Where is a Cup and What is it Good for?

Crafting an ASP-based Commonsense Knowledgebase for Robotic Agents

Master Thesis of
Stefan Jakob
Student number:
31205450
submitted in the department
Distributed Systems

Primary Reviewer: Prof. Dr. Kurt Geihs
Secondary Reviewer: Prof. Dr. Bernhard Sick
Advisor: Stephan Opfer M. Sc.

July 4, 2017

University of Kassel
Faculty of Electrical Engineering and Computer Science
Summer Semester 2017
Affidavit

Eidesstattliche Erklärung:

Ich versichere, dass ich die Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind, sowie dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt wurde.

Affidavit:

I hereby declare that I wrote this thesis on my own and without the use of any other than the cited sources and tools and all explanations that I copied directly or in their sense are marked as such, as well as that the thesis has not yet been handed in neither in this nor in equal form at any other official commission.

Kassel, July 4, 2017
Abstract

Commonsense knowledge has a highly important role in everyday human life. Human beings rely on this knowledge while solving mundane tasks or during conversations, in which it is often omitted. An example of an everyday task is to fetch a cup. Every human in possession of commonsense knowledge knows that a cup can usually be found in a cupboard or on a shelf in the kitchen and therefore can easily solve the task. A robotic agent lacks this knowledge. In order to enable a robotic agent to perform everyday tasks, it needs to be equipped with commonsense knowledge and the ability to apply this knowledge to the given task, enabling it to solve the task. Fulfilling both requirements demands a combination of a commonsense knowledge source and a reasoning paradigm, which is able to cope with big knowledgebases. Therefore, we combine the commonsense knowledge source ConceptNet 5 and Answer Set Programming (ASP), which offers multi-shot solving and non-monotonic reasoning. Additionally, we introduce an algorithm that efficiently removes inconsistencies from the commonsense knowledge given by ConceptNet 5 and translates this knowledge into ASP. As an evaluation scenario, we created a typical household scenario containing an agent, which is supposed to clean up the household. Therefore, the agent has to find the typical locations of the given objects by relying upon its commonsense knowledge and its reasoning abilities. Furthermore, runtime tests have been conducted to evaluate the time efficiency of the introduced combination of ASP and ConceptNet. The results of these tests show that the creation of a commonsense knowledgebase differs in runtime depending on the number of concepts connected to a queried concept. Hereby, the results have varied from 0.37 s for small concepts that are connected to roughly 300 concepts up to 3.15 s for large concepts, which are connected to more than 2000 other concepts. Furthermore, the results show that the presented combination of ASP and ConceptNet is able to handle scenarios consisting of 100 different household objects, which are represented by 38 unique concepts.
## Contents

1 Introduction 1

2 Definition Of Task 5
   2.1 User Interface for Clingo and ConceptNet 5 5
   2.2 Running a Local Copy of ConceptNet 5 6
   2.3 Create a Consistent Commonsense Knowledgebase from ConceptNet 5 6

3 Foundations 9
   3.1 Answer Set Programming 9
   3.2 Expansions to Answer Set Programming by Clingo 14
   3.3 Module Property 15
   3.4 Answer Set Programming Query Mechanism 19
   3.5 ConceptNet 5 23

4 Related Work 27
   4.1 SenticSpace: Visualizing Opinions and Sentiments in a Multi-dimensional Vector Space 27
   4.2 The Glass Infrastructure 28
   4.3 RESI - A Natural Language Specification Improver 29
   4.4 How Flexible Is Answer Set Programming? An Experiment in Formalizing Commonsense in ASP 30
   4.5 Visual Common-Sense for Scene Understanding 31
   4.6 Answer Set Programming for Collaborative Housekeeping Robotics 32
   4.7 Differentiation from the Presented Related Work 33

5 Implementation of the KnowledgebaseCreator 35
   5.1 Graphical User Interface 35
   5.2 Command History and Command Structure 39
   5.3 ASP Solver Wrapper 42
   5.4 Automatic Satisfaction of the Module Property for Queries 43
   5.5 Running a Local Copy of ConceptNet 5 46

6 Consistent Knowledge from ConceptNet 5 51
   6.1 Influence of the ConceptNet 5 Edge Weight 51
   6.2 Possible Inconsistencies in ConceptNet 5 52
## 6.3 Removal of Inconsistencies and Representation in ASP

### 7 Evaluation

- **7.1 Evaluation Setup** ................................................. 61
- **7.2 Time Efficient Queries to CNS** ................................. 61
- **7.3 Comparison of Cached and Non-Cached Queries** ............ 64
- **7.4 Runtime Evaluation of the Inconsistency Removal** .......... 65
- **7.5 Runtime Evaluation of the ASP Knowledgebase** ............ 67
- **7.6 Application of the KnowledgebaseCreator in a Household Scenario** ................................. 68

### 8 Conclusion

- **8.1 Summary** .......................................................... 71
- **8.2 Future Work** ...................................................... 72

## List of Figures

## List of Listings

## Bibliography
1 Introduction

In order to solve everyday tasks, human beings heavily rely on their commonsense knowledge, which is, as stated in [9], knowledge about our environment that every human being is supposed to know. Reducing these everyday tasks to household chores, for example cleaning a room, preparing lunch, or to get a cup of coffee, shows how much human beings rely on this knowledge. Focussing the task of fetching a cup of coffee, let us consider the question asked in the title of this elaboration: “Where is a cup and what is it good for?”. This question can be easily answered by a human as shown in the Figures 1.1(a) and 1.1(b).

![Figure 1.1: Where is a cup and what is it good for?](image)

(a) Common location of a cup.  
(b) Common usage of a cup.

As you can see in these pictures, a typical location for a cup is a cupboard in the kitchen and a cup is typically used to serve hot beverages. Both information can be considered as commonsense knowledge. A human in possession of such knowledge will use this to fulfill the given task of fetching a cup of coffee. In order to understand the importance of commonsense knowledge, let us further break down this task. The first steps of this task require to enter the kitchen, to open a cupboard, and to fetch a cup. Hereby, the properties of a cup can be considered as commonsense knowledge, too. A cup is normally made from ceramic and is equipped with a handle. After fetching the cup, it has to be placed in the coffee machine. Since coffee is related to food & drinks, it can be found in the kitchen, as well. As you can see, the simple task of fetching a cup of coffee requires commonsense knowledge in order to be fulfilled. Furthermore, this simple task shows an interesting property of commonsense knowledge. As stated in [28], “commonsense knowledge spans a huge portion of human experience but is typically omitted from social communications”, which means that a huge amount of information is skipped while assigning a task to a human being. A
problem arises when such kind of tasks are given to a robotic agent, which is not equipped with commonsense knowledge. Without a detailed definition of the task, the agent would not be able to solve the task. To tackle this problem this elaboration provides access to the commonsense knowledge database ConceptNet 5, which represents commonsense knowledge as a directed hypergraph of concepts connected by weighted relations. In order to provide this knowledge to an agent, Answer Set Programming is used, which is a non-monotonic knowledge representation and reasoning formalism. Thus, this elaboration equips agents with commonsense knowledge and provides the capability to efficiently reason about it. This enables an agent to fulfill tasks like presented above and to answer the question: “Where is a Cup and What is it Good for?”.

Commonsense knowledge, as provided by the ConceptNet 5 database, can be a source of inconsistencies. Generally speaking, commonsense knowledge introduces a world equipped with inconsistencies for both humans and robotic agents. This world includes ambiguous concepts, incomplete knowledge, and false assumptions. An example for arising inconsistencies is the concept of a table. A table is a piece of furniture, which is normally made of wood, plastic, or metal. Furthermore, a table can be used to arrange data in rows and columns. Inconsistencies like this can often be hard to find for humans and even harder for robotic agents that are not able to extract the meaning of a symbol that is used to abstract an object in its environment. Since the commonsense knowledge is usually omitted during the process of setting a task, inconsistencies like these are not solved during this process, which can cause a false execution of the given task. Therefore, inconsistencies have to be prevented when creating a commonsense knowledgebase. Thus, this elaboration tries to eliminate as many inconsistencies as possible during the provision of commonsense knowledge. The focus is set on two possible sources of inconsistencies. The first source can be considered as badly supported or false assumptions, which are part of ConceptNet 5. These inconsistencies are caused by the way ConceptNet 5 was built. It was built by combining several knowledge sources, which resulted in weighted relations between the concepts. In these sources, Games with a Purpose were included that asked human users to enter commonsense knowledge, which introduced further knowledge to ConceptNet 5 but also introduced false statements given by the users of the games. The second source of inconsistencies is incorrectly supposed properties. These properties arise from the variety of knowledge introduced by ConceptNet 5. A suitable example to explain this type of inconsistencies is the concept of a plant. In general, plants can be split into two categories, a category of plants that keep their leaves during the year (evergreen) and a category that loses their leaves during a given period in the year (deciduous). A plant combining both contradicting properties (evergreen and deciduous) is difficult to imagine for a human being, which is equipped with commonsense knowledge. ConceptNet 5 represents contradictions like this via an Antonym relation that allows a human being that is able to understand the meaning of the Antonym relation by its commonsense knowledge to deduce that a plant can have only one of the contradicting properties. By
introducing ConceptNet 5 to a robotic agent that is not in able to understand the meaning of the Antonym relation, the agent could conclude from the knowledge presented in this database that such a plant exists, since both properties are connected to the concept of a plant. Therefore, this elaboration removes contradicting properties from concepts to prevent robotic agents from deriving false assumptions from the properties of a concept, resulting in a consistent commonsense knowledgebase that allows a robotic agent to reason about its environment and given tasks.

The remainder of this elaboration is structured as follows: The tasks of this elaboration are presented in Chapter 2. Furthermore, the necessary foundations to understand the concept and the implementation of this elaboration are given in Chapter 3. Additionally, Chapter 4 is an overview of related papers. This includes different applications of ConceptNet 5 as a source of commonsense knowledge and the use of Answer Set Programming as knowledgebases. The implementation of the given tasks is divided into two chapters. In Chapter 5, the creation of the graphical user interface and the installation of ConceptNet 5 are explained. In addition to this, the handling of inconsistent knowledge from ConceptNet 5 is shown in Chapter 6. The implementation presented in both chapters is then evaluated in Chapter 7. In conclusion, a summary of this elaboration and an outlook for future expansions are given in Chapter 8.
2 Definition Of Task

In this chapter, the tasks that have to be solved in the context of this master thesis are presented. In Section 2.1 the task to create a user interface for Clingo and ConceptNet 5 is described. Furthermore, in Section 2.2 the requirements for the interaction with ConceptNet 5 are presented and as a last point, the task for handling the commonsense knowledge of ConceptNet 5 is explained in Section 2.3.

2.1 User Interface for Clingo and ConceptNet 5

In order to interact with the Clingo ASP solver \cite{17, 18} and ConceptNet 5 \cite{48}, a graphical user interface (GUI) has to be created. This GUI, in the following chapters denoted as the KnowledgebaseCreator, should enable the user to use all language aspects of Answer Set Programming \cite{19}. These aspects include the functionality to add ASP programs to the solver, ground them, and solve them. Additionally, the ASP query mechanism presented in \cite{34} should be usable in this GUI. Since this query mechanism uses the old version 4.5.3 of Clingo, the query mechanism has to be adapted to the current version 5.2.0. Furthermore, the query mechanism must automatically satisfy the Module Property of ASP programs, in order to ease the creation of queries for the user. In addition to the general syntax of ASP, features provided by Clingo should be accessible for the user as well. This includes External Statements and Program Sections.

The created GUI should provide access to ConceptNet 5, which is a commonsense knowledge base representing its knowledge by using concepts and relations. The access to ConceptNet 5 includes a connection to the API and queries for complex relations between the concepts of ConceptNet 5. These queries should provide access to concepts and any combination of relations and concepts. By using these queries a robotic agent, for example, should be able to answer the question given in the title of this elaboration: “Where is a Cup and What is it Good for?”. In this case, the agent could formulate a query for the usage of a cup, utilizing the UsedFor relation of ConceptNet 5 in combination with the concept of a cup, which, for example, results in the commonsense knowledge that a cup is usually used for serving hot beverages. Still, the agent needs to know typical locations of a cup, in order to solve the asked question. This could be done by using the AtLocation relation in the query, which results in usual locations of a cup. These locations include tables, shelves, or
cupboards. By combining the resulting commonsense knowledge of both queries, the agent should be able to answer the question asked: A cup contains hot beverages and can normally be found on a table, a shelf or a cupboard. Since the agent is in possession of such knowledge, it should be able to fulfill everyday household tasks. For example, it could fetch a cup of coffee, in order to support humans even without a very detailed description of the given task.

Furthermore, the GUI should provide the functionality to save and load the progress of the current session. Therefore, a command history has to be provided, which enables the user to track the current workflow and in the case of an error, to undo the last command. To save the current state of work, this command history should be serialized and saved persistently in a file. This file should be loadable in order to continue from the last state of work.

2.2 Running a Local Copy of ConceptNet 5

The web API of ConceptNet 5 is limited to 600 queries per minute, respectively, 6000 queries per hour. Therefore, a local copy of ConceptNet 5 is necessary. Since ConceptNet 5 is a large database, the way of installing this database has to be evaluated. This evaluation should consider the amount of disk space used, the way of installing the database, and the interaction with the API.

2.3 Create a Consistent Commonsense Knowledgebase from ConceptNet 5

The third task of this elaboration is to create a commonsense knowledgebase from the knowledge of ConceptNet 5. In this task, the concepts have to be translated into ASP rules forming the commonsense knowledgebase. Since ConceptNet 5 is a very large commonsense knowledge database, which is combined from multiple sources, inconsistencies can arise. The inconsistencies could be caused by wrong statements from the sources or contradictory knowledge from different sources. This task includes finding and handling inconsistencies in the commonsense knowledge provided by ConceptNet 5. In this process, the following questions have to be considered:

1. Which kinds of inconsistencies can appear in ConceptNet 5?
2. Which inconsistencies can be detected automatically?
3. Which contradictions should not be included in the knowledgebase?

Considering these questions, this task includes evaluating, which kind of inconsistencies can be found automatically and to create an algorithm, which is able to remove these
2.3 Create a Consistent Commonsense Knowledgebase from ConceptNet 5

inconsistencies. An example for inconsistencies for robotic agents in ConceptNet 5 is a pair of contradicting properties, since a robotic agent could conclude that a concept has both contradicting properties, resulting in false assumptions about the concept. Therefore, the created algorithm should prevent inconsistencies like these. Furthermore, the commonsense knowledgebase extracted from ConceptNet 5 should be as broad as possible. Hence, the queries should return as many concepts as possible without adding inconsistencies.
3 Foundations

In this chapter, the foundations needed to understand this elaboration are presented. At first, the declarative logic programming approach Answer Set Programming (ASP) is presented in Section 3.1. Clingo, the Answer Set Programming solver used in this elaboration, expands the syntax of ASP by several features, which are shown in Section 3.2. Another important aspect of the Answer Set Programming paradigm is the Module Property that ensures the correct derivation of Stable Models. A definition of this property is given in Section 3.3. Furthermore, a query mechanism to interact with the ASP solver is presented in Section 3.4. As the last point, the commonsense knowledge database ConceptNet 5 (CN5) is presented in Section 3.5 which is used as a commonsense knowledge source in this elaboration.

3.1 Answer Set Programming

Answer Set Programming (ASP) is a declarative logic programming approach, tackling NP-search problems. As stated in [4], ASP can be seen as the result of research in the areas of knowledge representation, logic programming, and constraint satisfaction. The base of every ASP program are atoms, respectively predicates. Atoms are statements that can be either true or false. Furthermore, atoms and their negation (not) are called literals. With the help of these literals, rules are modeled. An example for a rule is given in Figure 3.1.

![Figure 3.1: Common example of an ASP rule.](image)

As you can see in this figure, a rule consists of atoms, respectively literals \((x_1, \ldots, x_k, y_1, \ldots, y_m, \text{not } z_1, \ldots, \text{not } z_n)\), which are predicates that can be either true or false. Further-
more, a rule consists of two parts. The first part of a rule is the head. In Figure 3.1, the head consists of the literals $x_1, \ldots, x_k$. The second part is the body, namely $y_1, \ldots, y_m$, not $z_1, \ldots$, not $z_n$. Additionally, the body can be divided into two parts, namely the positive part, which contains the positive literals and the negative part containing the negated literals.

The head of a rule can be derived if all positive literals in the body can be derived and no literal in the negative body holds. In this case, the default negation not is used as negation-as-failure. This means a negative literal is only considered to hold if the positive form of the literal can not be proven to hold. In addition to negation-as-failure, ASP also provides the notion of classical negation, which is expressed by a literal of the form: $-x$. Thereby, a classical negated literal is the complementary to the positive literal $x$, meaning $-x$ and $x$ cannot hold at the same time without rendering the ASP program inconsistent. Furthermore, ASP provides two special types of rules, which should be mentioned. The first special type is a rule without a body. These rules are unconditionally true and therefore are considered as facts. The second special type of rules are rules without a head, which are named constraints. A constraint is used to restrict the solution of an ASP program since only the constant false is derivable from the combination of literals in the body and therefore this combination should be forbidden in the solution. To show the usage of ASP, a commonly used example to demonstrate the functionality of ASP is presented in Listing 3.1:

```
1 bird(tweety).
2 bird(pingu).
3 flies(X) :- bird(X).
4 :- flies(X), -flies(X), bird(X).
```

Listing 3.1: Example of an ASP program.

Lines 1 and 2 of this example are facts, which state that both, tweety and pingu, are birds. Line 3 of this example is a rule, which expresses that all birds can fly. Hereby, $X$ is a variable, denoted by the capital letter, which is substituted with any element of the ASP program’s Herbrand universe [41]. The Herbrand universe of this example contains two constants, denoted by starting with a lower case letter. These constants are {tweety, pingu}. Furthermore, Line 4 is a constraint, which prohibits that a bird can fly and can not fly at the same time since both predicates would introduce an inconsistency to the ASP program. Solving this ASP program results in the following Answer Set: {bird(tweety), bird(pingu), flies(tweety), flies(pingu)}. If you now receive the information, that pingu is a special kind of bird, namely a penguin, which is not able to fly, the ASP program in Listing 3.1 is expanded by the following fact:

```
5 -flies(pingu).
```

Listing 3.2: Pingu is not able to fly.

The appearance of Line 5 in the example program violates the constraint in Line 4, which results in an empty Answer Set. This shows the non-monotonic property of Answer Sets.
If an ASP program is expanded, it is not guaranteed that all parts of the former solution stay part of the current solution. To avoid inconsistent programs like the current expanded example, Line 3 has to be exchanged by the rule presented in the following Listing 3.3:

```prolog
flies(X) :- not -flies(X), bird(X).
```

**Listing 3.3:** Closed World Assumption formulated in ASP

By exchanging Line 3 with this rule the ASP program is no longer inconsistent since it is no longer possible to derive that *pingu* is able to fly and therefore the constraint is no longer violated. Solving this ASP program will result in the Answer Set: `{bird(pingu), bird(tweety), -flies(pingu), flies(tweety)}`.

Apart from the syntax and semantics of ASP, the way an ASP program is solved is a very important aspect. In general ASP solvers, for example, Clingo [16–18], calculate the solution of an ASP program in two steps. The first step while solving an ASP program is called grounding, which, informally speaking, replaces variables inside the logic program with parts of the program’s Herbrand universe. A basic grounding algorithm is presented in [16] and is shown in Algorithm 1.

**Algorithm 1: NaiveGrounding** [16]

<table>
<thead>
<tr>
<th>Input</th>
<th>A safe logic program P</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>A ground logic program P’</td>
</tr>
</tbody>
</table>

```plaintext
D := ∅;
P’ := ∅;
repeat
   D’ := D;
   foreach rule ∈ P do
      B := positiveBody(rule);
      foreach θ ∈ Θ(B, D) do
         D := D ∪ {head(rule)θ};
         P’ := P’ ∪ {ruleθ};
      end
   end
until D = D’
```

This algorithm is used to ground a safe ASP program, which is an ASP program in which every variable in the head of every rule, appears positively in the body of the corresponding rule. Therefore, only the positive parts of the bodies of each rule have to be considered. In this algorithm B (bodies) and D (domain predicates) are sets of atoms, θ is a match of atoms appearing in B by ground atoms appearing in D and Θ(B, D) is a set of minimal matching atoms from B in D. In order to understand the functionality of this algorithm, it is demonstrated by the example shown in Listing 3.4.
Once the algorithm is initialized, the grounding process starts by entering Line 5 of Algorithm 1. To show the full functionality of this algorithm, Rule 4 of Listing 3.4 is selected first. In this case, B only consists of the atom bird(X). Since set D is still empty there is no possible match for the atoms occurring in B and therefore, Lines 7 till 10 are skipped. This marks the end of an iteration of the outer foreach loop and the next rule is selected. In this case, one of the three remaining facts, e.g. bird(tweety), is chosen. Since the body of a fact is an empty set, no a match is needed and the inner foreach loop is entered. In this loop, the fact is added to set D and the ground program P’, which has been empty so far, is expanded by the fact. This is done analogously for the remaining two facts. P’ then consists of the three facts: bird(tweety), bird(pingu), and bird(woody). Since the set D has changed in this process, the outer foreach loop is started again. Hereby, Rule 4 of Listing 3.4 is selected again. In comparison to the previous iteration, set D now contains three facts matching the body of Rule 4 and therefore, the rule heads with replaced variables, namely flies(tweety), flies(pingu), and flies(woody), are added to D. Additionally, P’ is expanded by the grounded rules. Again, the set D has changed and another iteration is triggered. In this iteration, all rules are grounded again, but this time, the set D remains unchanged. Since the set D has not changed, the algorithm stops by returning the ground instance of the given ASP program. As you can see, this is a very simple approach for the grounding procedure and many improvements can be made, e.g., checking for already grounded rules or removing rules containing predicates not appearing in any rule head. This is for example done by the solver Clingo and is described in Chapter 7 of [16].

As soon as the grounding is finished, the process of finding Stable Models of the ASP program is started. This process is named solving. In general, common SAT solving techniques like the Davis–Putnam–Logemann–Loveland (DPLL) algorithm [10] are used. This algorithm assigns truth values to the grounded atoms and in the case of inconsistent assignments uses backtracking to revert these assignments. Another commonly used algorithm is Conflict-Driven Clause Learning (CDCL) [3, 42], which is inspired by the DPLL algorithm. Both algorithms are used to solve problems presented in the conjunctive normal form [36] and both start with a deterministic search step, which is called unit propagation. In this step, clauses with only one literal are assigned the corresponding truth value to satisfy the literal. After this step three possible states of the solving process can occur: The first state is that all literals are assigned and the process stops by returning the result. In contrast to the first state, not all literals are assigned in the second state. Hereby, a literal is randomly selected and a
truth value is assigned, which can cause inconsistencies. If an inconsistency occurs, state three is reached, in which both algorithms differ. In the DPLL algorithm, the last assignment of a truth value is reversed and the opposite truth value is assigned. In contrast to this, CDCL analyses the conflict and jumps back to the point where the conflict originates. Furthermore, CDCL prevents already created conflicts by adding constraints. After solving all conflicts the solved program is returned by both algorithms. In ASP, the solutions for a program are referred to as Stable Models or Answer Sets.

Generally speaking, a Stable Model or Answer Set is a set of ground atoms, which either contains every atom belonging to the rule heads, or does not include all atoms of the positive part of the rule bodies, or contains atoms of the negative bodies of the rules. Furthermore, a Stable Model is as small as possible and consists only of atoms that are justified by the facts of the ASP program. Answer Sets or Stable Models have a special semantic, which is presented in \[20\]. During the generation of Stable Models, two cases have to be considered: The first case is a program without negation. In this case, there is only one Stable Model containing only the head atoms of the satisfied rules. The second case is a program \(P\) containing negative predicates. Hereby, a reduction \(P'\) of program \(P\) is formed by dropping the negated parts of \(P\). If any model of \(P\) is a model for \(P'\) it is called a Stable Model. This is illustrated in the example in Listing 3.5.

```
1 //Program P
2 bird(tweety).
3 bird(woody).
4 flies(X) :- bird(X), not -flies(X).
5
6 //Program P'
7 bird(tweety).
8 bird(woody).
9 flies(X) :- bird(X).
```

**Listing 3.5:** ASP program with its reduced version.

A possible solution for program \(P\) is the set \(S\): \{\texttt{bird(tweety), bird(pingu), flies(tweety), flies(pingu)}\}. In order to check if \(S\) is a Stable Model (Answer Set) for \(P\) the reduction of \(P\) regarding \(S\) has to be created. This is done by removing every negative body atom not appearing in \(S\). This results in the reduced program \(P'\). If the solution of \(P'\) is identical to the solution of \(P\), \(S\) is a Stable Model (Answer Set) of program \(P\), which is the case in this example. Otherwise, \(S\) is no Stable Model and therefore no solution for \(P\).

In addition to the syntax presented in this section, ASP provides further features like disjunctive rule heads or the use of aggregate functions. To obtain an overview of these features, the interested reader is advised to [19].
3.2 Expansions to Answer Set Programming by Clingo

In this elaboration, the ASP grounder and solver Clingo \[17, 18\] in version 5.2.0 was chosen. Clingo was selected since it introduces two expansions to the ASP syntax, which enables the change of the truth value of an atom and the division of ASP programs into smaller parts. The first expansion of the ASP syntax is named *External Statements* \[13, 14\]. *External Statements* are atoms, respectively, predicates that are marked with the keyword \#external. The general notion of *External Statements* is that their truth value is not known during the grounding process and therefore they stay part of the grounded program, even if there is no evidence that the *External Statement* is true or derivable. Furthermore, their truth value can be explicitly set to true or false, without an additional grounding step. An example for the usage of *External Statements* is presented in Listing 3.6.

```prolog
1 bird(tweety).
2 flies(X) :- bird(X), not -flies(X).
3 #external -flies(tweety).
```

Listing 3.6: Example for the usage of *External Statements*.

Compared to the initial example with `bird(tweety)`, this example is expanded by an *External Statement* named `flies(tweety)` in Line 3. As already stated, the *External Statement* stays part of the grounded program given to the solver. After the grounding process is finished, the truth value of the *External Statement* can be changed before the program is solved. If it is not assigned, the *External Statement* is assumed to be false. After solving this program, the resulting Stable Model consists of the following atoms: `{bird(tweety), flies(tweety)}`. Once the solving process is finished, the truth value of the *External Statement* can be changed to true, changing the Stable Model without an additional grounding step. The changed Stable Model then contains the following atoms: `{bird(tweety), -flies(tweety)}`. As you can see in this example, the use of *External Statements* allows changes of a grounded ASP program and therefore can influence the Stable Models.

*Program Sections* are the second expansion to the syntax of ASP made by Clingo and are used to structure an ASP program into segments, which can be grounded and solved independently. The order in which the *Program Sections* are grounded can influence the resulting Stable Models. An example for this is given in Listing 3.7. This example contains two *Program Sections*, namely `facts` and `rules`. The resulting Stable Model of this ASP program depends on the order in which the *Program Sections* are grounded. If the *Program Section* `rules` is grounded before `facts`, the resulting Stable Model contains only the atom `bird(tweety)`, since there was no match to replace `bird(X)` during the grounding. If the order of grounding is reversed, meaning `facts` is grounded before `rules`, the resulting Stable Model consists of the atoms `bird(tweety)` and `flies(tweety)`. Since `facts` has been grounded first there is a match for `bird(X)` in the `rules` *Program Section* and therefore the head of the rule in Line 4 can be derived.
3.3 Module Property

The Module Property is a central aspect of ASP, since violating this property will render the ASP program unsolvable. This property can be violated when two ASP programs are combined into a single one, which is done by giving them to the same solver instance. This, for example, happens during the queries presented in Section 3.4. Therefore, different
3 Foundations

definitions of this property and a Module itself are presented and discussed. Finally, a summary of the Module Property and a way to avoid a violation are shown.

Engineering an Incremental ASP Solver [15]

In this paper, a way of creating an incremental ASP solver is described, which builds a solution by incrementally combining ASP program parts to solve the complete program. These ASP program parts can be considered as Modules, which are the central part of the Module Property. A Module $\mathcal{P}$ consists of three sets $\mathcal{P}$, $I$ and $O$. The set $\mathcal{P}$ (program) consists of ASP rules. The set $I$ is considered as the input and $O$ as the output of $\mathcal{P}$. Both sets $I$ and $O$ are disjoint subsets of the universe $\text{grd}(A)$ ($I \subseteq \text{grd}(A)$). Furthermore, the relations between $I$, $O$, and $\mathcal{P}$ are defined in the following way: The atoms of $\mathcal{P}$ are a subset of the union of $I$ and $O$ ($\text{atom}(\mathcal{P}) \subseteq I \cup O$) and $O$ is a superset of the rule heads of $\mathcal{P}$, denoted by $\text{head}(\mathcal{P})$. Additionally, further constraints are introduced to the ground program $\mathcal{P}$. One of these constraints is the projection of $\mathcal{P}$ onto a set $X$ ($X \subseteq \text{grd}(A)$). This projection reduces the program to the atoms given in $X$: If a rule contains a negative literal $z$, which is not an element of $X$, the literal is removed. If $z$ is part of the positive body, the rule itself is removed. This projection results in a program whose body only consists of atoms given by $X$. Given this reduction, a Module $\mathcal{P}$ associated with input $I$ ($\mathcal{P}(I)$) is defined as follows: $\mathcal{P}(I) = (\text{grd}(\mathcal{P})|_{I \cup \text{head}(\text{grd}(\mathcal{P})|_{X})}, I, \text{head}(\text{grd}(\mathcal{P})|_{X}))$, where is $X$ is the union of $I$ and the heads of the grounded rules of $\mathcal{P}$ ($\text{head}($grd$(\mathcal{P}))$). By using this definition of a Module, a join of two Modules $\mathcal{P}$ and $\mathcal{Q}$ is performed as follows: $(\mathcal{P}(\mathcal{P}) \cup \mathcal{P}(\mathcal{Q}), I(\mathcal{P}) \cup (I(\mathcal{Q}) \setminus O(\mathcal{P})), O(\mathcal{P}) \cup O(\mathcal{Q}))$. Hereby, the union of both programs and both sets of output atoms is formed. The input sets cannot simply be united and have to be treated in a separate way. Thereby, the input of $\mathcal{P}$ is united with the input of $\mathcal{Q}$ except for all atoms in $I(\mathcal{Q})$, which are also part of $\mathcal{P}$’s output. By doing this, a recursion between the Modules is avoided. For example, a recursion between the two Modules $\mathcal{P}$ and $\mathcal{Q}$ is given, if an atom in the input of Module $\mathcal{P}$ is dependent on an atom appearing in the output of $\mathcal{Q}$ and vice versa. A recursion between both Modules $\mathcal{P}$ and $\mathcal{Q}$ would mean that both Modules would depend on each other and therefore cannot be considered as Modules anymore. This would violate the Module Property. A violation of this property would mean that the two Modules cannot be combined and therefore, the incremental solving process of this ASP solver would not be able to finish the solving process. Therefore, a violation of the Module Property should be prevented.

Answer Set Solving in Practise [38]

In this elaboration, a Module $\mathcal{P}$ is defined as a triple of sets ($\mathcal{P}$, $I$, $O$). $\mathcal{P}$ is a ground program over universe $\text{ground}(A)$ (ground atoms of $\mathcal{P}$) and both, $I$ and $O$, are disjoint subsets of $\text{ground}(A)$.
and are denoted as input and output, respectively. Furthermore, all atoms appearing in \( P \) are either part of \( I \) or \( O \) and all rule heads are part of \( O \). Given this definition of a Module, two Modules \( P \) and \( Q \) are compositional, meaning their join will not violate the Module Property if the following two conditions are met. The first condition is that the output sets of both Modules are disjoint, meaning that they do not share a common predicate. The second condition relies on strongly connected components \([40]\). A strongly connected component is a subset of a directed graph, in which every vertex is reachable by any other vertex in this subset. In order to check this condition, all strongly connected components of the union of \( P \) and \( Q \) (SCC) have to be considered. If any strongly connected component in SCC has a non-empty intersection with both sets \( O(P) \cap SCC \neq \emptyset \) and \( O(Q) \cap SCC \neq \emptyset \), this condition is violated, resulting in a violation of the Module Property. A strongly connected component, which is part of both output sets would mean that there is a recursion between both Modules. As stated above, a recursion between two Modules \( P \) and \( Q \) is given, when their input and output sets depend on each other. This violates the Module Property and therefore prevents solving the combination of both Modules.

**Modular Equivalence for Normal Logic Programs** \([33]\)

A Module \( P \) is a triple of sets \((P, I, O)\) with the following properties: \( P \) is a finite set of rules consisting of a head, a positive and a negative body. \( I \) and \( O \) are disjoint sets of propositional atoms and the rule heads of \( P \) are disjoint to \( I \). Furthermore, there is no restriction to the size of \( I \) and \( O \), both sets can contain atoms that are not part of \( P \). In order to define the Module Property, the positive dependency graph of \( P \) \( \text{Dep}^+(P) \) is necessary. A propositional atom \( x \) depends positively (\( \leq \)) on a propositional atom \( y \) if \( y \) is part of a rule containing \( x \) in the positive body. By using this relation \( \text{Dep}^+(P) \) is formed: \( \text{Dep}^+ = \{<x,y> | x,y \in Hb(P), x \leq y\} \), where \( Hb(P) \) is the Herbrand base of program \( P \). The Module Property is then defined by utilizing the strongly connected components extracted from \( \text{Dep}^+ \) allowing the combination of two Modules. Two Modules \( P = (P(P), I(P), O(P)) \) and \( Q = (P(Q), I(Q), O(Q)) \) are compositional if there is no positive recursion between these Modules. Hereby, a positive recursion between two Modules is given if there is any strongly connected component \( C \), whose intersection with both Modules is not empty \((C \cap O(P) \neq \emptyset \) and \( C \cap O(Q) \neq \emptyset \)). By using the definition of strongly connected components the violation of the Module Property is easily visible. If there is a strongly connected component that has a common part with both Modules \( P \) and \( Q \), there is at least one output atom \( x \) in \( P \) that positively depends on an output atom \( y \) in \( Q \). Since both atoms are part of a strongly connected component \( y \) depends positively on \( x \), too. Thus, resulting in a positive recursion, which violates the Module Property and renders the program unsolvable. Furthermore, to understand the influence of the Module Property and its violation, the Stable Models of each Module have to be considered, as stated in \([33]\). In order to form the Stable Models of the
3 Foundations

composition of both Modules, their Stable Models have to be combined. Hereby, the Stable Models have to be compatible, meaning they share the same common, even empty, part of the Herbrand Base of the corresponding other Module. A combination of two Modules $P$ and $Q$ is done in the following way: $(P(P) \cup P(Q), I(P) \setminus O(P) \cup (I(Q) \setminus O(P)), O(P) \cup O(Q))$.

In order to understand the effects of a violation of the Module Property, let us consider the following example, which is adapted from $[33]$ and consists of three Modules that violate the Module Property, since they form a positive recursion. These Modules are $P = (\{b :- not c.\}, \{c\}, \{\} )$, $Q = (\{c :- not a.\}, \{a,c\}, \{\} )$, and $R = (\{a :- not b.\}, \{b\}, \{a\} )$. Furthermore, their possible Stable Models are $SM(P) = \{b\}, \{c\}$, $SM(Q) = \{a\}, \{c\}$, and $SM(R) = \{a\}, \{b\}$. Their Herbrand Bases are $Hb(P) = \{b,c\}$, $Hb(Q) = \{a,c\}$, and $Hb(R) = \{a,b\}$, respectively.

In order to find the Stable Model of the join of these three Modules, a combination of Stable Models from each set of Stable Models $SM(P)$, $SM(Q)$, and $SM(R)$ has to be found. Therefore, all Models have to be checked if they are compatible. The first compatible sets of Models are $\{a\}$ and $\{c\}$ of $R$ and $P$, since $a$ is not part of and $Hb(P)$ and $c$ is not part of $Hb(R)$, they share the same part of the corresponding Herbrand Base. The second compatible set of Models are $\{b\}$ and $\{b\}$ of $R$ and $P$. Hereby, the common part of the corresponding Herbrand Base is the set $\{b\}$ for each Module. Any other combination of Models is not compatible.

For example, let us consider Model $\{b\}$ of Module $P$ and $\{c\}$ of Module $Q$. Forming the intersection of $\{b\}$ with $Hb(Q)$ results in an empty set, while the intersection of $\{c\}$ with $Hb(P)$ results in the set $\{c\}$, thus these set are not compatible. The other combinations of Models can be checked analogous to this. Since no combination of Models from $SM(P)$, $SM(Q)$, and $SM(R)$ can be found there is no Stable Model for the combination of these Modules, which is caused by a recursion these Modules. Thus, violating the Module Property has to be prevented, in order to obtain Stable Models from a composition of Modules, which possess a Stable Module if solved on their own.

Summary

To sum up, a Module itself is defined in the presented papers as a triple of sets, including the program $P$, the input atoms $I$, and the output atoms $O$. Additionally, $O$ is the center part to comply with the Module Property. During the combination of two Modules $P$ and $Q$, the output of one Module is used as the input for the other Module. This can cause a violation of the Module Property, namely if there is a strongly connected component containing atoms from the output of both Modules. For example, the output of $P$ contains an atom $z$, which depends on an atom $x$ in $Q$ and vice versa. Since both atoms depend on each other, they form a strongly connected component between both Modules, thus violating the Module Property. For example, Module $P$ is combined with Module $Q$. Therefore, the output from $P$ is used as the input of $Q$. Since the output of $Q$ contains an atom that depends positively on an atom in $P$, the output of $Q$ has to be the input of $P$, creating a positive recursion between these
Modules, thus violating the Module Property. In order to understand the effects of violating the Module Property, the Stable Models of each Module have to be considered. Hereby, each Module has its own Stable Models, which form the Stable Models of the resulting combination of the Modules. If Modules are combined that form a positive recursion, their Stable Models are not compatible \[33\], which prevents the creation of a Stable Model for the combination. Since no Stable Model for the combination of the Modules can be found, this ASP program is unsolvable and the Stable Models of each Module do not apply anymore. In order to prevent this, a violation of the Module Property has to be prevented, when combining several Modules, since a violation of this property would result in a loss of Stable Models and an unsolvable ASP program.

An approach to avoid the violation of the Module Property was proposed by Torsten Schaub in \[37\]. Since a violation of the Module Property can only occur in the output sets of a Module, which consists of the rule heads of the corresponding Modules, a prevention has to be applied to these sets. A possible prevention of a violation is using unique rule heads. By using unique rule heads no positive recursion between the Modules can be formed and both sets can not share a common predicate, thus the Module Property cannot be violated.

### 3.4 Answer Set Programming Query Mechanism

Clingo itself does not provide a mechanism to formulate queries in ASP. To cope with this, a query mechanism is presented in \[34\]. This query mechanism includes two kinds of queries. One kind is used to filter the Stable Models for a set of predicates and one kind is used to alter the current program of the solver. The first kind of query is named \texttt{ASPFactsQuery} and is used to filter Stable Models for a given set of atoms or predicates. Generally speaking, the \texttt{ASPFactsQuery} is a search algorithm applied to the Stable Models of the ASP program. The following algorithms (Algorithm 2, Algorithm 3) demonstrate how this search is performed.

Algorithm 2 is the main part of the \texttt{ASPFactsQuery} and computes the matches between the queried predicates and the predicates belonging to the Stable Models. The matches are then returned in a map containing the queried predicates as keys and a list of fitting predicates from the Stable Models as values. After the initialization of the map \(M\) in Line 1, each predicate is going to be compared with the Stable Model. Therefore, it is checked if the domains of the Stable Model contain the domain of the predicate, which is the name of the predicate, its sign, and its arity. If the domain is not found in the Stable Model, the next predicate is chosen, since the Stable Model does not contain any predicate that could match the queried predicate. If instead the domain is found, the predicates belonging to the domain are compared to the query predicate. This is done by Algorithm 3 (\texttt{checkMatchValues}). If this algorithm returns true, the domain predicate is added to the corresponding key in map \(M\). Once all query predicates have been checked, Algorithm 2 is finished and the map is returned.
to the caller of the \texttt{ASPFactsQuery}.

Algorithm 2: Construct Query Answer

\begin{verbatim}
Input : A Stable Model
Output: A map matching model predicates to the query predicates

1 Map M
2 foreach predicate in Query do
3     M.emplace(predicate, ∅)
4     D := Find Predicate Domain
5     if D is empty then
6         continue
7     end
8     foreach domainPredicate in D do
9         if checkMatchValues(predicate, domainPredicate) then
10            M.at(predicate).add(domainPredicate)
11        end
12     end
13 end
14 return M
\end{verbatim}

Algorithm 3: checkMatchValues(Predicate first, Predicate second)

\begin{verbatim}
Input : Two Predicates to be compared
Output : Boolean value corresponding to the match

1 if first.name != second.name then
2     return false
3 end
4 if first.numberOfArguments != second.numberOfArguments then
5     return false
6 end
7 for int i = 0; i < first.numberOfArguments; i++ do
8     if first.argument(i).name == wildcard then
9         continue
10    end
11 if both predicates are not constants then
12    if !checkMatchValues(first.argument(i), second.argument(i)) then
13        return false
14    end
15 end
16 if first.argument(i) != second.argument(i) then
17    return false
18 end
19 end
20 return true
\end{verbatim}

The \texttt{checkMatchValues} algorithm receives two predicates as input that are to be compared.
Since solely predicates of the same domain are given as input to this algorithm, only equally named predicates with an equal number of arguments are given to this algorithm. Therefore, the checks for different names in the Lines 1 to 3 and different numbers of arguments in the Lines 4 to 6 are not needed in the first place but are necessary if this algorithm is called recursively. The last parts of predicates from the same domain, which still can differ are the arguments itself. Therefore, all arguments are compared in Lines 7 to 19. The first step in these lines checks if an argument of the queried predicate is a special wildcard string, which symbolizes that no specific value is given and that it matches any other value. Hence, this argument is skipped and the next argument is compared. If an argument is a predicate itself, the algorithm is called recursively. In this case, the names of the predicates and their number of arguments can differ. If the argument is not a wildcard string and no predicate, the names of the arguments are compared. If these names do not match, the algorithm returns false. Finally, if none of the mentioned criteria is violated, the predicates match and true is returned.

In order to demonstrate the functionality of both algorithms let us consider the following example of a Stable Model, which consists of typical locations and usages of objects in a household. The model consists of the following predicates:

{location(redCup, cupboard), location(blueCup, cupboard), location(knife, drawer), location(spoon, drawer), location(fork, drawer), usedFor(redCup, drink), usedFor(blueCup, drink), usedFor(spoon, eat), usedFor(fork, eat), householdObject(fridge)}.

The domains in this example are location/2, usedFor/2, and householdObject/1. A possible ASPFactsQuery could be the location of the red cup, which is formulated by the predicate location(redCup, wildcard). The corresponding domain to this query predicate is location/2, which contains five predicates: {location(redCup, cupboard), location(blueCup, cupboard), location(knife, drawer), location(spoon, drawer), location(fork, drawer)}. From this set, the only matching one is location(redCup, cupboard), which is then returned to the caller of this query.

The second kind of a query is the ASPVariableQuery, which is used to alter an ASP program. This is done by adding additional facts and rules to an ASP program. Therefore, the ASPVariableQuery uses Algorithm 4 to alter the ASP program and to get the resulting representation of the rule heads in the Stable Models. Hereby, this query distinguishes three kinds of rules. The first kind of rule is the queryRule, whose results are returned to the caller. The second kind of these are normal ASP rules used to alter the queried ASP program and the third kind are facts that are added to the program. Furthermore, this kind of query has a lifetime. Once this lifetime has expired, the ASPVariableQuery removes itself from the ASP program and restores the previous ASP program.

The first step in Algorithm 4 prepares the later returned map M by adding each predicate of the queryRule’s head. Afterward, a unique Program Section is created by Algorithm 5, which is then added to the current ASP program and is grounded in the next step. This unique Program Section contains an External Statement, which has to be set to true in order
Algorithm 4: ASPFactsQuery
\begin{verbatim}
Input : queryRule, Set of Rules, Set of Facts, LifeTime
Output : Matching models to queryRule head
Map M
foreach predicate in queryRuleHead do
  M.emplace(predicate, φ)
end
P = CreateProgramSection(unique ID, query)
gound(P)
assignExternal(P.uniqueExt, True)
solve(P)
LifeTime-
if LifeTime == 0 then
  Restore ASP program
end
return M
\end{verbatim}

The first step of ASPFactsQuery is to derive the query parts. In the next step, the current ASP program is solved and the Stable Models containing the queryRule predicates are emplaced in map M. Furthermore, this marks the end of a query cycle and therefore, its lifetime is reduced. If the lifetime is reduced to zero, the External Statement is set to false and released, removing the query from the current ASP program. As a last step, the mapping of the head of the queryRule to the corresponding Stable Model is returned to the caller. The central part of this algorithm is Algorithm 5 which creates a unique Program Section formulating the query.

Algorithm 5: CreateProgramSection
\begin{verbatim}
Input : Unique ID and Query
Output : Unique Program Section
Program Section P
P.append(#program “query“ + uniqueld)
P.append(#external “extQuery“ + uniqueld)
P.append(expand(query.QueryRule))
foreach rule in query.Rules do
  P.append(expand(rule))
end
foreach fact in query.Facts do
  P.append(expand(fact))
end
\end{verbatim}

The first step of CreateProgramSection creates the Program Section containing a unique ID and a unique External Statement, which is going to be added to every rule body. In the next step, the queryRule is added to the Program Section and its body is expanded by the unique External Statement, which allows removing this rule as soon as the lifetime has expired. The same procedure is done to every other rule besides the queryRule. Facts have to be treated in
a special way. Since facts are a special kind of rule without a body, they have to be expanded by an artificial body consisting of the unique External Statement.

To understand how Algorithm 5 creates a unique Program Section, let us consider an example, which could be part of a simple navigation task in a household scenario. A common goal for a navigation task is to find a way from a start point, normally the current position to a destination or goal point, which could be a room in a household scenario. Furthermore, a predicate is necessary to express that two waypoints are reachable from each other. A possible ASPVariableQuery in this kind of situation would be a query setting the start and end point of the navigation and additionally check their reachability. This results in an ASPVariableQuery containing a query rule of the form: goalReachable(X) :- reachable(X, Y), start(Y), goal(X). Additionally, the query would contain two facts: start(kitchen), goal(livingRoom). Assuming that this is the first query added to the ASP program, the unique query ID is set to 1. With this information, the Algorithms 4 and 5 are used to create a Program Section to be added to the ASP program. The resulting Program Section is shown in Listing 3.9.

```
1 #program query1.
2 #external extQuery1.
3 goalReachable(X) :- reachable(X, Y), start(Y), goal(X), extQuery1.
4 start(kitchen) :- extQuery1.
5 goal(livingRoom) :- extQuery1.
```

Listing 3.9: ASP query used for navigation.

### 3.5 ConceptNet 5

ConceptNet 5 (CN5) is a multi-language knowledgebase presented in [46-48]. Furthermore, CN5 is a semantic hypergraph that contains commonsense knowledge expressed in natural language. This knowledge is a collection from different sources like the Open Mind Common Sense project, Wiktionary, WordNet 3.0, DBPedia and Wikipedia. The CN5 hypergraph consists of nodes representing concepts of natural language. A concept (/c) is a combination of several fields. These fields include a term, meaning an expression in natural language, a language tag, which is a country code in ISO 639-1 [25], and a sense label. The sense label expresses, which kind of word the concept is. Thereby, words are distinguished in nouns (/n), verbs (/v), adjectives (/a) and adverbs (/r). Furthermore, each concept is represented by a unique URI that contains the information given in the fields. For example, the URI for the verb run is /c/en/run/v. Besides the concepts, the hypergraph consists of edges.

---

1 Website: https://www.media.mit.edu/research/groups/5994/open-mind-common-sense
2 Website: https://de.wiktionary.org
3 Website: https://wordnet.princeton.edu/
4 Website: http://wiki.dbpedia.org/
5 Website: https://www.wikipedia.org/
Edges contain the sources that were used to create it, e. g., WordNet 3.0, its weight, and the concepts (start and end) connected to it. Hereby, the weight is a sum calculated from specific weights derived from the sources of the edge. Furthermore, each edge contains a relation denoting the meaning of the edge. The relations are either automatically extracted from sources like Wikipedia, which can be in any language or are part of a given set of English base relations. Table 3.1 is a collection of these English base relations and their meaning if it is placed between two Concepts A and B.

<table>
<thead>
<tr>
<th>Relation</th>
<th>Meaning</th>
<th>Relation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>IsA</td>
<td>A is a kind of B</td>
<td>SymbolOf</td>
<td>A represents B</td>
</tr>
<tr>
<td>UsedFor</td>
<td>A is used for B</td>
<td>ReceivesAction</td>
<td>A can be B</td>
</tr>
<tr>
<td>NotUsedFor</td>
<td>A is not used for B</td>
<td>HasPrerequisite</td>
<td>A requires B</td>
</tr>
<tr>
<td>CapableOf</td>
<td>A is able to B</td>
<td>MotivatedByGoal</td>
<td>You would A because you want B</td>
</tr>
<tr>
<td>NotCapableOf</td>
<td>A is not able to B</td>
<td>CausesDesire</td>
<td>A would make you want B</td>
</tr>
<tr>
<td>Desires</td>
<td>A wants to B</td>
<td>MadeOf</td>
<td>A is made of B</td>
</tr>
<tr>
<td>NotDesires</td>
<td>A does not want to B</td>
<td>DefinedAs</td>
<td>A is defined as B</td>
</tr>
<tr>
<td>Causes</td>
<td>The effect of A is B</td>
<td>AtLocation</td>
<td>A is located at B</td>
</tr>
<tr>
<td>HasFirstSubevent</td>
<td>The first thing happening for A is B</td>
<td>LocatedNear</td>
<td>You are likely to find A near B</td>
</tr>
<tr>
<td>HasSubevent</td>
<td>One of the things happening for A is B</td>
<td>CreatedBy</td>
<td>A is created by B</td>
</tr>
<tr>
<td>HasLastSubEvent</td>
<td>The last thing happening for A is B</td>
<td>HasA</td>
<td>A has B</td>
</tr>
<tr>
<td>HasProperty</td>
<td>A is B</td>
<td>PartOf</td>
<td>A is part of B</td>
</tr>
<tr>
<td>NotHasProperty</td>
<td>A is not B</td>
<td>SimilarTo</td>
<td>A is similar to B</td>
</tr>
<tr>
<td>HasContext</td>
<td>A appears in the context of B</td>
<td>RelatedTo</td>
<td>There is any kind of relation between A and B</td>
</tr>
<tr>
<td>Synonym</td>
<td>A is a synonym of B</td>
<td>Antonym</td>
<td>A is an antonym to B</td>
</tr>
</tbody>
</table>

**Table 3.1:** CN5 base relations [48].

By using the concepts in combination with edges and relations a complex hypergraph is formed, expressing the commonsense knowledge in CN5. Reconsidering the title of this elaboration, “Where is a cup and what is it good for?”, an answer by CN5 to this question is given in Figure 3.2. Since the concept of a Cup is connected to many other concepts, only a subset of edges and their corresponding weight is shown.

In this figure, the concept of a Cup is marked green and the edges are denoted with their relation and weight. Furthermore, the question “Where is a cup?” is answered by the concepts marked in blue. This includes the concept of a Table (weight 4.0), a Shelf (weight 3.4), a Dishwasher (weight 2.8), and a Kitchen (weight 2.0). Furthermore,
3.5 ConceptNet 5

Figure 3.2: Subgraph of CN5 showing the concept of a Cup.

the question “What is it good for?” is answered with the concepts marked in orange. A Cup is capable of Holding Liquids (weight 6.9), is used for Measuring (weight 3.4), and for Drinking (weight 3.4). Additional concepts related to a Cup are marked in gray.

In order to interact with the hypergraph, CN5 provides a REST API [30], which allows the use of the CN5 database and its query cache. A query to this API starts with the following URL: http://api.conceptnet.io/query?. To formulate queries, this URL can be expanded with six fields. These fields include start, end, rel, node, other, and sources. Start and end determine the start, respectively, end concept of the edge. Rel determines the relation queried and sources restricts the answer to a given set of sources. Node and other are used to set two concepts but do not state their position. Using these fields, a query for the location of a Cup is created in the following way: http://api.conceptnet.io/query?start=/c/en/cup&rel=/r/AtLocation, which is then given to CN5 via an HTTP GET request. The answer of CN5 to this query is a JSON [8] string containing the edges, which start from the concept of a Cup and are connected via the AtLocation relation to any other concept. Listing 3.10 is an excerpt from the returned edges. The first line (“@id”) of this excerpt is an ID representing the query given to CN5. Furthermore, this excerpt consists of one of the returned edges from CN5 containing the relation AtLocation (“rel”: {“@id”: ”/r/AtLocation”,“label”: ”AtLocation”}), the start concept denoted by the tag “start”, and the end concept denoted by the tag “end”. Furthermore, the
3 Foundations

dataset ("dataset") from which this edge originates, a list of sources ("sources"), and the weight ("weight") of the edge are given. In this case, the list of sources is only an excerpt from the complete list, which in total consists of seven sources supporting this edge. As the final point, the "surfaceText" is returned by CN5, which is a description of this edge in natural language. Besides the presented edge 22 other edges are returned, denoting all possible locations of a Cup given by CN5.

Listing 3.10: Excerpt from the edges returned by CN5 for the location of a Cup.
4 Related Work

The papers presented in this chapter set the context of this work and its implementation, which is further denoted as the KnowledgebaseCreator. This includes the use of ConceptNet 5 as a resource for commonsense knowledge (Sections 4.1, 4.2, and 4.3), for example, for finding the emotional context of texts or for providing information on public displays. Furthermore, Answer Set Programming is used to model knowledgebases and to express commonsense knowledge (Sections 4.4 and 4.5). Additionally, a paper is presented in Section 4.6, which combines ASP and ConceptNet. As a last point, Section 4.7 is a differentiation of the presented related work.

4.1 SenticSpace: Visualizing Opinions and Sentiments in a Multi-dimensional Vector Space

Cambria et al. present in [5] a way of perceiving emotions with the help of commonsense knowledge. In this paper, ConceptNet 5 is used as a commonsense knowledge source providing the background to understand texts. Furthermore, WordNet-Affect [49] is used as a source of affective knowledge. By combining these knowledge sources AffectNet is created, which links commonsense concepts to emotions. In order to extract emotions and opinions from free text, three modules are presented. A module for skimming the text, a module for extracting concepts, and the SenticSpace module. SenticSpace is derived from AffectNet by clustering the concepts and coloring them by a given scheme. The resulting space of concepts is shown in Figure 4.1. As you can see in this figure, SenticSpace locates concepts, which are related to the same emotion close to each other and colors them in the same way. For example, the blue cluster in the center of this figure consists of concepts related to music and is assigned the emotion “amazement”. The color scheme of SenticSpace is defined by the Hourglass of Emotions presented in [6].

By using SenticSpace, Cambria et al. address a current problem in the fields of Opinion Mining and Sentiment Analysis [35]. Hereby, the problem arises that opinions and affective states have to be explicitly given in the texts, leaving out texts without explicit statements. SenticSpace addresses this problem by combining commonsense knowledge from CN5 with affective knowledge by creating a space that maps concepts in natural language with an
affective adjective, allowing to extract opinions and feelings from texts in which they are not explicitly stated.

SenticSpace and the 
KnowledgebaseCreator can be compared in the use of CN5. Both elaborations use it as a source of commonsense knowledge in order to solve the problem they are addressing. In the case of SenticSpace, the concepts extracted from CN5 are mapped onto affective knowledge to solve the problem of automatically assigning opinions to texts, in which they are not explicitly stated. In contrast to this, the 
KnowledgebaseCreator set its focus on providing commonsense knowledge to robotic agents instead of focussing on emotions and opinions. Furthermore, the 
KnowledgebaseCreator provides an ASP-based commonsense knowledgebase, which supports robotic agents in automatically solving everyday tasks, which could not be fulfilled without this knowledge. As a last point, the representation of both elaborations differs, since SenticSpace relies on a vector-based approach and the 
KnowledgebaseCreator relies on a symbolic knowledge representation.

4.2 The Glass Infrastructure

Havasi et al. [21] present an interactive information kiosk (Glass Infrastructure) for the MIT Media Lab[1]. This kiosk provides visitors of this laboratory support to plan their trip, to get additional information, and to store their plans via RFID tags. To create such a kind of system, a knowledge and information base is needed. In the case of the Glass Infrastructure, Havasi et al. used short project descriptions provided in the Media Lab. Before these project

[1]Website: https://www.media.mit.edu/
descriptions are used in the system, they are preprocessed. This includes removing stop words and finding terms and phrases, which appear in each document using a predecessor of CN5. Furthermore, rarely appearing words are dropped and a matrix is formed linking the terms and phrases to concepts. In the last step, this matrix is combined with CN5 to extract commonsense knowledge regarding the specific projects in order to show the extracted concepts to the visitor and to provide support for their trip.

By creating the Glass Infrastructure, Havasi et al. try to support visitors of places equipped with a high amount of interesting items, artifacts, or projects, for example, museums or the MIT Media Lab. In these cases, the Glass Infrastructure provides support to a human user by providing additional information to visited objects or projects by relying on commonsense knowledge extracted from CN5. Hereby, the Glass Infrastructure automatically provides concepts to the user, which are related to the objects or projects they have already visited and therefore help the users to find further projects they could be interested in. This can be compared to the problem, which the KnowledgebaseCreator addresses. The KnowledgebaseCreator is used to provide further information on concepts a robotic agent needs in order to fulfill its given task. A contrast between both elaborations is given in the way the extracted concepts are treated. In the case of the Glass Infrastructure, the extracted concepts are presented to the help a user by planning a trip. In contrast to this, the KnowledgebaseCreator creates a symbolic knowledgebase, allowing the robotic agent to reason about the extracted concepts and to solve tasks containing unknown symbols.

4.3 RESI - A Natural Language Specification Improver

The Requirements Engineering Specification Improver (RESI) presented in [26] is a tool used for solving typical problems in Requirements Engineering [27]. The problems include ambiguous words, nominalization, similar meanings, and incompletely specified process words. To tackle these problems, RESI provides four steps in order to support the user of this tool. These steps are shown in Figure 4.2. The first step is to import the specification into RESI. The second step is used to tag words with their part of speech, meaning nouns or verbs, and their base form. The third step of RESI uses commonsense knowledge sources, which include CN5, to detect problems in the specification and mark them for the user. The last step is to export the specification alongside the suggested improvements made in step three.

In comparison to the KnowledgebaseCreator, RESI does not use CN5 to create a knowledgebase. Instead, CN5 is used to determine concepts of similar meaning or ambiguous words. In comparison to this elaboration, this can be seen as handling inconsistent knowledge. In this case, inconsistencies can arise in the understanding of words, therefore ambiguous words should be replaced in case of RESI, in order to provide an unambiguous specification.
This can be compared with the removal of inconsistencies in the KnowledgebaseCreator. Furthermore, the queries to CN5 of both elaborations can be compared. Given the way CN5 is used in RESI, the queries to CN5 are reduced to a set of synonymic relations like Synonym and DefinedAs. In contrast to this, the KnowledgebaseCreator provides access to all English base relations.

### 4.4 How Flexible Is Answer Set Programming? An Experiment in Formalizing Commonsense in ASP

Balduccini presents the process of solving a riddle that involves adding commonsense knowledge to an ASP program in [2]. The riddle itself is about two cowboys who want to marry the daughter of a rancher. The rancher suggests a race in which the last who will reach the goal line will marry his daughter. Since this race could last very long, the cowboys decide to ask a wise man about the race. The wise man tells them four words and the cowboys are ready to race. What words does the wise man tell to the cowboys?

To solve this riddle, Balduccini presents a formalization of this riddle, which has to rely on a formalization of commonsense knowledge in order to be consistent. An example for the need of commonsense knowledge is that if both cowboys cross the goal line at the same time, both are first, resulting in a loss for both. By applying rules to express further commonsense knowledge, the riddle can be solved by switching the horse with the opponent if you have the faster horse. Thus resulting in the four words the wise man told the cowboys: “Take each other’s horse”. This shows that commonsense knowledge is important to solve riddles and different kinds of everyday problems.
4.5 Visual Common-Sense for Scene Understanding

Both, this paper and the KnowledgebaseCreator share a common goal since they try to solve tasks by providing commonsense knowledge. A difference between both is given in the way the commonsense knowledge is introduced. Balduccini introduces commonsense knowledge manually, in order to solve the presented riddle. In contrast to this, the KnowledgebaseCreator provides automatically extracted knowledge from CN5. Furthermore, the problem that both elaborations address can be compared. A simple household task, e.g., to fetch a cup can be considered as solving a riddle for a robotic agent, which the agent is not able to solve unless it is expanded by further knowledge. To solve this riddle, the possible locations of a cup have to be added to the knowledgebase of the agent, which could either happen manually or automatically by relying on a commonsense knowledge source.

4.5 Visual Common-Sense for Scene Understanding

Aditya et al. address in [1] the problem of combining a visual system perceiving symbols with a reasoning approach involving commonsense knowledge. Therefore, they present a system, which is capable of reasoning about visual perception of actions using ASP and commonsense knowledge. Hereby, actions were tracked using a Kinect [53] depth camera and are formulated as ASP predicates containing time stamps. The predicates include the start and end of an action, the action itself, and the object the action has been performed on. By using these relations, an ASP program is formed, which then is used to recognize the activity. In this case, commonsense knowledge can be applied in two ways. The first way is to express the general structure of actions, e.g., that they can be divided into subactivities making the recognition of activities easier. The second application of commonsense knowledge is used to learn the general structure of actions, for example, their temporal relation. By using the relations extracted from the Kinect video stream and the predicates given from commonsense knowledge, a symbolic activity recognition is created. An example of this process is given in Figure 4.3.

This example shows the extraction of objects and activities from a given picture. The objects hereby include a bowl, a knife, and a piece of tofu. Furthermore, the activity of cutting is assumed. These objects and activities are then formulated in ASP predicates, which then are manually expanded by commonsense knowledge, e.g., that a hand is holding the knife while cutting the piece of tofu.

This can be compared to the application of the KnowledgebaseCreator since it uses a set of symbols to define a task, which are expanded by commonsense knowledge. Furthermore, both the system presented by Aditya et al. and the KnowledgebaseCreator use ASP as a reasoning mechanism. A contrast between both elaborations is given in the way the commonsense knowledge is added to the corresponding knowledge base. In the case of [1], the knowledge is added by manually forming ASP rules expressing knowledge about the extracted symbols.
In comparison to this, the *KnowledgebaseCreator* extracts the commonsense knowledge needed to fulfill a given task automatically from CN5, which is then automatically added to the knowledgebase of the agent.

### 4.6 Answer Set Programming for Collaborative Housekeeping Robotics

Erdem et al. present in [11] a framework utilizing ASP and CN5 for representing commonsense knowledge. The commonsense knowledge is used to plan and execute household tasks. Hereby, ASP is used to represent the task of tidying a house consisting of three rooms that include a kitchen, a bathroom, and a living room. Furthermore, the possible actions of a robot are modeled in ASP by using rules that are dependent on time steps. The commonsense knowledge used by Erdem et al. is extracted from ConceptNet 4. This includes two relations given by ConceptNet: The first relation used, is *AtLocation*, denoting the estimated location of an object and is used for finding the possible locations of objects in the given rooms. The second relation is *HasProperty*, which is used in the context of fragile objects. Both relations are then used to formulate queries to ConceptNet resulting in possible locations of the objects and the information about which objects have to be treated carefully.

Comparing the paper presented by Erdem et al. with the *KnowledgebaseCreator*, both works utilize ConceptNet to extract commonsense knowledge, which is then represented in ASP, forming a knowledgebase. This knowledgebase can be used by robotic agents to solve given tasks. For example, to place objects at their usual location. A difference between both elaborations is given by the amount of ConceptNet relations that are used. In comparison to
Erdem et al., the KnowledgebaseCreator is not restricted to two relations but instead uses all relations introduced by ConceptNet. Furthermore, the KnowledgebaseCreator introduces an algorithm to prevent inconsistencies in the automatically extracted commonsense knowledge, which is not needed for the manual expansion with commonsense knowledge provided in [11].

4.7 Differentiation from the Presented Related Work

To sum up this chapter, the presented related work can be separated into two categories. The first category relies on CN5 while solving the presented problems. This category includes SenticSpace [5], which adds affective knowledge to concepts in order to determine opinions in texts in which they are not clearly stated. Furthermore, the Glass Infrastructure [21] is situated in this category, since it is used to support its users that plan a trip by providing commonsense knowledge. The last paper belonging to this category is RESI [26] since it uses commonsense knowledge extracted from CN5 to solve problems appearing in the field of Requirements Engineering. The second category presented in this chapter are papers that utilize ASP to formulate tasks, which have to be expanded by commonsense knowledge in order to be solvable. The first elaboration that is part of this category is presented by Balduccini [2]. Hereby, the process of solving a riddle in ASP is presented, which has to be expanded by commonsense knowledge in order to be solvable. The second paper is the ASP-based application of commonsense knowledge onto a set of symbols extracted from a video stream, which is presented in [1]. Generally speaking, the KnowledgebaseCreator combines features given in both categories. It provides commonsense knowledge extracted from CN5 in order to support robotic agents that have to solve tasks, which rely on this knowledge. Furthermore, the KnowledgebaseCreator provides this knowledge in ASP which additionally enables the robotic agent to reason about the extracted commonsense knowledge. This can be compared to the elaboration presented by Erdem et al. in [11]. In this work, ASP is combined with commonsense knowledge extracted from CN5, which is restricted to two given relations. In contrast to this, the KnowledgebaseCreator is able to use the complete set of relations given by CN5 and therefore provides broader commonsense knowledge.
5 Implementation of the **KnowledgebaseCreator**

In this chapter, the implementation of the graphical user interface and the installation of a local instance of ConceptNet 5 are shown. The created graphical user interface (**KnowledgebaseCreator**) is presented in Section 5.1. One part of this graphical user interface is the **Command History**, which shows the performed steps of the program. Furthermore, the **Command History** provides the functionality to save a program, load a program, and to undo performed steps. The implementation of the **Command History** is presented in 5.2. Furthermore, a wrapper for the Clingo solver is presented in [34], which has to be expanded in order to use it in the **KnowledgebaseCreator**. The expansions made to this wrapper are shown in Section 5.3. Additionally, the query mechanism presented in [34] is used to interact with the created knowledgebase, but by using it the Module Property can be violated. Therefore, an expansion to automatically satisfy the Module Property has been developed and is introduced in Section 5.4. As a last point, the ways of installing ConceptNet 5 (CN5) are presented in Section 5.5.

5.1 Graphical User Interface

In this section, the implementation of the of a graphical user interface, the **KnowledgebaseCreator** is presented, which enables the user to execute all features provided by Clingo. This includes adding a **Program Section**, grounding, solving, and changing the truth value of an **External Statement**. Furthermore, the **KnowledgebaseCreator** provides access to the queries presented in [34] and to interact with CN5. Figure 5.1 is a depiction of the **KnowledgebaseCreator**.

As you can see in this figure, the main window of the **KnowledgebaseCreator** is divided into three parts. The upper part of the GUI is a text field (**programField**) that provides the possibility to enter ASP rules and to formulate queries to both, the ASP solver and CN5. The second part of this GUI is a row of buttons, which are used to interact with Clingo and CN5. The first button (**Add**) of this row is solely used to add the ASP program shown in the **programField** to Clingo. This includes a check if every rule is terminated by a dot. Is this not the case, the user is informed by returning the corresponding line. Besides the **Add**
5 Implementation of the KnowledgebaseCreator

Figure 5.1: Main window of the KnowledgebaseCreator.

button, there are buttons for separated grounding and solving of ASP programs. The Ground button provides two functionalities: The first one is to add an ASP program to Clingo and ground this program in a second step. The second functionality is to ground an already added Program Section, which is done by providing the corresponding Program Section in the programField. After an ASP program has been added and grounded, the program can be solved using the Solve button. Furthermore, the user has the option to formulate the queries presented in Section 3.4 by using the Query button. Hereby, the two kinds of queries, ASPFactsQuery and ASPVariableQuery, are distinguished by the way the query is structured in the programField. An ASPFactsQuery is chosen if facts, meaning rules without a body, are given in the programField. An ASPVariableQuery is chosen if the programField contains at least one rule, which has both a head and a body. As the last point of this GUI part, the Get From ConceptNet button is used to pass queries to CN5. The third part of this GUI is used to present the results from queries (Query Result) and the Stable Models of the ASP program (Current Models and Sorted Models).

The results of a query are shown in Figure 5.2. Hereby, the programField contains an ASP program, which is then passed to Clingo in form of an ASPVariableQuery. This query can be considered as a typical part of solving household tasks since tasks in this kind of scenario often require to change the current location of the robot. Furthermore, it is part of a symbolic
5.1 Graphical User Interface

path planning approach presented in [34], in which a robotic agent is supposed to check if its navigation goal is reachable. The first line of this query is the queryRule. Hereby, the queryRule head is the answer to the query and is returned after the query has been solved. In this case, the queryRule head is derivable if the start room and the goal room are reachable. Alongside the queryRule, this query contains a rule, stating that the rooms r1410 and r1411 are always reachable. As a last point, this query consists of two facts marking the start and the end point of the navigation. After the query has been passed to Clingo, the results are returned to the KnowledgebaseCreator and are presented in the Query Results tab as shown in Figure 5.2. Since the query used in this example is an ASPVariableQuery, the type and the queryRule are shown in the first line. Additionally, the result of this query is shown. This can be either an empty set of predicates, in case the query does not have a solution, or the predicates matching the queryRule head. In this example, the query is solvable and the predicate goalReachable(1405B) and the Stable Models of the current ASP program are returned.

![Figure 5.2: KnowledgebaseCreator after executing a query.](image)

Furthermore, the current ASP program is shown in Current Program tab. Additionally, the KnowledgebaseCreator provides a way to interact with the External Statements of the current ASP program. The External Statements are automatically extracted from the ASP program and are presented in the External Statements Tab, which is shown in Figure 5.3.

This tab lists all External Statements present in the current ASP program and their corresponding truth value. For example, the External Statement -atLocation(cup1, table1) has the truth value False. To change the truth value of an External Statement, its name has to be entered in the text field below and the corresponding truth value has to
be selected. By pressing the Apply button, the truth value is applied. These include True, which states that the External Statement can be derived, False, which states that the External Statement cannot be derived, and Release, which removes the External Statement from the current ASP program. Furthermore, this part of the KnowledgebaseCreator contains a tab containing the Command History, which is going to be explained in detail in Section 5.2.

Besides the presented ways to interact with Clingo and CN5, the KnowledgebaseCreator provides further ways to influence the resulting Stable Models, the ASP program, and the returned concepts from CN5. This can be done by using the menus presented in Figure 5.4.

Figure 5.4(a) is a depiction of the menu, which is used to create a new program, to save a program or the current Stable Models, and to load a program or background knowledge. Hereby, the option New Program creates a new instance of Clingo and clears the previous program if there was one. The option Save Program stores the current progress in JSON [8], which can be loaded by using the option Load Program. Furthermore, the resulting Stable Models can be saved and background knowledge ASP programs can be loaded. Alongside this menu, the menu presented in Figure 5.4(b) is used to influence the Stable Models and
the concepts extracted from CN5. Thereby, four settings are provided. The first setting is the number of returned Stable Models. By default, this option is set to -1 stating that the Clingo default is used, which is the first Stable Model found. By changing this value to \( n \), Clingo returns at max \( n \) Stable Models in case there is more than one Stable Model. Hereby, the number zero is a special case, which causes Clingo to return all Stable Models found. Furthermore, the minimum weight of a CN5 edge can be set. The default value for this option is set to 1.0, which states that edges with a lower weight are ignored. By changing the third option, the way query results are shown is changed. By setting this option to “Yes”, queries are shown with the corresponding Stable Model to get an easy overview of the result. As the final point, the option \( \text{Save Models sorted?} \) determines if the Stable Models are saved in the way Clingo returned them or are sorted alphabetically before saving them.

5.2 Command History and Command Structure

The \textit{Command History} is another important feature of the KnowledgebaseCreator. It is used to show the steps the user has performed in the current interaction with Clingo and CN5. Figure 5.5 is an example of the \textit{Command History}.

![Command History](image)

\textbf{Figure 5.5: Command History containing entries for the performed commands.}

As you can see in this figure, the interaction with the KnowledgebaseCreator consisted of several queries to CNS, e. g., a query for the concept of a \textit{Clock}. Furthermore, two Program Sections have been grounded and the current ASP program has been solved twice. Besides the presentation of the performed steps, the \textit{Command History} is used to save the progress of the interaction with the KnowledgebaseCreator. In order to provide this functionality, the Command Pattern \textsuperscript{[50]} is used. In this pattern, method calls are encapsulated in command objects derived from a common interface class. This interface class provides fields and methods that are used to execute or to undo the command. The implementation of this design pattern is shown in Figure 5.6. Hereby, the interface is marked red and the derived
classes are marked blue.

![Class diagram of the command classes.](image)

The command interface is shown in the center of this figure. This interface provides one field, which contains the type of command, and three methods. These methods are `execute()`, `undo()` and `toJSON()`. While the usage of the first two methods can be derived from their names, the `toJSON()` method is used to save the information needed to execute the command in the JSON format, which is then later used to save the interaction with the `KnowledgebaseCreator`. Furthermore, undoing commands is not possible in general, since the commands interact with Clingo, which does not provide the functionality to undo its operations. Therefore, most undo methods remove the command from the `Command History`, so that the user can save the progress after undoing a command and reload the program afterward.

The first command derived from this interface is the `NewSolverCommand`, which is used to create a new instance of Clingo. Additionally, by executing this command the current program progress is deleted. The next command is the `AddCommand`, which extracts the `Program Section` from the ASP program given in the `programField` and adds the rules using the extracted `Program Section` to Clingo. Furthermore, the `GroundCommand` and the `SolveCommand` are used to provide the corresponding functionality of Clingo. In the case of the `GroundCommand`, two ways of grounding are distinguished. If the `programField` contains an ASP program, it is grounded in the given `Program Section`. If instead the `programField` solely contains a `Program Section` containing variables, the already added ASP program containing this `Program Section` is grounded with the given variables. Listing 5.1 is an example of this application of the `GroundCommand`. Hereby, the Lines 2 and 3 are an ASP program, that is added to the solver. By entering Line 6 in the `programField` and pressing the `Ground` button, the appearance of `n` is replaced by `tweety` and the program is grounded.
5.2 Command History and Command Structure

Alongside the presented commands, the AssignExternalCommand is used to change the truth value of an External Statement via the External Statements Tab. As a first point, this command checks if the entered string is valid and if the entered string is an External Statement. If both requirements are fulfilled, the truth value of the External Statement is changed to the given value. Furthermore, this command can be undone, if the External Statement was not released. This is done by saving the previous truth value and assigning this value in case the undo() method is called. The next command is named ConceptNetCommand and is used to interact with CN5. Since this command involves the handling of inconsistencies, a detailed explanation is given in Section 6.3. Moreover, two commands are provided encapsulating the functionality of the queries presented in Section 3.4. These commands are the FactsQueryCommand and the VariableQueryCommand. Additionally, two commands are provided to load background knowledge and saved programs. The first command is the LoadBackgroundCommand, which is executed via the menu presented in Figure 5.4(a). This command loads files with the file extension .lp containing ASP programs. After loading the ASP program, it is inserted into the programField so that the user can edit the program if necessary. The second command is the LoadProgramCommand, which is used to load the progress of a previous session. Thereby, the necessary fields are extracted from a file with the .cnlp extension. This file contains a saved Command History and the corresponding commands are created and executed. Additionally, two features have been implemented without using the interface presented in Figure 5.6 since their appearance in the Command History is not needed. These features are used to the current Stable Models in a .txt file and to save the Command History in a .cnlp file. In order to save the current progress of the session, the toJSON() method of each command in the Command History is called, which extracts the necessary fields from the commands and saves them with their corresponding types. Hereby, the necessary fields are shown in the derived command classes in Figure 5.6. For example, the resulting JSON output of a GroundCommand is shown in Listing 5.2.

```json
{
  "program": "#program bird(n).
  \n  bird(n).
  \n  #external a(n).
  -flies(X).
  \n  \n  flies(X) :- bird(X), not -flies(X).
  \n  #program bird(tweety).
  \n  #program added to the solver
  #program bird(n).
  bird(n).
  \n  #Program Section, which is grounded
  #program bird(tweety).
}
```

Listing 5.2: Excerpt of a saved Command History in JSON.

In this example, the program that has to be grounded and the type of the command are
saved. If this example is loaded via a \textit{LoadProgramCommand}, the “type” is extracted and a corresponding command is created. In this case, a \textit{GroundCommand} is created and the string with the tag “program” is grounded by Clingo.

5.3 ASP Solver Wrapper

The \textit{KnowledgebaseCreator} provides access to the query mechanism presented in [34], which enables the user to filter the Stable Models of the ASP program and to temporarily alter the ASP program. The use of this query mechanism, which is provided by the \textit{ASPSolverWrapper} as presented in [34] has one major drawback: It is integrated into the engine of ALICA (A Language for Interactive Cooperative Agents) [43], which is a language to model the behavior of teams of autonomous agents. Since the \textit{KnowledgebaseCreator} should not depend on ALICA, the query mechanism has been adapted. The adaptation of the presented query mechanism is shown in Figure 5.7.

![Figure 5.7: Expansion of the query mechanism presented in [34].](image)

As you can see in this figure, the query mechanism has been split into three parts. The central part of the adapted \textit{ASPSolverWrapper} is the ASPCommons package, which is marked in red. The ASPCommmons package contains the interfaces that have to be implemented in order to use an ASP solver in the \textit{KnowledgebaseCreator} and ALICA. Furthermore, these interfaces are used by the ALICA\_ASPSolver, marked in blue, that enables ALICA the use of ASP and the query mechanism. Besides these packages, the package ASPSolver, marked in green, has been created. This package contains three classes. The first class is the ASPSolver, which is a wrapper for Clingo and provides its functionality. This includes adding \textit{Program Sections} to the solver, grounding, solving, and the change of \textit{External Statement’s} truth.
5.4 Automatic Satisfaction of the Module Property for Queries

The query mechanism presented in [34] enables the user to efficiently filter the Stable Models and to temporarily alter an ASP program. Searching the Stable Models can be done without risking a violation of the Module Property (Section 3.3), but altering the ASP program can cause a violation of the Module Property. By violating the Module Property, the ASP program is no longer able to be grounded and cannot be solved anymore. Furthermore, a violation of the Module Property causes a crash of Clingo and therefore should be avoided. So far, the ASPVariableQuery presented in [34] did not provide any support for satisfying the Module Property and the user had to avoid a violation of the Module Property manually. As already discussed in Section 3.3 the satisfaction of the Module Property can be achieved by creating unique rule heads for every rule in the query.

In order to manually satisfy the Module Property, the user has two possibilities. The first way is to maintain a counter and to add the value of the counter to every rule head. This has two drawbacks. The first drawback is that the user has to maintain the counter, which can cause problems if it has to be maintained during several sessions. Another drawback is that the arity of the queryRule head is changed, which has to be considered when searching for the predicate. The second way to satisfy the Module Property is to enclose the rules’ heads inside a unique predicate, which can be realized using a fixed string and a counter. This way has the drawback that the counter has to be maintained and can cause problems while being used in several sessions. Moreover, this way does not change the arity of the rule heads but replaces it with a new predicate, which has to be considered when looking for the predicate in the Stable Models. A drawback both ways of satisfying the Module Property
have in common is that the user has to maintain the information, which unique rule head has been used so far. This can cause problems in long interactions with an ASP program or an ASP program that has been created in several sessions.

To cope with this problem and to make the Module Property transparent for the user, the KnowledgebaseCreator provides an automatic satisfaction of the Module Property for ASPVariableQueries. Therefore, the second way of satisfying the Module Property mentioned in the previous paragraph is chosen. Hereby, the rule heads are wrapped inside a unique predicate, which is exchanged for every query. The unique predicate is created by combining a prefix with a counter, which is maintained by the ASPSolver class shown in Section 5.3. The algorithm that has been developed to satisfy the Module Property is shown in Algorithm 6.

Algorithm 6: Automatic satisfaction of the Module Property

Input : ASPVariableQuery q, Counter c
Output : Unique expanded ASPVariableQuery

1. Create unique Program Section ps
2. Create unique External Statement ex
3. foreach fact in q.facts do
   4. replace occurrence of fact in q.queryRule.body by ps(fact)
5. end
6. add(q.queryRule.body, ex)
7. q' = duplicate q.queryRule
8. foreach rule in q.rules do
   9. replace occurrence of rule.head in q' by ps(rule.head)
10. end
11. foreach rule in q.rules do
12.   add(rule.body, ex)
13. end
14. foreach fact in q.facts do
15.   replace fact in q.queryRule by ps(fact)
16.   expand(fact, ex)
17. end
18. return q

This algorithm uses an ASPVariableQuery q and the value of the counter c, which is maintained by the ASPSolver, as input and returns a modified unique ASPVariableQuery. Hereby, the first two steps in this algorithm create a unique Program Section ps and a unique External Statement ex, which are used in the following steps. In Step 3 (Lines 3 - 5), the query facts appearing in the queryRule body are encapsulated in a new predicate consisting of ps, making them unique. After this process is completed, the queryRule is expanded by the unique External Statement ex, which allows the removal of the query after it has been answered. Step 4 of this algorithm is the duplication of the queryRule q’, which will be explained based on an example in the following paragraphs. Furthermore, in Step 5 (Lines
the occurrences of all rule heads are encapsulated in a new predicate consisting of $ps$ as well, which marks the end of altering the $queryRule$. Additionally, the rules and facts have to be adapted, since they can still violate the Module Property. Therefore, Step 6 (Lines 11 - 14) alters the rule head predicates by encapsulating them in the predicate consisting of $ps$ and adding $ex$ to the body of every rule. This creates unique rules, which can later be removed from the ASP program. The same process is used in Step 7 (Lines 15 - 18) for altering the query's facts, which results in a unique query that is finally returned.

An example of the way this algorithm works is presented in Listing 5.3. This example is part of the evaluation scenario presented in [34], which is a topological path planning approach that is situated in the Distributed Systems Department of the University of Kassel. Summarizing this scenario it can be said that it models the department using ASP predicates and verifies the reachability of a navigation goal using $ASPVariableQueries$. Therefore, it is suited to show the advantages of an automated satisfaction of the Module Property. In Listing 5.3 a path planning query is formulated, which is used to check if a given goal position is reachable. This example consists of the $queryRule$ in Line 1, a rule stating that if room 1405 and room 1406 are given as predicates they are reachable from each other, and two facts defining the start and goal position of the navigation. These two facts can cause a violation of the Module Property since they can form strongly connected components [3.3] if this query is used again. Therefore, the automated satisfaction of the Module Property is used.

```
1 goalReachable(X):- reachable(X,Y), goal(X), start(Y), room(X), room(Y).
2 reachable(r1405,r1406):- room(r1405), room(r1406).
3 goal(r1405B).
4 start(r1411).
```

Listing 5.3: $ASPVariableQuery$ before applying the automated satisfaction of the Module Property.

The result of this process is shown in Listing 5.4. In this example, it is assumed that this is the first query and therefore, the counter $c$ managed by the ASPSolver has the value 1.

```
1 #program query1.
2 #external extQuery1.
3 query1(goalReachable(X)):- reachable(X,Y), query1(goal(X)), query1(start(Y)), room(X), room(Y), extQuery1.
4 query1(goalReachable(X)):- query1(reachable(X,Y)), query1(goal(X)), query1(start(Y)), room(X), room(Y), extQuery1.
5 query1(reachable(r1405,r1406)):- room(r1405), room(r1406), extQuery1.
6 query1(goal(r1405B)):- extQuery1.
7 query1(start(r1411)):- extQuery1.
```

Listing 5.4: $ASPVariableQuery$ after applying the automated satisfaction of the Module Property.
Lines 1 and 2 have already been added in the original version of the *ASPVariableQuery*, which is a unique *Program Section* and *External Statement*. This part of the query is followed by the *queryRule* and its duplicate in Line 3 and 4. In both cases the heads are expanded by a new predicate *query1* that is identical to the *Program Section*, rendering the heads unique. Furthermore, the appearance of every fact is replaced by its expanded version, which is shown in the Lines 6 and 7 of this example. The only difference between the duplicates is the handling of the rule heads of additional rules like Line 5. In order to satisfy the Module Property, the head of the rule has to be expanded as well, which causes a change of the arity and the name of the predicate. This change has to be conducted in the *queryRule*, too, in order to use the corresponding predicate. This can cause problems when solving the query. In this example, the reachable predicate is expanded by the rule \( \text{reachable}(r1405, r1406) : \text{room}(r1405), \text{room}(r1406) \). Therefore, the head has to be expanded and its occurrence in the *queryRule* has to be replaced. This causes that the reachable predicates in the knowledgebase are no longer appearing in the *queryRule*. Thus, leaving out possible solutions for the query. To cope with this problem, the duplicated *queryRule* is used. The appearance of the additional rule heads is not replaced, leaving the possibility to use the knowledgebase. Another way to cope with this problem would be to use every combination of expanded and non-expanded predicates. This would allow using further combinations of rules and background knowledge. A major drawback of this solution is the combinatorial explosion if several rules are added. Therefore, the presented approach has been chosen since the number of created *queryRules* is independent of the number of rules. Before the results of the query are returned to the user, the unique predicate is removed to recreate the original query predicate.

In summary, it can be stated that the presented approach of an automated satisfaction of the Module Property allows the user to create *ASPVariableQueries*, without the opportunity to violate it. Furthermore, this approach allows expanding predicates appearing in the knowledgebase without losing the possibility to use the knowledgebase. As a last point, this approach is transparent to the user, who formulates the query and is given the result without any additional predicates encapsulating the original query.

### 5.5 Running a Local Copy of ConceptNet 5

ConceptNet 5 (CN5) provides two ways of accessing the database. The first way to access CN5 is the web API provided at the URL “api.conceptnet.io”. This API can be simply accessed by a browser but has two drawbacks: The first drawback is the limitation of queries to the API. The web API allows only 600 queries per minute and at maximum 6000 queries per hour. For example, the concept of a plant is connected to 2393 other concepts in CN5. To check this concept would last roughly four minutes, assumed that a query lasts 100 ms and that
20 concepts are returned per query. The second drawback is the required connection to the internet. This requirement can prevent the usage of CN5 on robots that are not connected to the internet. Due to these drawbacks, the use of a local copy is recommended.

CN5 provides two ways of installing a local copy of its database. The first way of installing CN5 is a manual compilation of the database. This installation requires 240 GB disk space and at least 14 GB available RAM. Once the installation process is finished, this manual installation provides a PostgreSQL database and several CSV files containing the edges and concepts. A major drawback of this installation is the system requirements that are hard to fulfill for smaller systems. Furthermore, the manual installation does not provide the REST API of CN5 that enables the handling of complex queries in form of an URL. Alongside the way of installing a local copy manually, CN5 provides a Docker container consisting of a prepared database with an installed REST API. Generally speaking, Docker is a lightweight virtual machine that does not need a guest operating system. Instead, Docker uses the operating system of the host and provides an interface for Docker containers, which are images containing everything that is necessary to run the application. For example, these containers include the code, libraries, settings, and the needed runtime environment. For a detailed description, the interested reader is referred to [24]. In comparison to the manual installation, the installation using a Docker container only requires 120 GB disk space and 4 GB RAM. Hereby, the lesser disk space requirement is caused by leaving out the CSV files mentioned in the previous installation, since they are not needed for the interaction with the database. The only drawback of this way of installing CN5 is the need for additional packages, which have to be installed in order to run and install CN5.

To sum up, the web API has the lowest system requirements but can only be used while a connection to the internet is established and is restricted in the number of queries per minute. In comparison to this, both ways of running a local copy do not need an active connection to the internet but require high amounts of disk space and RAM. Hereby, the installation via Docker requires fewer system resources and provides an easy access to CN5’s local REST API. Because of these advantages, the installation with Docker is chosen in this elaboration. Therefore, an installation guide for properly setting up CN5 is provided in the next section.

**Installing ConceptNet 5 via Docker**

In order to properly use a local copy of CN5 and its web API, several steps need to be performed to set up Docker, install Docker-Compose, and finally install CN5. Hereby, Docker is used for wrapping the requirements needed to run a specific piece of software in a container. This includes the source code, libraries, a runtime environment and necessary tools. Docker-Compose is an expansion for Docker, which enables it to handle multi-container Docker applications like CN5 (database, input data, web server cache, and REST API).
Since installing these tools requires many steps, a guide to properly install CN5 is presented in this section, which is inspired by the following tutorials \[22, 23, 44\]. The first step is to install Docker. In order to install this package on a new system, the Docker repository has to be set up properly. Furthermore, Ubuntu 16.04 with a kernel version of 3.10 or higher and at least 750 MB of disk space are required to run Docker. Apart from these requirements, further system requirements are introduced by each container, which is used. Hereby, the required disk space and the necessary RAM depends on the container itself. The command lines to do this are presented in Listing 5.5.

1. $ sudo apt-get install -y --no-install-recommends apt-transport-https ca-certificates curl software-properties-common
2. $ curl -fsSL https://apt.dockerproject.org/gpg | sudo apt-key add -
3. $ sudo add-apt-repository "deb https://apt.dockerproject.org/repo/ubuntu-$(lsb_release -cs) main"
4. $ sudo apt-get update
5. $ sudo apt-get -y install docker-engine

Listing 5.5: Command lines used to install Docker.

The first and second lines in 5.5 are used to install the dependencies, that are needed by Docker and to get a key required for installing Docker. This is followed by adding the corresponding repository to apt, updating apt and finally installing Docker with Line 5. Once these steps are finished, Docker is properly installed and Docker-Compose can be installed. Docker-Compose requires at least Ubuntu 16.04 with a kernel version of 3.10 or higher and at least 40 KB of available disk space. The installation of Docker-Compose is done by the command lines shown in Listing 5.6.

1. $ sudo apt-get install python-pip
2. $ pip install docker-compose

Listing 5.6: Command lines used to install Docker-Compose.

Once Docker-Compose is installed, the CN5 repository can be downloaded and the query limitation of 600 queries per minute and 6000 queries per hour has to be removed. This is done by the commands in Listing 5.7.

1. $ sudo apt-get install git
2. $ git clone git@github.com:commonsense/conceptnet5 -b version5.5
3. $ vim conceptnet5/web/conceptnet_web/api.py

Listing 5.7: Download CN5 and remove query limitations.

After installing git \[29\] and downloading the CN5 repository with Commands 1 and 2, the configuration file for the interaction with the API is opened with Command 3. This file contains the line

\[
\text{limiter} = \text{Limiter}(\text{app}, \text{global_limits}=\text{["600}}
\]
5.5 Running a Local Copy of ConceptNet 5

which limits the access frequency by the entries inside the square brackets. In order to remove this limitation this line has to be replaced by the following line

```python
limiter = Limiter(app, global_limits=[])  
```

By replacing this line all limitations are released and the API can be queried in any frequency. After these steps, the system is ready to install CN5, if there are at least 120 GB disk space available. The installation process is done by the command lines in Listing 5.8.

```bash
1  $ cd conceptnet5  
2  $ docker-compose up --build
```

Listing 5.8: Install CN5.

Line 1 is used to switch to the folder `conceptnet5` containing the data and scripts to build the CN5 database, set up the Docker image, and configure the REST API. Line 2 is then used to install CN5, which consists of three Docker containers. The first one is `conceptnet5_psql`, which stores the database. The second one is named `conceptnet_cn5data`, containing the loaded data and the third one, `conceptnet5_nginx`, containing the server cache. This process will last several hours and will occupy roughly 120 GB of disk space. Once the installation is done, the final step of the installation is configuring the `/etc/hosts` file of the system by adding the line `127.0.0.1 api.localhost`, which enables to use the CN5 API via a web browser. This can be done in two ways: `http://localhost/`, which is used to show the local web interface of CN5 and `http://api.localhost`, which is used to access the API returning query results in JSON format. Furthermore, this step allows the interaction of the `KnowledgebaseCreator` with the REST API of CN5 since it maps the start of each query URI to the address of the local instance of CN5.
6 Consistent Knowledge from ConceptNet 5

In this section, the extraction of consistent knowledge from ConceptNet 5 (CN5) is discussed. Therefore, the influence of the weight of a CN5 edge is discussed in Section 6.1. Furthermore, possible sources of inconsistencies are presented in Section 6.2. Finally, the implementation of an algorithm to remove these inconsistencies and to create ASP rules from the extracted concepts is shown in Section 6.3.

6.1 Influence of the ConceptNet 5 Edge Weight

The weight of a ConceptNet 5 edge has an important role in the evaluation of the reliability of a CN5 edge and the information stored in it since it allows to draw conclusions about the sources of the edge. In CN5 every edge has a positive weight. In contrast to this, in previous versions of CN5 edges were able to possess negative weights. These negative weights were not supposed to express that the opposite of this edge is true, e. g., a negative Synonym edge is not an Antonym. Instead, they were used to express that CN5 believes that this edge is not true. This means an edge with a negative weight could have been considered as an inconsistency in the way that CN5 contained edges representing false information. Hence, negative edges were removed after tests, which are presented in [48]. Thus, only positive edges remain part of the current version of CN5. Still, the positive weight of an edge is helpful, since it states how reliable the edge is. Hereby, two factors have to be considered: The first one is the weight itself. The most common value for the weight of an edge is 1.0. In most cases, this value is caused by a verified source, for example, WordNet. Since sources like this are actively maintained, they provide a high weight value. Edges with a weight below 1.0 lack a verified source and originate from websites collecting knowledge or word games [51]. This results in not verified edges, which can be considered as weakly supported knowledge and therefore should not be used. An example of this can be found in Figure 3.2. In this figure, two edges exist between the two concepts Coffee and Drinking. These edges are UsedFor with a weight of 2.8 and RelatedTo with a weight of 0.1. In this case, an agent could randomly choose the RelatedTo edge, which is the most general relation in CN5 and therefore lose information given by the UsedFor edge. Besides the weight itself, the number of supporters can indicate how reliable an edge is. Hereby, a high number of supporters implies a more reliable edge, but a low number does not indicate, that an edge is
unreliable, since the supporter could be a verified source. In summary, it can be stated that
the weight defines how reliable the edge is and that only reliable edges should be included in
the knowledgebase as shown in the example above. Hence, in this elaboration, a minimum
weight of 1.0 is used as a default value for edges to be accepted since a weight above 1.0
indicates that the edge is supported by at least one verified source.

6.2 Possible Inconsistencies in ConceptNet 5

Since CN5 is a huge commonsense knowledge database, which has been created by combining
several knowledge sources, inconsistencies in its knowledge can hardly be avoided. In this
section, several types of inconsistencies that may arise are presented and discussed.

A source of inconsistencies for robotic agents can be found in the properties of concepts.
The result of a query to CN5 is a set of edges connecting the queried concept with other
concepts. Among these returned concepts some are labeled as adjectives. These adjectives
can be seen as the properties of the queried concept. Since CN5 is a source of commonsense
knowledge, these adjectives can include contradicting concepts, which is intended by CN5 to
provide as broad as possible knowledge about a concept. The appearance of contradicting
properties can cause semantic inconsistencies. Hereby, an agent could conclude that a
concept has contradicting properties since both properties have been extracted from CN5.
An example of this kind of inconsistencies can be found in the properties of the concept of a
Plant. The concept of a Plant is connected to 2393 concepts by edges with a weight
of 1.0 or higher. Among these concepts, 165 are marked as adjectives and can be seen as
properties of a Plant. Furthermore, six pairs of these properties contradict each other and
are shown in Figure [6.1]. Thereby, contradicting properties are marked in the same color and
their relation to the concept of a Plant is shown.

One pair of contradicting properties are the concepts Evergreen and Deciduous. The
contradiction in these two concepts can easily be seen. A Plant with the property
Evergreen has leaves throughout the year, meaning they are always green. In contrast
to this, a Plant with the property Deciduous loses its leaves seasonally. Therefore, the
custom of a Plant cannot have both properties at the same time. By adding both concepts
to the created knowledgebase, an agent could derive, that an instance of a Plant can have
both properties, which will create inconsistent knowledge about an object. Therefore, the
deduction of such knowledge has to be prevented. As you can see in this figure, every pair
of contradicting properties is connected via the Antonym relation with at least a weight of
1.0, which means that Antonym edges can be used to find contradicting properties, which
can cause semantic inconsistencies if given to a robotic agent. Thus, removing one of the
contradicting concepts will prevent this kind of inconsistencies in the knowledgebase. In
order to decide which of the contradicting concepts is to be removed, the weight of the edge
between the queried concept and the contradicting concepts could be used. Hereby, the concept, which is connected to the higher weighted edge should be kept, since it is supported by more knowledge sources or more reliable knowledge sources. If both concepts have the same weight, the concept that is returned first by CN5 is kept, since there is no other automatic way to determine, which concept should be kept in the knowledgebase.

The ambiguity of concepts can be considered as an inconsistency since a robot could derive false information for the different senses of a concept. For example, let us consider the concept of a Table. A Table can be either a piece of furniture or an arrangement of data in rows and columns. Both ways of understanding the concept of a Table result in different associations to other concepts, e.g., that a Table is made of wood or that it can be used in mathematical calculations. Alongside the concept of a Table, there are several other examples of this kind of inconsistencies. One of these ambiguous concepts is a Fork. A Fork can be a tool that a person uses to eat, a part of a tree, and a place where two roads meet. Furthermore, the concept of a Pipe can introduce inconsistencies, too. A Pipe can be used to smoke tobacco or to transport water or gas. While igniting a Pipe filled with tobacco is rather save, a robot should not try to do the same with a Pipe containing gas. Furthermore, the concept of Glass can cause inconsistencies. A Glass can be used to serve beverages, is part of a window, and the plural of this concept help persons to see the world clearly. As you can see, without further context, the sense of an ambiguous concept
Consistent Knowledge from ConceptNet 5 is hardly understandable. In order to cope with this problem, CN5 would have to provide further context to ambiguous concepts, which is not intended so far. As stated by Robert Speer in [45], a previous version of CN5 used sense information from Wiktionary\(^1\) and WordNet\(^2\) This caused a decrease in accuracy during their evaluation of word relatedness in texts and therefore was removed. Thus, CN5 does not provide information to disambiguate ambiguous concepts, meaning that CN5 does not provide any information on its edges that show to which sense of a concept they belong. Furthermore, there is no standardized sense of a concept since the sense of these ambiguous concepts depends on the context they appear in. Since there is no way to disambiguate these concepts, they stay part of the knowledgebase because removing them would drastically decrease the extracted knowledge and would reduce the applicability of the KnowledgebaseCreator.

### 6.3 Removal of Inconsistencies and Representation in ASP

In order to interact with CN5, the KnowledgebaseCreator provides a special query mechanism handling the interaction with CN5. Therefore, the command ConceptNetCommand has been implemented following the Command Pattern presented in Section 5.2. The ConceptNetCommand hereby provides six different kinds of queries enabling the use of the CN5 REST API. In order to differentiate these queries from the ASP queries presented in Section 3.4, they are marked with the prefix “cn5_”. The queries provided by the ConceptNetCommand are presented in Table 6.1.

<table>
<thead>
<tr>
<th>Query</th>
<th>Returned edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>cn5_Concept</td>
<td>Any edges connected to the given concept</td>
</tr>
<tr>
<td>cn5_Wildcard(cn5_Concept, cn5_Concept)</td>
<td>Any edge containing both concepts</td>
</tr>
<tr>
<td>cn5_Relation(cn5_Wildcard, cn5_Concept)</td>
<td>Every edge having the given relation and ending with the given concept</td>
</tr>
<tr>
<td>cn5_Relation(cn5_Concept, cn5_Wildcard)</td>
<td>Every edge having the given relation and starting with the given concept</td>
</tr>
<tr>
<td>cn5_Relation(cn5_Concept, cn5_Concept)</td>
<td>The defined edge, if it exists in CN5</td>
</tr>
<tr>
<td>cn5_Relation(cn5_Concept)</td>
<td>Any edge containing the given relation and concept</td>
</tr>
</tbody>
</table>

Table 6.1: Possible queries to CN5 via the ConceptNetCommand.

As shown in this table, the query mechanism provides several combinations of concepts and relations. Hereby, cn5_Relation can be replaced by any CN5 base relation and cn5_Wildcard is a special string stating that the value of this field is not given. An example of this kind of queries is cn5_Wildcard(cn5_cup, cn5_table), which results in one edge

\(^1\)Website: https://de.wiktionary.org
\(^2\)Website: https://wordnet.princeton.edu/
containing the `AtLocation` relation. The formulation of one of these queries marks the first step of the interaction with CN5 and the creation of a consistent ASP knowledgebase. Figure 6.2 is a depiction of the interaction with CN5 and the creation of the ASP-based knowledgebase.

![Workflow of a CN5 query.](#)

The workflow of a CN5 query consists of three major steps, which will be presented in the following paragraphs. These steps include the extraction of edges and concepts marked in blue, the removal of inconsistencies marked in green, and the creation of the ASP knowledgebase marked in red. The first step in this workflow is the interaction with CN5 in order to extract knowledge in form of edges and concepts. Hereby, a user enters a query as presented in Table 6.1. This query is checked for syntactic errors, which include
Consistent Knowledge from ConceptNet 5

missing prefixes or unsupported relations (Section 3.5). In the case of an error, the query is rejected and can be corrected by the user. If the query is correct, a URI is created by using the information given in the query, which is then passed via an HTTP GET request to the REST API of CN5. Figure 6.3 is an example for the resulting URI for a given CN5 query. Hereby, the arrows indicate the translation of the query into the URI.

![Figure 6.3: Translation of a CN5 query into a URI.](http://api.localhost/query?rel=/r/AtLocation&start=/c/en/cup&end=/c/en/table)

The HTTP GET request is given to the local instance of CN5 and contains a query, which consists of the relation `AtLocation` and both concepts (`Cup` and `Table`). This URI is then passed to CN5 and the corresponding concepts and edges are returned. In the case of this example, CN5 returns one edge that states that a `Cup` can be found at a `Table` with the weight of 4.0, which is generated by seven supporters. The result of one edge is typically for very specific queries like the query presented in the example above. In contrast to this, more general queries like for the concept of a `Plant`, produce more output than just one edge. By default, the returned edges are limited to the 20 edges with the highest weights, but the returned result also provides a URI for querying the next 20 edges. This number is the default value provided by CN5 but is increased to the maximum value of 1000 in this elaboration in order to maintain a fast communication with CN5. The reasons for this choice are shown in Chapter 7.2. These edges are saved in a data structure wrapping the returned edges in C++ objects. Furthermore, the process of extracting edges is stopped as soon as an edge with a lower weight than the given threshold (Section 6.1) is found, which can be done since the returned edges are ordered by weight. Once all edges of an HTTP GET request are extracted, the contained URI to the next edges is passed to CN5. If there is no URI pointing to further edges, or the weight has fallen below the threshold, the extraction of knowledge from CN5 is finished and as a result, a C++ object is created encapsulating every edge related to the query, with a weight higher than the given threshold. This marks the end of the first step of the interaction with CN5.

The second step of the interaction with CN5 is the removal of contradicting properties among the gathered edges. Like mentioned in the paragraph above, only edges with a weight higher than the threshold have been extracted from CN5. Among the returned edges some are marked as adjectives. These adjectives can be seen as properties of the queried concept. Since CN5 provides as broad as possible commonsense knowledge, these adjectives can contradict each other. Hence, this kind of inconsistencies is removed during this step.
Therefore, all concepts marked as adjectives are determined and saved in the set *Adjectives*. In order to find inconsistencies among them, the REST API of CN5 is used. Hereby, the *Antonyms* of all concepts saved in *Adjectives* are determined and stored in a map, the *AntonymMap*, mapping each adjective to its *Antonyms*. After determining the *Antonyms*, the first check for contradicting properties is conducted. Thereby, it is checked if any adjective in *Adjectives* is mapped as an *Antonym* to another adjective. If such kind of combination is found, the adjectives contradict each other and would introduce an inconsistency to the properties of the queried concept. Therefore, one of the edges representing these adjectives has to be removed from *Adjectives*. Thereby, the weight of the edges plays an important role. As already stated, the weight of an edge represents how reliable the information stored in the edge is, thus the concept connected with the lower weighted edge is removed from the set *Adjectives*, removing the inconsistency. In the case that both concepts are connected with equally weighted edges, the edge, which was returned later by CN5 is removed. This process is continued until all adjectives have been checked, resulting in the removal of inconsistent properties. Still, inconsistencies can be found inside the set *Adjectives*. These inconsistencies can be caused by synonyms of the adjectives in set *Adjectives*. Therefore, an additional set of queries is formulated and given to the REST API of CN5. Hereby, a variant of the broad search is used to find concepts linked to the adjectives via the relations *IsA*, *Synonym*, *DefinedAs*, and *PartOf*. This search creates queries from the remaining adjectives in *Adjectives* and the four relations given. The results are filtered by weight and stored in the set *NewAdjectives*, as long as it is not already part of it. This results in a set containing adjectives with a synonymic character to the ones in *Adjectives*. After every adjective in *Adjectives* has been queried, the adjectives in *NewAdjectives* are used in the search in order to find all possible synonyms to the ones in *Adjectives*. After the second step is finished, the ASP knowledgebase can be created, which is an ASP program consisting of two *Program Sections*, respectively, layers. These layers are the *Meta-Knowledge* and the *KnowledgeBase* itself. The *Meta-Knowledge* provides commonsense knowledge for the user, while the *KnowledgeBase* provides ASP rules for applying the
Meta-Knowledge to specific symbols. Therefore, the Meta-Knowledge layer is explained first. Generally speaking, the Meta-Knowledge provides a translation of the CN5 edges into ASP. Furthermore, it is denoted as Meta-Knowledge since it does not provide knowledge about the symbols in the current environment of the robot, but instead, provides knowledge about the concepts of CN5. This knowledge includes the relation between the concepts and a weight, which expresses how reliable this knowledge is. The idea behind this layers is that a robot should search for possible solutions for a given task in this knowledge. For example, the robot is supposed to fetch a Cup. Therefore, the robot looks for predicates in the Meta-Knowledge, which indicate possible location for a Cup, e.g. a Table. Furthermore, the Meta-Knowledge consists of ternary ASP predicates and provides the knowledge stored in the extracted edges of CN5. Listing 6.1 is an extract of the Meta-Knowledge created for the concept of a Cup.

```
#program cn5_metaKnowledge.
cn5_CapableOf(cn5_cup, cn5_hold_liquids, 90).
cn5_AtLocation(cn5_coffee, cn5_cup, 8).
cn5_AtLocation(cn5_cup, cn5_dishwasher, 8).
cn5_UsedFor(cn5_cup, cn5_drinking_out_of, 1).
```

Listing 6.1: Extract of the Meta-Knowledge for the concept of a Cup.

The predicates shown in this listing are an excerpt from 277 edges that are connected to the concept of a Cup. As you can see in this listing, all predicates that are part of the Meta-Knowledge have the same structure and are expanded with the prefix “cn5_”, in order to indicate that the predicates are part of the Meta-Knowledge. Hereby, a CN5 edge is translated in the following way. The relation of the edge is used as the name of the predicate. Furthermore, the start and end concepts of the edge are the first and second argument of the resulting predicate. The third argument is a combination of the weight and the number of supporters. Therefore, the weight is multiplied by the number of supporters, in order to give an idea how reliable the resulting predicate is. Additionally, the Meta-Knowledge is summarized in the Program Section cn5_metaKnowledge. Finally, this Program Section is grounded and solved, which adds this knowledge to the Stable Models of the current ASP program. Besides the Meta-Knowledge, the KnowledgebaseCreator provides a second layer of ASP rules, which is the KnowledgeBase. The KnowledgeBase provides ASP rules that are used to apply the commonsense knowledge of the Meta-Knowledge onto specific symbols, which represent objects in the environment of the robot. In contrast to the Meta-Knowledge, the KnowledgeBase is only added to the program of the solver and has to be grounded and solved separately. Furthermore, the ASP rules are summarized in the Program Section cn5Knowledge(n, m). This Program Section allows a robot to ground this Program Section with two concepts. By grounding this Program Section with two concepts, the KnowledgeBase applies the extracted knowledge onto specific symbols from the environment, resulting in predicates that express the commonsense knowledge between the specific symbols. Furthermore, the Program Sections of the KnowledgeBase contain an
External Statement, which is used to dynamically change the derived knowledge. An example for the created rules is given in Listing 6.2.

```plaintext
#program cn5Knowledge(n, m).
#external -atLocation(n, m).
atLocation(n, m) :- not -atLocation(n, m), cup(n), table(m).
atLocation(n, m) :- not -atLocation(n, m), cup(n), shelf(m).

#program cn5Knowledge(n, m).
#external -derivedFrom(n, m).
derivedFrom(n, m) :- not -derivedFrom(n, m), coffee_cup(n), cup(m).
derivedFrom(n, m) :- not -derivedFrom(n, m), teacup(n), cup(m).

#program cn5Knowledge(n, m).
#external -similarTo(n, m).
similarTo(n, m) :- not -similarTo(n, m), cup(n), mug(m).
similarTo(n, m) :- not -similarTo(n, m), cup(n), pannikin(m).
```

Listing 6.2: Extract of the KnowledgeBase for the concept of a Cup.

As shown in this excerpt of the KnowledgeBase, the KnowledgeBase consists of several parts, which are combined in the same Program Section. In this example, the KnowledgeBase includes parts for the CN5 relations AtLocation, DerivedFrom, and SimilarTo. Furthermore, every part of the KnowledgeBase is equipped with an External Statement representing a relation of CN5, e.g., -atLocation(n, m). Hereby, the External Statement is set to false by default and therefore, the head of the rule is derivable, as long as the External Statement is not set to true. Additionally, all rules that are part of the KnowledgeBase are constructed by the same pattern: The head consists of a single predicate, whose relation is derived from CN5 and contains two specific symbols as arguments. The body of each rule consists of the strongly negated External Statement and two predicates defining the predicates of the symbols, for example, cup(n). The pattern of building these rules allows an agent to ground the Program Section with two concrete symbols, e.g., cup(blue_cup) and table(kitchen_table). Using the example above, this would result in the predicate atLocation(blue_cup, kitchen_table) stating that the blue cup can be found on the kitchen table, which then can be checked by the agent. If this is not the case, the agent can change the truth value of the corresponding External Statement, removing the predicate from the Stable Models of the ASP program.

Combination of the Meta-Knowledge and the KnowledgeBase

The deployment of consistent knowledge from CN5 for autonomous agents presented in this elaboration is based on the combination of the Meta-Knowledge and the KnowledgeBase. In order to understand the collaboration of both parts, let us consider an autonomous agent that is able to perceive and interact with its environment and to abstract symbols from objects.
in its world, e.g., a Cup or a Table. A typical task in such kind of environment would be to fetch a Cup. This task can be very easy for a human being that is in possession of commonsense knowledge. This knowledge could include that a Cup can be found on a Table or on a Shelf. An autonomous agent like presented above is missing this kind of knowledge. This problem is tackled by the Meta-Knowledge. Hereby, the agent, which is supposed to fetch a Cup, can query CN5 for commonsense knowledge of a Cup. As a result, the commonsense knowledge is returned in form of the Meta-Knowledge, which consists of ternary predicates stating commonsense relations between objects and how reliable this knowledge is. In this example, CN5 would return predicates stating that a Cup can normally be found at a Table, on a Shelf, or a Dishwasher. By acquiring this knowledge, the agent is now able to use the KnowledgeBase to map the perceived symbols onto the extracted knowledge, which then enables the agent to fulfill its task. For example, the agent has perceived a blue Cup and the kitchen Table and stores this information in the symbols cup(blue) and table(kitchen). Since the agent is ordered to fetch a Cup it needs to know where a Cup can usually be found, it queries CN5 for this concept and both the Meta-Knowledge and the KnowledgeBase are returned. The agent can now look for typical locations of a Cup inside the Meta-Knowledge and learns that a Cup can normally be found on a Table. To solve the given task, the agent uses the KnowledgeBase to apply the given ASP rules to the symbols and receives predicates containing the relation between the symbols from its world. These predicates indicate the possible positions of the Cup and the agent is now able to plan and solve its given task. As you can see from this example, the combination of Meta-Knowledge and the KnowledgeBase enables an agent to extract knowledge about unknown symbols and to solve tasks, which it would not be capable of without any source of commonsense knowledge.
7 Evaluation

In this chapter, the evaluation of the KnowledgebaseCreator is presented, which is performed using the setup given in Section 7.1. A comparison of different query sizes is presented in Section 7.2 and the influence of the query cache is shown in Section 7.3. Furthermore, the runtime of the inconsistency removal algorithm is evaluated in Section 7.4. Additionally, Section 7.5 is an evaluation of the runtime of the created ASP knowledgebase. As a last point, a household scenario has been built that is used to evaluate the interaction of ASP and CN5. The results of this evaluation are shown in Section 7.6.

7.1 Evaluation Setup

The tests presented in the following sections have been performed on a Lenovo W550s Workstation equipped with an Intel Core i7-5500U Dual-Core mobile processor and 16 GB DDR3L-1600 MHz SO-DIMM RAM. Furthermore, Ubuntu 16.04 with kernel version 4.4.0-75-generic x86_64 was used as the operating system. The version of CN5 is 5.5.3, which is running in Docker version 17.04.0-ce in combination with Docker-Compose version 1.11.1. Additionally, the ASP solver Clingo 5.2.0 was used, which provides access to the grounder Gringo with version 5.2.0 and the solver Clasp with version 3.2.1. Hereby, all presented tests have been conducted 50 times.

7.2 Time Efficient Queries to CN5

CN5 provides the functionality to limit the number of concepts that are returned when a query is passed to it. Hereby, the default value is set to 20 concepts and the maximum value is set to 1000 concepts. Hence, this section presents the influence of the query size on the runtime of returning all edges connected to a concept and the number of queries that have to be sent to CN5. Therefore, three concepts have been chosen as representatives of differently sized concepts. The first of these concepts is the Plant, which is connected to 2393 other English concepts with a weight over 1.0 and represents large concepts. The second concept is the concept of a Dog that is connected to 1352 other concepts and represents medium-sized concepts. Finally, the concept of a Cup was chosen as the representative of small concepts,
since it is connected to 277 other concepts. Furthermore, the database of CN5 is equipped with a cache used to speed up queries containing already returned edges. Since this would influence the results, the cache has been cleared after each test. The results of these tests are shown in Figure 7.1.

The results for the concept of a Plant are shown in Figure 7.1(a). Since this concept is a very large concept, the resulting mean runtime is higher in comparison to the other concepts. Hereby, the highest runtime occurs for a number of 20 concepts per query with roughly 22.3 seconds, which can be considered as too long for the usage on real robots. Furthermore, the query for the concept of a Plant \(QP\) is split into 218 queries to CN5, which results in a high amount of traffic between the database and the KnowledgebaseCreator. Increasing the number of concepts in a query has two effects: The first effect is the reduction of the runtime. As shown in this figure, the runtime of \(QP\) is reduced exponentially when the number of concepts per query is increased resulting in a mean runtime of 1032.3 ms for \(QP\) when using the maximum value of 1000 concepts per query. Furthermore, the required queries are reduced to five, which results in a lower traffic between the KnowledgebaseCreator and CN5. Another interesting aspect to state is the low standard deviation of all chosen query sizes. In general, the standard deviation expresses the reliability of the mean as a representative of the collected data and how scattered the data is. Hereby, a low standard deviation states that the mean is a good representative for the data and that the data points are close to each other. This is the case for the mean values presented in Figure 7.1(a), which means that the runtime of the CN5 database is stable and does not vary by increasing the number of queries.

Figure 7.1(b) is a depiction of the results for the concept of a Dog. In comparison to the results for \(QP\), the runtime of the query for the concept of a Dog \(QD\) is lower. Hereby, the highest runtime was measured when choosing 20 concepts per query, which is roughly 5.3 seconds and requires 129 interactions with CN5. In comparison to the concept of a Plant, this runtime is only a quarter of this runtime, showing the influence of the number of edges a concept is connected to. Comparing the remaining query sizes of both concepts, an exponential influence on the runtime of a query can be seen, which can be caused by the complexity of searching a large part of the hypergraph of CN5. Furthermore, the maximum value of 1000 concepts per query introduces the best results for \(QD\), which is a mean value of 413.18 ms and requires only three interactions with CN5. Additionally, the results have a low standard deviation, too, which shows that the runtime of the CN5 database is stable and does not change significantly.

Conclusively, the results for the concept a Cup \(QC\) are given in Figure 7.1(c), which is the smallest of the evaluated concepts. Therefore, it introduces the lowest runtime of the presented concepts. A query size of 20 concepts results in a runtime of 875.28 ms for \(QC\) and requires 48 interactions with CN5. In comparison to the other concepts, the runtime does not decrease significantly after choosing 500 concepts per query since all related edges are returned before the maximum is reached.
7.2 Time Efficient Queries to CN5

Figure 7.1: Influence of the query size for different sized concepts.
Summarizing the results presented above, it can be stated that 1000 concepts per query to CN5 introduce the lowest runtime for large and medium-sized concepts like the concept of a Plant and a Dog, but does not further increase the runtime for small concepts, like the concept of a Cup. Since choosing 1000 concepts in a query decreases the runtime for most of the concepts as shown in Figure 7.1(c), this number of concepts has been chosen in the KnowledgebaseCreator in order to reduce the runtime of a query as much as possible. Additionally, all measurements have a very low standard deviation, showing that the mean values presented are good representatives for the data and that the database of CN5 has a reliable but exponentially increasing response time. Furthermore, the measured query runtimes show the applicability of the KnowledgebaseCreator in real scenarios. Therefore, let us consider a human user that is assigning the task to clean up a room to a robot. In order to solve this task, the robot has to extract the possible locations of the objects, which have to be put away. This includes several queries to CN5, whose runtime depends on the size and number of concepts. This runtime can range up from several hundred milliseconds for a small number of concepts up to several seconds for larger concepts of a high number of concepts. Besides this runtime, further runtimes have to be considered. In order to understand the task, the robot needs to process the task given in natural language. Furthermore, the robot needs to plan the execution of the given task. Keeping the runtime of these processes in mind, it can be stated that a robotic agent would need 10 till 15 seconds to execute the task after it has been given to it by the human user.

### 7.3 Comparison of Cached and Non-Cached Queries

CN5 is a huge commonsense knowledge database that relies on a hypergraph to represent the contained knowledge. Hereby, queries to CN5 return a part of this hypergraph in form of a set of edges. In order to improve the runtime of queries, CN5 provides a query cache, which is used to cache the edges returned from previous queries. This impacts queries that have already been given to CN5 and queries, which contain edges already cached. To evaluate the influence of this cache, the queries presented in Section 7.2 were given to CN5 without clearing the cache. In order to achieve comparable results, the queries have been formulated for the concepts of a Plant, a Dog, and a Cup. Furthermore, the number of returned concepts per query has been set to 1000. The average runtime of the tests and the resulting standard deviation are presented in Figure 7.2.

Hereby, the mean runtime of a query for the concept of a Plant is 109.86 ms, if the query has already been cached. In comparison to this, a non-cached query for the concept of a Plant has a mean runtime of 1032.3 ms, which is roughly 10 times slower than the cached query. This improvement can be observed for the other concepts, too. The cached query for the concept of a Dog has an average runtime of 54.6 ms, which is roughly 10 times faster than
7.4 Runtime Evaluation of the Inconsistency Removal

After the results of a query have been returned from CN5, inconsistent properties have to be removed. Since the runtime of a query to CN5 has already been evaluated in the previous sections, the runtime of the inconsistency removal is evaluated separately in this

---

**Figure 7.2:** Runtime of cached CN5 queries.

---

the average runtime of the non-cached query, which is 413.18 ms. Additionally, the cached query for a Cup has a runtime of 17.54 ms, which is eight times faster than the non-cached variant. The results have a low standard deviation of approximately 3 ms, which states that the runtime of a cached query does not vary. Furthermore, the low standard deviation states, that the measured values were similar and that the mean value is a reliable representative of the measured values. Additionally, the here presented results show the importance of the query cache to CN5. The usage of this cache improves the runtime of queries, which already have been given to CN5, but also improves the runtime of queries to related concepts of already stated queries, since parts of the result are already cached. Therefore, the usage of this cache has positive effects on the overall runtime of a query to CN5 and can influence the removal of inconsistencies as well, since the algorithm presented in Section 6.3 contains several queries to CN5 containing connected concepts. Furthermore, the usage of the cache improves the runtime of queries if a robotic agent is rebooted. In this case, the agent has to extract the concepts from CN5 again in order to build its knowledgebase. Since the previous queries have been saved in the cache, the runtime of this process is roughly 10 times faster since the cache is used.
section. Hereby, the cache was cleared after each test, which is done to create results that are independent of previous tests. Furthermore, the concept of a Plant is used as a representative for large concepts, since it is connected to 2393 other concepts from which 174 are marked as adjectives and therefore have to be considered during the removal of inconsistencies. As a representative for small concepts, the concept of a Cup was chosen, which is connected to 277 other concepts containing 15 adjectives. In contrast to Sections 7.2 and 7.3 the concept of a Dog, which is connected to 1352 other concepts, was not chosen as a representative because only 16 of the connected concepts are marked as adjectives. This can be compared to the number of adjectives connected to the concept of a Cup and therefore, the representative for medium-sized concepts is replaced by the concept of Air. The concept of Air is connected to 706 other concepts that contain 96 adjectives. Furthermore, several pairs of contradicting adjectives have been found during this process. These include six pairs of contradicting adjectives for the concept of a Plant and one pair for the concept of Air. No pairs have been found during the check of the concept of a Cup. The results of the test are depicted in Figure 7.3.

![Figure 7.3: Runtime evaluation of the inconsistency removal algorithm.](image)

The highest average runtime of 2035.54 ms has been measured for the concept of a Plant, which is followed by the concept of Air with a runtime of 1096.64 ms. The lowest average runtime achieved is 213.26 ms for the concept of a Cup. As shown in this figure, the runtime of the inconsistency removal depends linearly on the number of adjectives, which have to be checked during the algorithm. The worst case for this algorithm is a concept without contradicting properties. Hereby, the first step of this algorithm gathers $k$ antonyms for each of the $n$ adjectives connected to the queried concept. In the second step, these $n$ adjectives
are compared to the found antonyms and removed if they are contradicting, resulting in \( m \) remaining adjectives. As the last step, synonyms are gathered for the remaining \( m \) concepts, which are then compared with the \( k \) antonyms. In the worst case, this results in a complexity of \( O(n + 2nk) = O(n) \). Furthermore, as stated in Section 6.3 all adjectives have to be considered in the first steps of the algorithm, which causes the linear relation between the number of adjectives and the average runtime of the inconsistency removal. Furthermore, these measured runtimes have a low standard deviation, as already presented in the previous sections.

In comparison to the results presented in Section 7.2, the inconsistency removal has a higher runtime than a query to CN5, whose number of concepts is set to 1000. This is caused by the way the algorithm for the inconsistency removal is built. In the first step of this algorithm, the antonyms of all adjectives connected to the queried concept are determined by queries to CN5. This increases the runtime of this algorithm since several interactions with the database of CN5 have to be performed. These interactions can either be cached or non-cached, which depends on the queried concept. In another step, the algorithm searches synonyms for the remaining adjectives after the first removal of inconsistencies, which is done by queries to CN5, which increases the runtime. Both steps combined require a high number of rather small queries to CN5. Furthermore, this algorithm depends linearly on the number of adjectives and therefore is suitable for concepts, which are connected to a high number of adjectives.

### 7.5 Runtime Evaluation of the ASP Knowledgebase

The last step in the creation of the Meta-Knowledge and the KnowledgeBase is the translation of the consistent edges into ASP. In order to test the runtime of this step, the concepts of a Plant, a Dog, and a Cup have been used, since they are suited representatives for large, medium, and small sized concepts, respectively. Figure 7.4 is a depiction of the test results.

In this figure, the highest runtime presented is the average runtime of the concept of a Plant, which is 84.36 ms with a standard deviation of 1.96 ms. This is followed by the results for a Dog, which has a runtime of 53.84 ms and standard deviation of 1.41 ms. The lowest runtime was measured for a Cup, which is 19.8 ms with a standard deviation of 1.05 ms. In comparison to the presented results in the previous sections, the runtimes are very low, because this step does not require queries to CN5, which would increase the runtime. Furthermore, the runtime presented in Figure 7.4 depends linearly on the size of the queried concept, since every edge connected to the queried concept has to be translated into ASP. This shows the applicability of the creation of the Meta-Knowledge and the KnowledgeBase since it scales linearly with the number of edges connected to a concept and has a very low runtime compared to the other parts of this algorithm.
7 Evaluation

![Runtime evaluation of the ASP knowledgebase creation.](image)

### Figure 7.4: Runtime evaluation of the ASP knowledgebase creation.

#### 7.6 Application of the KnowledgebaseCreator in a Household Scenario

In this section, an evaluation scenario is presented that is used to test and to evaluate the usage and the runtime of the Meta-Knowledge and the KnowledgebaseCreator presented in Section 6.3. This scenario consists of 100 symbols, which represent objects and rooms in a household scenario and are associated with 38 concepts. Furthermore, the agent situated in this scenario is supposed to tidy up this scenario and therefore needs to know the usual positions of the objects to fulfill its task. The process flow of this scenario is presented in Figure 7.5.

The first step of this scenario is the extraction of symbols from the world in which the agent is situated, which is done by using a prepared set of symbols representing objects from the environment. After the symbols have been added to the knowledgebase, the interaction with CN5 is started, which creates the Meta-Knowledge and the KnowledgeBase. Therefore, the names of a symbol, e.g., bathroom for the symbol bathroom(room3) are expanded with the prefix cn5_ resulting in the following query to CN5: cn5_bathroom. This step is done for the remaining 37 concepts of this scenario. After the Meta-Knowledge and the KnowledgeBase have been created, the agent is able to use both parts to apply the extracted commonsense knowledge to the given symbols. Therefore, it checks the Meta-Knowledge for combinations of concepts, which are then grounded and solved to apply the commonsense knowledge to the symbols representing objects in the agent’s world. This enables the agent to find the usual location of the objects it is supposed to tidy up.
7.6 Application of the KnowledgebaseCreator in a Household Scenario

Figure 7.5: Process flow of the evaluation scenario.

In order to test the KnowledgebaseCreator, this scenario has been reduced to the parts that are supported by it. This includes the grounding and solving of the initial symbols, the creation of the Meta-Knowledge and the KnowledgeBase, and the grounding and solving of the related symbols. Hereby, one grounding and one solving step are required to add the symbols to the solver. Furthermore, 38 queries to CN5 are required for the concepts given by the symbols. As a last point, 84 grounding and 84 solving steps are performed in order to apply the extracted commonsense knowledge to the symbols. As stated in Section 7.1, this test scenario has been run 50 times and the cache has been cleared after every run, which resulted in an average runtime 127.18 s and a standard deviation of 3.13 s. Hereby, this result is caused by choosing an efficient way of querying ConceptNet, as presented in Section 7.2 and is positively influenced by the database cache presented in Section 7.3. Furthermore, this result shows the efficiency of the KnowledgebaseCreator, since the measured average runtime to create a big knowledgebase as presented above is relatively low. Additionally, the low standard deviation states that the process of creating such kind of knowledgebases has a reliable runtime. Furthermore, the measured runtime of roughly two minutes shows that the developed KnowledgebaseCreator is able to handle scenarios consisting of a high amount of objects from different concepts in a reasonable time. Finally, the resulting Stable Models have been checked for predicates containing the extracted commonsense knowledge in order to check the functionality of the KnowledgebaseCreator. For example, the results indicate, that Forks should be placed in a Drawer. Furthermore, the results indicate that a Cup is usually placed on a Table which answers the question given in the title: “Where is a cup and what is it good for?”. This is, e.g., expressed by the resulting predicates: \texttt{atLocation(cup1,table1)}, \texttt{atLocation(cup2,table1)}, \texttt{atLocation(fork1,drawer1)}, and \texttt{cn5_AtLocation(cn5\_cup,cn5\_table,28)}. 
8 Conclusion

In the final chapter of this elaboration, a summary of the presented work and evaluation is given in Section 8.1. Furthermore, possible expansions to the KnowledgebaseCreator and further future work are presented in Section 8.2.

8.1 Summary

The presented elaboration is divided into two parts. The first part is a graphical user interface (KnowledgebaseCreator) for the ASP grounder and solver Clingo. Hereby, the KnowledgebaseCreator provides access to all features that are provided by Clingo. These features include adding, grounding, and solving ASP programs. Furthermore, the user is able to define Program Sections and External Statements. Besides the definition of External Statements, the user is able to change their truth values and therefore is able to change the Stable Models of an ASP program dynamically. Additionally, the KnowledgebaseCreator provides the query mechanism presented in [34], which is expanded by an automatized satisfaction of the Module Property. This allows the user to formulate queries without the possibility to violate the Module Property and therefore simplifies the use of complex ASP programs consisting of several Program Sections. Besides the usage of ASP, the second part of the KnowledgebaseCreator provides access to the commonsense knowledge database CN5. Generally speaking, CN5 is a graph consisting of edges that connect concepts, e.g. a Cup, with other concepts via weighted relations. The KnowledgebaseCreator provides a query mechanism that allows its user to query different parts of CN5. This mechanism includes queries for a concept, for edges containing a concept and a relation, or for two concepts without a given relation. By using these queries, the user of this elaboration is enabled to derive commonsense knowledge from CN5. Furthermore, this elaboration provides a consistent ASP-based commonsense knowledgebase, which is extracted from CN5. This knowledgebase is divided into the Meta-Knowledge and the KnowledgeBase, which enable the user to extract consistent knowledge from CN5. Hereby, the Meta-Knowledge provides consistent weighted relations between concepts and the KnowledgeBase provides the functionality for mapping this knowledge on already defined symbols. To sum up, this elaboration provides a tool to create ASP based knowledgebases, which can be enhanced by consistent commonsense knowledge derived from CN5.
8 Conclusion

In order to evaluate the created KnowledgebaseCreator, runtime tests have been performed on several parts of this elaboration. Hereby, the most efficient way to interact with CN5 was determined by evaluating the runtime of queries to CN5 with different numbers of returned concepts per query. The results of this evaluation showed that using the maximum value of 1000 concepts per query resulted in the fastest responses and therefore was used in the KnowledgebaseCreator. Furthermore, the influence of CN5 database cache has been evaluated showing that queries to CN5 are improved if parts of them are already cached. Besides the interaction with CN5, the algorithm used to remove inconsistencies has been evaluated. Hereby, the runtime of this algorithm is higher than queries to CN5, since the algorithm requires many but small queries to CN5. The worst case of the algorithm is the absence of inconsistencies since in this case, all adjectives have to be examined. Furthermore, the translation of CN5 edges into ASP was evaluated. The results show that the translation is the fastest part in the creation of the Meta-Knowledge and the KnowledgeBase and that the runtime of this step depends linearly on the size of the concepts to translate. As a last point, the interaction with the KnowledgebaseCreator has been tested using a household scenario during the tests. The results indicate that the KnowledgebaseCreator is able to provide the solution for tasks that rely on commonsense knowledge and is able to cope with such kind of scenarios in a reasonable time.

8.2 Future Work

In this section, possible expansions and improvements for the implemented KnowledgebaseCreator are presented. Hereby, three selected improvements and expansions are shown and their advantages and drawbacks are briefly explained.

Outsourcing of the local ConceptNet 5 instance

One possible expansion and improvement of the KnowledgebaseCreator would be the outsourcing of the local instance of CN5. As already stated in Section 5.5, a local instance of CN5 requires at least 120 GB of disk space and at least 4 GB of available RAM, which are high requirements for autonomous robots. Therefore, the local instance of CN5 could be outsourced to a local server or a more powerful agent, which can be done by applying the techniques presented in [12, 39]. This would allow agents with low-performance hardware to use the KnowledgebaseCreator and therefore would provide the access to commonsense knowledge to them. In comparison to the web API available at http://api.conceptnet.io, a server instance in a local network would not require a connection to the internet. Additionally, queries would not be slowed down, since the query limit of 600 queries per minute can be removed from the local server instance. Hereby, the network load has to be kept in mind. High network load can be caused in two ways: The first way is to query a large concept,
8.2 Future Work

e.g., the concept of a Plant. Given a weight threshold of 1.0, a Plant is connected to almost 2400 other concepts, which results in large messages. The second way is the filtering of inconsistencies. This can require many, but small messages, depending on the number of properties of a concept, which could result in a flooding of the network if several agents query the local server at the same time.

**Expanding Clingo with a network interface**

In this elaboration, an adapted ASPSolverWrapper has been presented, which enables an agent to interact with Clingo by using different kinds of queries. An expansion to this wrapper is to implement a network interface for an ASP solver, which could be realized as a Cloud service. This interface could be used in two ways: The first way to use a network interface would be the outsourcing of grounding and solving to other agents or a server in the network. This allows low-performance agents or agents with high load to use Clingo without relying on its own resources. The second way of using a network interface would be to share results in a team of autonomous agents. Hereby, an agent could share the results of grounding and solving ASP programs resulting in less load for the rest of the team. This could improve the performance of the team since the agents can cooperate solving difficult problems and can exchange already found solutions. A drawback of this expansion would be an increased network load in case of large models or ASP programs. Furthermore, the integration of received models or ASP programs can cause problems, since they could contain contradicting predicates. To solve this problem, a way of handling contradicting predicates is needed in order to keep a consistent knowledgebase among the team of autonomous agents.

**Implementation of an agent using the KnowledgebaseCreator**

So far, the KnowledgebaseCreator has been tested by human users. An expansion to this elaboration is the creation of an autonomous agent that is able to interact with the Meta-Knowledge and the KnowledgeBase. In order to implement such an agent, the ALICA modeling language could be used, which provides an environment for modeling the agent’s behavior by utilizing finite state machines. Hereby, the realization of the agent could be either a pure software agent or a robotic agent that is able to interact with the environment. In the case of a pure software agent, the agent could read symbols from a user and the derived commonsense knowledge from the KnowledgebaseCreator, for example, to support the user by searching for related topics. In contrast to this, a robotic agent could be used to do household tasks. Therefore, it needs to be able to extract symbols from its environment. These symbols can then be combined with the KnowledgebaseCreator to equip the robotic agent with commonsense knowledge. This has the advantage that the agent is able to automatically gain knowledge about the symbols in its environment and perform tasks,
which would either require commonsense knowledge or the interaction with a human user. An example for this kind of task is to fetch a Cup. Hereby, let us assume that the agent knows what a Cup is and what it looks like. In order to fetch a Cup the agent now needs to know where a Cup can normally be found. As a human being, this task is easily solvable by relying on our commonsense knowledge, which includes that a Cup can normally be found in a Shelf or Cupboard that is usually located in the kitchen. An autonomous agent without this knowledge would not be able to solve this task. By providing commonsense knowledge this kind of task becomes solvable without further interaction with a human user.
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Where is a cup and what is it good for?</td>
<td>1</td>
</tr>
<tr>
<td>3.1</td>
<td>Common example of an ASP rule</td>
<td>9</td>
</tr>
<tr>
<td>3.2</td>
<td>Subgraph of CN5 showing the concept of a Cup</td>
<td>25</td>
</tr>
<tr>
<td>4.1</td>
<td>Example of a part of SenticSpace</td>
<td>28</td>
</tr>
<tr>
<td>4.2</td>
<td>Workflow of RESI</td>
<td>30</td>
</tr>
<tr>
<td>4.3</td>
<td>Symbol extraction from a given picture</td>
<td>32</td>
</tr>
<tr>
<td>5.1</td>
<td>Main window of the KnowledgebaseCreator</td>
<td>36</td>
</tr>
<tr>
<td>5.2</td>
<td>KnowledgebaseCreator after executing a query</td>
<td>37</td>
</tr>
<tr>
<td>5.3</td>
<td>Tab showing the External Statements of the current ASP program</td>
<td>38</td>
</tr>
<tr>
<td>5.4</td>
<td>Menus provided to influence the current programs and Stable Models</td>
<td>38</td>
</tr>
<tr>
<td>5.5</td>
<td>Command History containing entries for the performed commands</td>
<td>39</td>
</tr>
<tr>
<td>5.6</td>
<td>Class diagram of the command classes</td>
<td>40</td>
</tr>
<tr>
<td>5.7</td>
<td>Expansion of the query mechanism presented in</td>
<td>42</td>
</tr>
<tr>
<td>6.1</td>
<td>Antonymic properties of the concept of a Plant</td>
<td>53</td>
</tr>
<tr>
<td>6.2</td>
<td>Workflow of a CN5 query</td>
<td>55</td>
</tr>
<tr>
<td>6.3</td>
<td>Translation of a CN5 query into a URI</td>
<td>56</td>
</tr>
<tr>
<td>7.1</td>
<td>Influence of the query size for different sized concepts</td>
<td>63</td>
</tr>
<tr>
<td>7.2</td>
<td>Runtime of cached CN5 queries</td>
<td>65</td>
</tr>
<tr>
<td>7.3</td>
<td>Runtime evaluation of the inconsistency removal algorithm</td>
<td>66</td>
</tr>
<tr>
<td>7.4</td>
<td>Runtime evaluation of the ASP knowledgebase creation</td>
<td>68</td>
</tr>
<tr>
<td>7.5</td>
<td>Process flow of the evaluation scenario</td>
<td>69</td>
</tr>
</tbody>
</table>
# List of Listings

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Example of an ASP program.</td>
<td>10</td>
</tr>
<tr>
<td>3.2 Pingu is not able to fly.</td>
<td>10</td>
</tr>
<tr>
<td>3.3 Closed World Assumption formulated in ASP</td>
<td>11</td>
</tr>
<tr>
<td>3.4 ASP program used to demonstrate the grounding process.</td>
<td>12</td>
</tr>
<tr>
<td>3.5 ASP program with its reduced version.</td>
<td>13</td>
</tr>
<tr>
<td>3.6 Example for the usage of External Statements.</td>
<td>14</td>
</tr>
<tr>
<td>3.7 An ASP program divided into Program Sections.</td>
<td>15</td>
</tr>
<tr>
<td>3.8 Program Section containing a variable.</td>
<td>15</td>
</tr>
<tr>
<td>3.9 ASP query used for navigation.</td>
<td>23</td>
</tr>
<tr>
<td>3.10 Excerpt from the edges returned by CN5 for the location of a Cup.</td>
<td>26</td>
</tr>
<tr>
<td>5.1 Grounding a Program Section containing a variable.</td>
<td>41</td>
</tr>
<tr>
<td>5.2 Excerpt of a saved Command History in JSON.</td>
<td>41</td>
</tr>
<tr>
<td>5.3 ASPVariableQuery before applying the automated satisfaction of the Module Property</td>
<td>45</td>
</tr>
<tr>
<td>5.4 ASPVariableQuery after applying the automated satisfaction of the Module Property</td>
<td>45</td>
</tr>
<tr>
<td>5.5 Command lines used to install Docker.</td>
<td>48</td>
</tr>
<tr>
<td>5.6 Command lines used to install Docker-Compose.</td>
<td>48</td>
</tr>
<tr>
<td>5.7 Download CN5 and remove query limitations.</td>
<td>48</td>
</tr>
<tr>
<td>5.8 Install CN5.</td>
<td>49</td>
</tr>
<tr>
<td>6.1 Extract of the Meta-Knowledge for the concept of a Cup.</td>
<td>58</td>
</tr>
<tr>
<td>6.2 Extract of the KnowledgeBase for the concept of a Cup.</td>
<td>59</td>
</tr>
</tbody>
</table>
Bibliography


