rather the extraction of parameters at the end. It does not feel as intuitive as I would have liked it to be. Finally, setting up a new project with the structure I explained in section 4.1.6 is not straight-forward and involves importing a module, from which no symbol is ever actually needed anywhere. I imagine there has to be a better way to setup the Action Designator resolution system. If not, maybe a boilerplate code generator could be a nice addition.

6.3 Possible Extensions and Limitations

As mentioned before this thesis' work should serve as a basis for more specific extensions in the future. Most of what goes beyond the core functionality of Task Trees is not implemented yet.

For now only a single Task Tree is recorded and available after execution of a plan. If another plan is executed, the Task Tree gets overwritten. This could of course be easily mended by having a list of past trees or something similar. But a more substantial extension would be to automatically store recorded Task Trees based on some criteria and be able to query for tree based on those. Criteria could be the actions involved, the environment of the plan execution or anything else. Another approach could be to form episodic memories of tasks a robot performs. This would also entail recording data from the sensors of the robot and creating a more exhaustive recording of past actions. This approach was already followed by CRAM developers in the past and could also be an interesting avenue for PyCRAM. A very practical and necessary addition for these extensions to make sense would be to implement a serialization method for Task Trees to save them in files for later reuse.

Instead of only checking for success or failure, a more complex reasoning mechanism for prospected actions would be another worthwhile extension. In the plans used for the evaluation only the prospected actions' success was of importance. But in most cases multiple parameterizations of the same action will lead to a successful result. But one can certainly imagine cases in which the cost of performing those parameterized actions differs greatly. PyCRAM does provide the interface to do those more complex cost calculations, but it is not convenient. A system where the developer only has to describe their desired cost function would be a possible solution to this inconvenience.

As mentioned in the opening section of this chapter, doing complex action reorganization can be cumbersome. A possible remedy are rule-based transformations. Developers would only need to design rules that work by, for example, pattern matching and a corresponding transformation that would be applied when a rule matches. Such rules would only have to implemented once and could then be reused all the time and thus would make the cumbersome

manual action reorganization unnecessary. PyCRAM would provide the interface to design these rules and transformations.
Aside from extensions to the features, the Task Tree implementation itself has some limitations. Most notably multi-threading is not fully supported as of now. In the original CRAM, tasks were implemented to each be run on separate threads. I decided against going that route because Python does not support interrupting threads. Since the tasks themselves were not supporting multi-threading I also decided against working on multi-threading support for this thesis. However I think it could be a pretty straight-forward addition, since the only critical resource is the Task Tree itself and only the `with_tree` decorator interacts with it.

Appendix A

Appendix

A.1 Publication of the Software

The PyCRAM framework is hosted on GitHub.com. The Task Tree extension is currently only available in a fork. Because PyCRAM as a whole is under active development and the Task Tree extension is not yet merged into the main repository, the code might change over time. But the software developed for this thesis will be stored on a branch called `thesis` of the forked repository. Also the hash is stated below to ensure no changes could have been made after the thesis' submission. A copy of the software in current form is also included on the USB drive submitted with the paper form of this thesis.

Software	Repository
PyCRAM	*https://github.com/cram2/pycram.git*
PyCRAM with Task Trees	*https://github.com/cpollok/pycram.git* Branch: thesis Hash: a1f178808faecef1dd25adad150a400dcdac47ee

A.2 List of Tables

A.3 List of Figures

A.4 List of Code Listings

A.5 Bibliography

[1] Andy Augsten and Dustin Augsten. "PyCRAM – Python-based concurrent reactive programming language for autonomous mobile manipulation". Bachelor's Thesis. University of Bremen, 2019.

[2] Michael Beetz, Dominik Jain, Lorenz Mosenlechner, Moritz Tenorth, Lars Kunze, Nico Blodow, and Dejan Pangercic. "Cognition-Enabled Autonomous Robot Control for the Realization of Home Chore Task Intelligence". In: *Proceedings of the IEEE* 100.8 (Aug. 2012), pp. 2454–2471.

[3] Antoine Cully, Jeff Clune, Danesh Tarapore, and Jean-Baptiste Mouret. "Robots that can adapt like animals". In: *Nature* 521.7553 (May 2015), pp. 503–507.

[4] Jonas Dech. "PyCRAM - Accurate Physics-based Environment for Executing Mobile Pick and Place Plans". Bachelor's Thesis. University of Bremen, 2019.

[5] R James Firby. "An Investigation into Reactive Planning in Complex Domains". In: *AAAI'87: Proceedings of the sixth National conference on Aritifical Intelligence.* Vol. 1. Seattle, Washington: AAAI Press, 1987, pp. 202–206.

[6] R. James Firby. "Building Symbolic Primitives with Continuous Control Routines". In: *Proceedings of the First International Conference on Artificial Intelligence Planning.* Elsevier, 1992, pp. 62–69.

[7] R. James Firby. "Task Networks for Controlling Continuous Processes". In: *Proceedings of the Second International Conference on Artificial Intelligence Planning.* 1994, pp. 49–54.

[8] Robert James Firby. "Adaptive Execution in Complex Dynamic Worlds". PhD Thesis. Yale University, 1989.

[9] Kristian J. Hammond. "Explaining and repairing plans that fail". In: *Artificial Intelligence* 45.1-2 (Sept. 1990), pp. 173–228.

[10] Gayane Kazhoyan and Michael Beetz. "Executing Underspecified Actions in Real World Based on Online Projection". In: *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS).* IEEE, Nov. 2019, pp. 5156–5163.

[11] Gayane Kazhoyan, Arthur Niedzwiecki, and Michael Beetz. *Towards Plan Transformations for Real-World Pick and Place Tasks.* Dec. 2018.

[12] Jaeho Lee, Marcus J Huber, Edmund H Durfee, and Patrick G Kenny. "UM-PRS: An implementation of the procedural reasoning system for multirobot applications". In: *Conference on Intelligent Robotics in Field, Factory, Service and Space (CIRFFSS 1994).* 1994, pp. 842–849.

[13] Severin Lemaignan, Anahita Hosseini, and Pierre Dillenbourg. "PYROBOTS, a toolset for robot executive control". In: *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS).* IEEE, Sept. 2015, pp. 2848–2853.

[14] Lorenz Mösenlechner. "The Cognitive Robot Abstract Machine A Framework for Cognitive Robotics". PhD Thesis. University of Bremen, 2016, p. 255.

[15] Lorenz Mösenlechner, Nikolaus Demmel, and Michael Beetz. "Becoming action-aware through reasoning about logged plan execution traces". In: *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, Oct. 2010, pp. 2231–2236.

[16] Jean-Baptiste Mouret and Jeff Clune. "Illuminating search spaces by mapping elites". In: *arXiv preprint arXiv:1504.04909* (2015).

[17] Armin Müller, Alexandra Kirsch, and Michael Beetz. "Transformational planning for everyday activity". In: *ICAPS 2007, 17th International Conference on Automated Planning and Scheduling*. 2007, pp. 248–255.

[18] Roger C Schank. *Dynamic memory: A theory of reminding and learning in computers and people*. cambridge university press, 1983.

[19] Jan Winkler. "Longterm Generalized Actions for Smart, Autonomous Robot Agents". PhD Thesis. University of Bremen, 2018.

[20] Jan Winkler, Moritz Tenorth, Asil Kaan Bozcuoglu, and Michael Beetz. "CRAMm - Memories for Robots Performing Everyday Manipulation Activities". In: *Advances in Cognitive Systems*. Vol. 3. 2014, pp. 91–108.