Reasoning for Autonomous Agents in Dynamic Domains

Stephan Opfer, Stefan Jakob and Kurt Geihs
Distributed Systems Research Group, University of Kassel, Wilhelmshöher Allee 73, Kassel, Germany

Keywords: Answer Set Programming, Region Connection Calculus, Spatial Reasoning, Multi-shot Solving.

Abstract: In contrast to simple autonomous vacuum cleaners, multi-purpose robots that fetch a cup of coffee and clean up rooms require cognitive skills such as learning, planning, and reasoning. Especially reasoning in dynamic and human populated environments demands for novel approaches that can handle comprehensive and fluent knowledge bases. A promising approach is Answer Set Programming (ASP), offering multi-shot solving techniques and non-monotonic stable model semantics. Our objective is to equip multi-agent systems with ASP-based reasoning capabilities, enabling a team of robots to cope with dynamic environments. Therefore, we combined ALICA - A Language for Interactive Cooperative Agents - with the ASP solver Clingo and chose topological path planning as our evaluation scenario. We utilised the Region Connection Calculus as underlying formalism of our evaluation and investigated the scalability of our implementation. The results show that our approach handles dynamic environments and scales up to appropriately large problem sizes.

1 INTRODUCTION

During the last decade autonomous robots started to play an increasingly important role in our everyday life. They are able to vacuum-clean our living room, mow the lawn, and clean the pool. Almost all car manufacturers are developing autonomous cars. Automated guided vehicles take care of the logistics in production plants or parcel service centres. Autonomous and interactive toys become more and more intelligent.

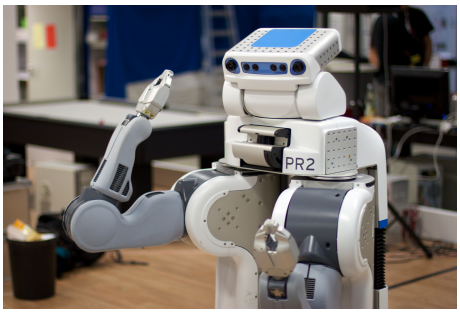

Figure 1: Domestic Service Robot PR2. (Wang, 2010).

In the research field of service robots, researchers focus on multi-purpose robots, instead of single-purpose ones. Figure 1 shows a state-of-the-art service robot, which can do everyday household tasks. The variety of those tasks tremendously increases the number of elements relevant for the robot's environmental representation and therefore, poses new challenges to the research community. Operating in human populated environments, e.g., requires the robot to handle dynamic environments. This is, because human beings insert, remove, and displace objects in the environment and are themselves moving obstacles from the robot's point of view. In order to cope with this problem, the robot needs cognitive capabilities such as learning, planning, and reasoning. Furthermore, a suitable level of abstraction is necessary to make reasoning about such complex and dynamic domains more tractable. A robot that should fetch a cup, for example, needs to know that a cup is usually situated in a cupboard in the kitchen, but does not have to know the cup's exact coordinates. Furthermore, it needs to reason about the reachability of the cup from its current position in the building, taking locked doors, obstacles, and the topology of the building into account.

A common way of tackling such problems is the symbolic knowledge representation and reasoning approach (Brachman and Levesque, 2003). Nevertheless, our experience is that most benchmarks present today[1] are designed in a one-shot fashion, i.d., continuously solving one problem after another, without

[1] http://www.satcompetition.org/ [Online; accessed December 9, 2016]
http://aspcomp2015.dibris.unige.it/ [Online; accessed December 9, 2016]

Opfer S., Jakob S. and Geihs K.
Reasoning for Autonomous Agents in Dynamic Domains.
DOI: 10.5220/0006297603400351
In *Proceedings of the 9th International Conference on Agents and Artificial Intelligence (ICAART 2017)*, pages 340-351
ISBN: 978-989-758-220-2

the possibility of using knowledge from previous problems. Our contribution is to enable multi-agent systems to continuously reason about dynamic environments by utilizing Answer Set Programming (ASP) – a non-monotonic knowledge representation and reasoning formalism (Gebser et al., 2014), suitable for multi-shot solving (Gebser et al., 2015). Furthermore, such dynamic environments demand a highly dynamic multi-agent coordination framework that in our case is provided by ALICA – A Language for Interactive Cooperative Agents (Skubch, 2013). In our preliminary work (Opfer et al., 2016) we captured the semantics of ALICA programs within an ASP logic program, in order to detect common modelling errors. This forms the basis for integrating ASP with ALICA and allows us to extend it with a general solver interface. Through this interface ALICA programs can utilise ASP-based reasoning techniques which in turn makes ALICA open for a wider set of application domains. Moreover, we had to extend the utilised ASP solver with query semantics according to (Gelfond and Kahl, 2014). In our evaluation, we chose topological path planning as our scenario. Therefore, we formulated a simplified version of the common Region Connection Calculus (Randell et al., 1992) in ASP, further denoted as RCC-4, and investigated the scalability of our RCC-4 formalisation in combination with our integration of ASP with ALICA.

The remainder of this paper is structured as follows. Section 2 introduces ALICA, ASP, and RCC-4. The integration of ALICA with ASP is described in Section 3. Furthermore, the query semantics extension is elaborated in Section 4. Section 5 provides the description of our evaluation scenario, whose results are presented in Section 6. Finally, we compare our work with other approaches in Section 7 and conclude with Section 8.

2 FOUNDATIONS

This section is divided into three subsections. In Subsection 2.1 the focus is set on concepts of ALICA that are necessary to understand in the context of this work. The same holds for Subsection 2.2 that is about the syntax and semantics of ASP. In Subsection 2.3 an explanation of the basic relations of the Region Connection Calculus 4 is given.

2.1 ALICA

The ALICA framework is designed to coordinate a cooperative team of autonomous agents. Explaining all features of this framework is beyond the scope of this work and we would like to point the interested reader to the dissertation of Hendrik Skubch (Skubch, 2013) and two supplementary publications (Skubch et al., 2011a; Skubch et al., 2011b). In this section, our goal is to explain the fundamental principles of ALICA and focus on the parts that we changed to make a wider set of general problem solvers accessible from within an ALICA program.

The ALICA framework is distributed in the sense that each agent in the team is running its own independent ALICA behaviour engine. Each behaviour engine determines the sequence of actions of the local agent while coordinating with other engines and taking the current situation as well as a given ALICA program into account. Sometimes frameworks like ALICA are also denoted as sequencer (Gat, 1998).

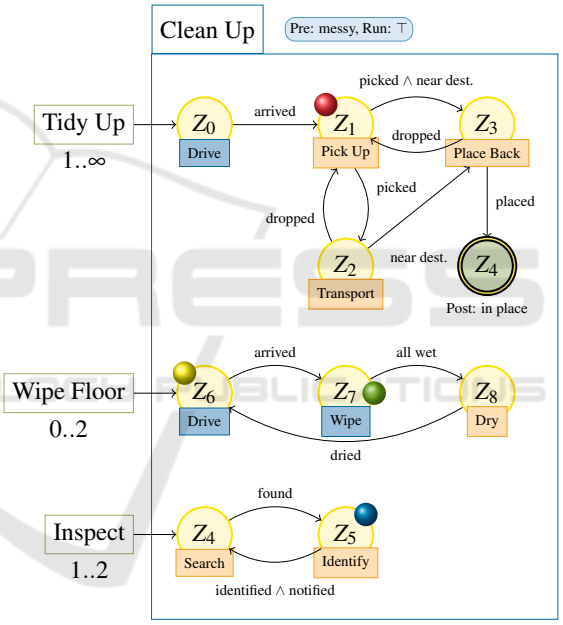

Figure 2: Simple Clean Up Plan.

An ALICA program is a special tree, whose interior nodes are plans and its leaf nodes are atomic behaviours. The *CleanUp* plan in Figure 2 is an example of such an interior node. A plan can include several states ($Z_0 \ldots Z_8$) that are connected with guarded transitions to create finite state machines (FSM). Each FSM is annotated with a task (*Tidy Up, Wipe Floor, Inspect*) and a pair of cardinalities for the minimum and maximum number of agents allowed in the corresponding FSM. Each state of a plan, except for terminal states (Z_4), can contain behaviours and plans that represent leaf or interior nodes respectively on the next level of the tree. In Figure 2 plans and behaviours are distinguished by the colour of their boxes, e.g., state Z_0 contains the plan *Drive*, which is blue,

and state Z_1 contains the behaviour *Pick Up*, which is orange. It is important to note that a plan, which is referenced in a state, is a complete plan like the *Clean Up* plan itself and therefore can include state machines with other behaviours and plans.

The coloured circles on top of some states in Figure 2 illustrate a possible global execution state of the plan. Each circle represents an agent. The red circle, for example, could be the local agent executing the *Pick Up* behaviour, while the other circles represent other agents in the team, whose corresponding behaviour engine have sent their execution state to the local agent.

In order to understand the extension of ALICA by a general problem solver interface, it is necessary to introduce the notion of ALICA plan variables. For simplicity let us modify the *Inspect* state machine from Figure 2 to create a plan on its own.

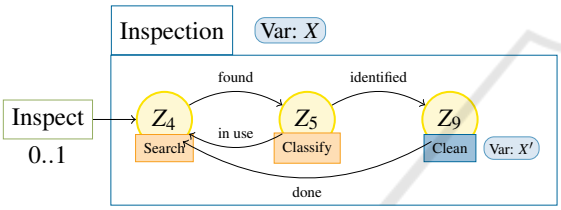

Figure 3: Inspection Plan.

The purpose of this plan, as shown in Figure 3, could be the identification of coffee cups that should be cleaned, because they are dirty and not used anymore. Therefore, an agent searches for coffee cups in state Z_4 and switches to state Z_5, in order to classify them. For remembering which cup was found in state Z_4 the plan variable X can be set to the corresponding cup and thereby influence the agent's behaviour during the rest of the plan execution. This influence can even reach to deeper levels of the plan tree, as indicated by variable X' of the *Clean* plan in state Z_9. It is possible to define variable bindings over states in an ALICA plan tree, e.g., stating that X denotes the same variable as X'. This allows to determine the agent's behaviour in the *Clean* plan, depending on which cup was found in the *Inspection* plan.

ALICA as presented in (Skubch, 2013) only provided one solver for assigning values to plan variables. The given solver addresses non-linear continuous constraint satisfaction problems and, as it was hardwired to ALICA, the applicability of ALICA for other domains was limited. Our extension of ALICA with a general solver interface tackles this issue (see Section 3).

2.2 Answer Set Programming

Answer Set Programming (ASP) is a declarative approach for solving NP-search problems. It can be seen as the result of decades of research in the areas of knowledge representation, logic programming, and constraint satisfaction. Thereby, its focus is on expressivity, ease of use, and computational effectiveness (Brewka et al., 2011). An ASP program is specified by a set of rules of the form $a_0 \leftarrow a_1 \wedge \ldots \wedge a_m \wedge \sim a_{m+1} \wedge \ldots \wedge \sim a_n$. Each a_i denotes a predicate $p(t_1,\ldots,t_k)$ with terms $t_1,\ldots,t_k$ build from constants, variables, and functions. Rules consist of three parts, namely the head a_0 , the positive part $a_1 \wedge \ldots \wedge a_m$ and the negative part $\sim a_{m+1} \wedge \ldots \wedge \sim a_n$ of the body. The semantics of the default negation $\sim$ is that of negation-as-failure. That means, $\sim a_i$ is considered to hold, if it fails to prove that a_i holds. Nevertheless, ASP also provides classic negation $\neg a_i$, whose semantics is that $\neg a_i$ holds, if a_i does not.

In order to create ASP programs, the rules have to be transformed in the following way: $\leftarrow$ is transformed to `:-`, $\wedge$ are replaced by `,` and a rule is ended by a dot. The `-` in front of `robot(X)` stands for classic negation $\neg$ and `not` in front to of `broken(X)` means default negation $\sim$. The `;` is a syntactic shortcut for creating several rules at once. Rule 1, therefore, creates `robot(chuck)`, `robot(fox)`, and `robot(lisa)`. Furthermore, Rules 1 and 2 have an empty body. So their heads are unconditionally true, and they are denoted as facts. Rule 3 makes use of the default negation $\sim$ and states that a robot can drive, as long as it cannot be proven, that it is broken (`not broken(X)`). Here `X` is a variable (starts with uppercase letter), which can be substituted with any element of the Herbrand universe of the given program.

```
1 robot(chuck; fox; lisa).
2 broken(fox).
3 canDrive(X):-robot(X),not broken(X).
4 highFailureRate :- Working
      = #count{X : canDrive(X)},
    Broken = #count{X : broken(X)},
    Working < Broken.
```

Listing 1: Robots can drive, as long as they are not broken.

The Herbrand universe of the program in Listing 2.2 only contains four constants (start with lower-case letter): {`chuck`, `fox`, `lisa`, `highFailureRate`}. Rule 4 derives the constant `highFailureRate`, if there are more broken robots than driving ones. As shown by this rule, ASP is capable of handling integers and provides aggregate functions like `#count` or `#sum` and arithmetic functions like `<` or `+`. The result of this program will state, that there are three robots

of whom `chuck` and `lisa` can drive, the constant `highFailureRate` does not occur.

State-of-the-art ASP solvers (Gebser et al., 2014; Leone et al., 2006) usually work in two steps. First they ground the program and afterwards determine all stable models of the grounded program. A program, as well as every part of it, is grounded if it does not contain any variable. In order to create a grounded program, informally speaking, the variables of each rule are replaced by each possible substitution with an element of the program's Herbrand universe. The Herbrand universe of a program is constructed from all constants and functions occurring in the program. Grounding a program that way would increase the number of rules enormously, therefore the utilised grounding algorithms try to keep the grounded program as small as possible, without altering the programs meaning. For example, Rule 3 of Listing 2.2 will not be part of the grounded program, if there is no robot available.

Solving a grounded program is often done with SAT solving techniques that are adapted to the stable model semantics of ASP. A model in ASP is a set M of ground predicates that for every rule either contains the rules head ($a_0 \in M$), or does not include all predicates of the positive part of the rule's body ($\{a_1,\ldots,a_m\} \nsubseteq M$), or contains predicates from the negative part of the rules body ($\{a_{m+1},\ldots,a_n\} \cap M \neq \emptyset$). Informally speaking, a stable model is as small as possible and contains predicates, only if they are justified by facts. For a detailed introduction into stable model semantics see (Eiter et al., 2009).

In our approach we choose the Clingo 4.5.3 ASP solver (Gebser et al., 2014), which introduces the notion of External Statements to ASP (Gebser et al., 2012; Gebser et al., 2015). External Statements in combination with program sections, explained later on, are the key concepts for enabling the query semantics described in Section 4. The External Statements are predicates marked with `#external`. These predicates are not removed from the body of a rule during grounding, even if they do not appear in the head of any rule. Furthermore, they can be explicitly set to `true` or `false`. An example for the use of External Statements is given in Listing 2.2.

```
1 #external closed(lab,hall).
2 c(lab,hall) :- not closed(lab,hall).
3 dc(lab,hall) :- closed(lab,hall).
```

Listing 2: Modeling a door using an External Statement.

This example is part of our evaluation scenario presented in Section 5 and will be used in ASPVariableQueries presented in Section 4.1. In this example `closed(lab, hall)` is marked as an External Statement and is therefore initially set to `true`. Given the predicate `closed(lab, hall)`, the head of Rule 3 holds and the predicate `dc(lab, hall)` is part of the stable model. The head of Rule 2 cannot be derived since `closed(lab, hall)` is set to be true. Usually, during the grounding procedure Rule 2 would be removed, because its body cannot be derived. However, since this rule contains an External Statement, it stays part of the grounded logic program. Therefore, it is possible to change a logic program without another grounding step by using External Statements. Furthermore, the size of the stable model does not change. For example, if the External Statement `closed(laboratory, hall)` is set to `true` the laboratory and the hall are disconnected from each other. If it is set to `false` the predicate `dc(lab, hall)` no longer holds but `c(lab, hall)` can be derived, thus the rooms are connected.

Additionally, Clingo introduces program sections (Gebser et al., 2014). Program sections are used to divide a logic program into different parts, which can be grounded separately. An example is given in Listing 2.2.

```
1 #program rcc4_composition_table.
2 dc(X,Z):-pP(X,Y),dc(Y, Z),X != Z.
3 #program rcc4_facts
4 pP(office1,offices).
5 dc(offices,studentArea).
```

Listing 3: Usage of program sections.

This example contains two program sections identified by the `#program` prefix, i.d., `rcc4_composition_table` and `rcc4_facts`. Moreover, the order in which they are grounded influences the facts, which appear in the stable models. If the section `rcc4_composition_table` is grounded before the section `rcc4_facts`, the resulting model would only contain the facts `pP(office1, offices)` and `dc(studentArea, offices)`. If section `rcc4_facts` is grounded first, the model will contain these two facts before the program section `rcc4_composition_table` is grounded. Once this section is grounded the model will additionally contain the fact `dc(office1, studentArea)`, since the body of this rule holds for the grounding of `X` by `office1`, `Y` by `offices`, and `Z` by `studentArea`.

2.3 Region Connection Calculus

In this section the base relations of the Region Connection Calculus 4 (RCC-4) are shown, which is based on the Region Connection Calculus 8 presented by Randell et al. in (Randell et al., 1992). We have

been inspired to use RCC-4 instead of using RCC-8 by the implementation of RCC-4, that is provided by the QSRlib Foundation (Gatsoulis et al., 2016). These calculi are commonly used in qualitative spatial reasoning and will be used to model our evaluation scenarios in Section 5. The foundation of the relations is the binary relation C(x,y), which expresses that two spatial regions of unknown size are connected. Informally speaking, they share at least one common point. Furthermore, C(x,y) is reflexive and symmetric. By using the C(x,y) relation four base relations are defined and shown in Figure 4.

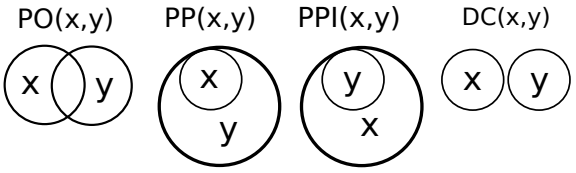

Figure 4: RCC-4 base relations.

Two regions are partially overlapping (PO), if they are connected (C), meaning they share a common point, region or part of their border. Region x is a proper part (PP) of Region y, if y contains Region x, which means that Region x is connected to y and no part of x is outside the border of y. Since this relation is not symmetric the inverse relation PPI is included as well. Additionally, two regions are disconnected (DC) if they do not share a common point. By using the composition table shown in Table 1 the transitive relations between Region x and z can be derived, given the relations between the pairs (x,y) and (y, z). Hereby, $*$ denotes that all four relations can hold. For example, if Region x is a proper part (PP) of Region y and Region y is disconnected (DC) from Region z, it can be derived that Region x is disconnected (DC) from Region z.

Table 1: RCC-4 Composition Table.

	DC	PO	PP	PPI
DC	*	DC PO PP	DC PO PP	DC
PO	DC PO PPI	*	PO PP	DC PO PPI
PP	DC	DC PO PP	PP	*
PPI	DC PO PPI	PO PPI	PO PP PPI	PPI

An example using RCC-4 to model a building is given in Section 5. Furthermore, the RCC-4 relations can be used to model the relation of objects to each other. For example, in the domain of domestic service robots RCC-4 can be used to model objects on a table without defining their exact positions, e.g., a cup and a plate could be proper parts of the table and could be disconnected from each other, if they are not touching each other.

3 EXTENDING ALICA WITH A GENERAL SOLVER INTERFACE

In this section we describe the extension of the ALICA framework with a general solver interface, in order to integrate different solvers into the ALICA framework. By now, ALICA has only been able to use a gradient solver, which is able to solve non-linear continuous constraint satisfaction problems.

In order to use other solvers besides this solver an interface `IConstraintSolver` has to be created. This interface includes four classes and provides two access methods. Every solver which is added by this interface has to inherit from the `IConstraintSolver` interface. Furthermore, solvers constraint classes derived from the `Variables` interface with a set of constraints composed from different `Terms`. Finally, the `ProblemDescriptor` encapsulates the constraints, that have been build by using `Terms` and provides possible restrictions for the range of the `Variables`. As already mentioned, the interface provides two access methods. The first method is named `existsSolution`. This method checks if a solution can be found and returns the corresponding truth value. The second method is named `getSolution`. Instead of checking if a solution can be found, this method actually calculates the solution for the given `Variables` and `ProblemDescriptor`. The calculated results are returned in a list of values defined by the solver. In this list the results are ordered in the same way the `Variables` are ordered, when they are given to the solver, so that the caller of this method can find the requested values.

For example, the gradient solver returns continuous (`double`) values or an ASP solver could return a set of predicates. Furthermore, the `Variables` utilise the hierarchy of ALICA plans. A `Variable` defined in a plan can be constrained by additional `Terms`, if the identical `Variable` is used by a plan lower in the plan tree. For example, in the domain of service robots, the robot has to get a cup of coffee from the kitchen. Therefore, the robot's navigation goal could be constraint to the area of the kitchen. In a plan, deeper in the tree, the robot's task could be defined in

detail. This would add additional constraints to the navigation goal, for example, that a robot has to be in a specific area in front of the coffee machine to place the cup in the machine. This is an advantage of the ALICA framework, since additional constraints on a `Variable` only have to be considered if an agent reaches the specific plan. As a consequence of this hierarchy the time the solvers takes to finish can be reduced, since complex constraints have only to be solved if an agent reaches plans deep in the plan tree.

So far, the presented interface has been used to integrate three different solvers. The first solver is the gradient solver, which was a part of the original ALICA framework and has been adapted to the new interface. Additionally, an ASP solver, which is presented in this work, is integrated by using this interface. Furthermore, Witsch presents in (Witsch, 2016) a middleware that enables a decision making process for a group of robots. This middleware uses the presented interface to exchange variables and proposals between agents.

4 EXTENDING CLINGO WITH QUERY SEMANTICS

This section is divided into two subsections. The created query structure is presented in Subsection 4.1. Furthermore, an ASPQuery's workflow is depicted in Subsection 4.2.

4.1 Query Structure

To enable ASP queries a data structure is needed that can be used by ALICA behaviours. This data structure is named ASPQuery. Following the interface mentioned above, an ASPQuery forwards an ASPTerm, which constraints an ASPVariable to the solver. The ASPTerm provides six fields and allows the user to define a set of ASP rules, which are part of a given ASP program section. These rules are then added to the solver's logic program and are used to derive knowledge from the stable models. Furthermore, it is possible to add facts to the ASPTerm in order to add further information to the stable models of the ASP solver. For example, to add information received from other robots. Moreover, a dictionary is used to change the truth value of the logic program's External Statements. Additionally, the lifetime of the query is given, that denotes, how long a query should stay part of the solver's logic program. To ensure this functionality, each query utilises the information given in the ASPTerm to create a unique External Statement, that is added to each rule and fact. As long as the query's lifetime is not expired this External Statement is set to `true` and the rules and facts are part of the logic program. Once the query's lifetime is expired, this External Statement is set to `false`, thus removing the rules and facts form the logic program. As a last point, the query's type is given. An example for the use of an ASPTerm is given in Listing 4.1.

```
1 term.rule(goalReachable(X) :-
      reachable(X,Y),g(X),s(Y).);
2 term.addFact(g(r1405B).);
3 term.addFact(s(r1411).);
4 term.programmSection(ds);
5 term.externals(doors);
6 term.lifeTime(1);
7 term.type(Variable);
```

Listing 4: Example of an ASPTerm.

Listing 4.1 is part of an ASP navigation, which will be explained in detail in Section 5. This example is used to query, if room `r1405B` can be reached from a given starting point `r1411`. Therefore, a rule and two facts are added to the program section `ds` (Distributed Systems). Since the resulting query will add rules and facts to the ASP program, this query can change the stable models of the ASP solver. Therefore, the type is set to `ASPQueryType::Variable`. Furthermore, External Statements and their truth values are added, which express if a specific door in the evaluation scenarios is open or closed. Finally, the query's lifetime is given.

The class model of the possible query types is depicted in Figure 5.

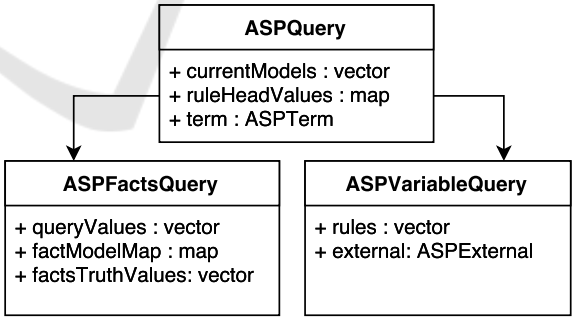

Figure 5: Classmodel of ASPQueries.

The base class ASPQuery provides three fields for the derived classes. The list `currentModels` is a placeholder in which the solver's stable models are saved after the solving process is finished. Additionally, the dictionary `ruleHeadValues` maps the rules heads of an ASP query to their grounded instances in the stable models. Furthermore, it contains an ASPTerm, that has been described above and encapsulates the rules and facts, which should be added to the solver's logic program.

The first class derived from ASPQuery is the ASPFactsQuery. In our approach the ASPFactsQuery provides the query functionality described in (Gelfond and Kahl, 2014). Gelfond et al. present a query structure that is used to check, if a ground predicate is part of the solver's stable models. Hereby the query returns `true`, if the predicate is part of the stable models, `false`, if the classical negation is part of the stable models and `unknown` if neither is part of the stable models. Therefore, the ASPFactsQuery is used to filter the current stable models for given facts, which are saved in the list `queryValues`. If there is a model that contains one of these facts, it is added to the `factModelMap`. Furthermore, the list `factsTruthValues` is used to get the corresponding truth value for each queried predicate. This list contains `true` for a positive predicate, `false` for a predicate with classical negation and `unknown` otherwise.

The second class derived from ASPQuery is the ASPVariableQuery, that is used to add rules and facts to an ASP program in order to derive knowledge from the solver's stable models. This kind of queries have the following form: $query(X_1,\ldots,X_n)$, where $X_1,\ldots,X_n$ are variables. The answer to this query is a sequence of ground terms $t_1,\ldots,t_n$ where $query(t_1,\ldots,t_n)$ is part of the solver's stable models. In order to enable such kind of queries, the ASPVariableQuery adds rules and additional facts to the solver's program, which are part of the ASPTerm. Hereby the rule heads have the form presented above. During the creation of this query, a unique External Statement and program section are created. This External Statement is added to the rules' bodies. Additionally, the facts are expanded to rules, in which the facts are used as the head of the rule and the External Statement as the body. The newly formed rules are added to the solver's logic program in the created program section, which is then grounded. Once the solver has finished the solving process, the results of the queried rules are returned to the querying ALICA behaviour. As soon as the `lifeTime` of the query is expired the External Statement is set to `false`, which removes the rules and facts from the solver's program and the stable models.

4.2 Workflow of Queries

In order to use the described ASPTerm and ASPQueries, the query structure has to be accessed by the ALICA behaviours and has to be given to the Clingo ASP solver. Therefore, a wrapper has been created, that encapsulates the Clingo ASP solver. This wrapper, named ASPSolverWrapper, is registered at the ALICA framework by using the interface explained in Section 3 and can be used by ALICA behaviours. The wrapper and its workflow are depicted in Figure 6.

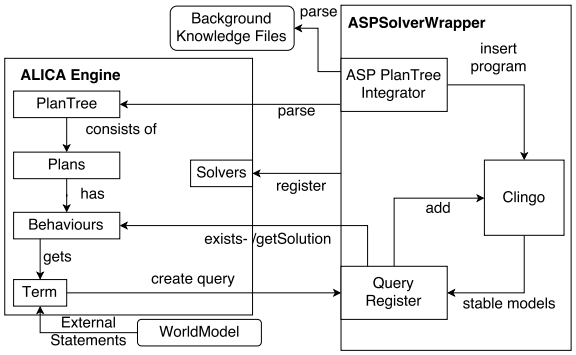

Figure 6: Query Workflow.

As mentioned in Subsection 2.1, an ALICA program consists of a directed acyclic plan tree. The ASPTerms needed to formulate queries are created inside the runtime conditions of such plans. Once the ASPTerm is created (example given in Listing 4.1), it can be used by an ALICA behaviour to formulate a query. Thereby, the External Statements' truth values are given by a worldmodel class, which encapsulates data that was perceived by an agent or received from other agents, e.g., the state of a door. After the ASPQuery has been formulated, it is registered at the wrapper, that passes it to the ASP solver Clingo. In case of an ASPVariableQuery, the wrapper adds the rules and facts to the solver's ASP program. Additionally, the ASPPlanTreeIntegrator parses the ALICA program's plan tree to enable reasoning about its structure, which is done by rules given in background knowledge files. These files, for example, contain rules to detect malformed ALICA programs. The corresponding ASP rules are presented in (Opfer et al., 2016). The ASPPlanTreeIntegrator is only used for the first registered query, since the predicates stay part of the following stable models as soon as they have been grounded. Furthermore, this increases the runtime of the first query but reduces the runtime of the following queries, since this program section has not to be grounded any more. Once all program parts (queries, background knowledge, and plan structure) have been added, the Clingo module grounds and solves the program. The hereby derived stable models are passed to the registered queries and saved to enable further use by other parts of the ALICA framework, especially the ALICA behaviours. Finally, the results are returned to the ALICA behaviour via two methods defined by the created interface. The first method is named `existsSolution`, which checks the truth value of the query without returning stable models or ground predicates. This

method can for example be used in combination with an ASPFactsQuery, which checks if an ASP predicate is part of at least one or all stable models, since the caller is only interested if the queried fact is part of the stable models. The second method is named `getSolution`. This method is used to return the derived stable models to the querying ALICA behaviour. This method can for example be used in combination with an ASPVariableQuery, where rules and facts are added. Since the ASPVariableQuery adds additional rules and facts to the ASP program the resulting stable models are returned to the querying ALICA behaviour. After the results are returned, the ALICA behaviour can react to the changes in the model. By returning the calculated results to the ALICA behaviour, the workflow of a query is finished and the queries lifetime is reduced by one. After this process, the ALICA behaviour can create the next query.

5 EVALUATION SCENARIO

Our approach of handling dynamic domains will be explained by using the following scenario. The scenario is placed in the Distributed Systems Department of the University of Kassel. A map of this department was created by using a TurtleBot produced by Willow Garage (WillowGarage, 2010), which is a small service robot equipped with a 3D camera sensor. Additionally, we equipped the TurtleBot with a 2D laser scanner in order to create a map of the department. Our adapted version of the TurtleBot is shown in Figure 7 and an annotated version of the resulting map is shown in Figure 8.

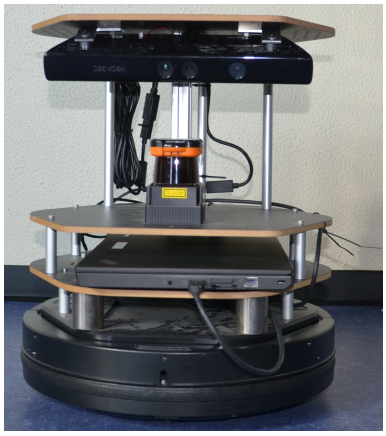

Figure 7: Adapted Version of a TurtleBot.

This map consists of 19 rooms and is divided into seven `areas`, which are highlighted in different colours. These areas include from left to right `studentArea` (red), `mainHallA` (black), `workshop` (green), `offices` (blue), `mainHallB` (purple), `utility` (yellow) and `organization` (orange). Additionally, 56 points of interest (POI), examples marked with dots, are placed on this map, e.g., different workplaces or the coffee machine. The robot's position is marked by a circle. The relations between the areas, rooms and POIs are modelled using the Region Connection Calculus 4. Hereby, a POI is a `properPart` of a specific room and `disconnected` to all other rooms. Rooms are either `partialOverlapping` with other rooms, `properPart` of areas or `disconnected` from both. Areas can either be `partialOverlapping` or `disconnected`. By using these relations the Distributed Systems Department has been modelled. Additionally, we used External Statements to model doors between two `partialOverlapping` rooms, as shown in Listing 2.2. The use of External Statements enables our approach to change the ASP solver's logic program without an additional grounding step after the initial grounding. This should decrease the runtime for answering ASPQueries and keep the models' size stable since no additional predicates have to be added to open or close doors. Moreover, we implemented an ALICA behaviour, which uses an ASPFactsQuery to check if the robot's goal position (cross) is reachable from its current position. Furthermore, we implemented a simple path planning approach relying on a transitive closure defined by the predicate `reachable(X,Y)`. The transitive closure to check if a goal is reachable is shown in Listing 5.

```
1 reachable(X,Y):-pO(X,Y),r(X),r(Y).
2 reachable(X,Y):-pO(X,Y),a(X),a(Y).
3 reachable(X,Y):-pP(X,Y),r(X),a(Y).
4 reachable(X,Y):-reachable(Y,X),X!=Y.
5 reachable(X,Z):-reachable(X,Y),
      reachable(Y,Z),X!=Y,Y!=Z,X!=Z.
```

Listing 5: Transitive closure of reachable relation.

Hereby, a room `r(X)` or area `a(X)` is `reachable` from another room `r(Y)` or area `a(Y)` if they are partial overlapping `pO(X,Y)`, which is defined by Rule 1 & 2. Furthermore, a room is `reachable` from an area if the room is a `properPart` (`pP`) of the area (Rule 3). In addition, the `reachable` relation is symmetric (Rule 4) and transitive (Rule 5).

6 EVALUATION

In this section the evaluation results are presented and discussed. Therefore, the presented evaluation scenario has been modelled in two ways. The first one

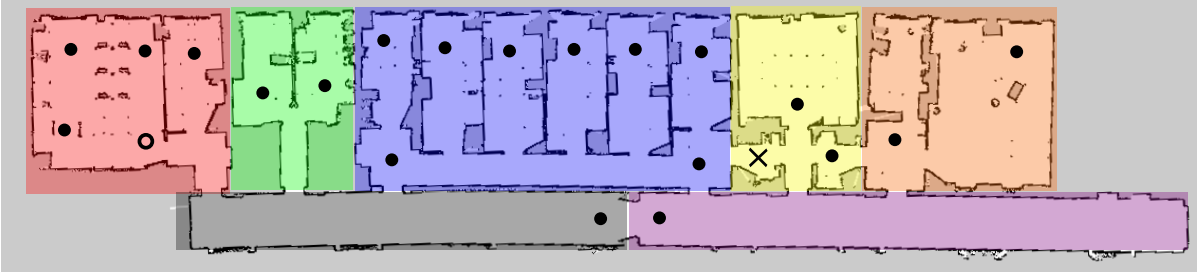

Figure 8: Map of the Distributed Systems Department.

purely relies on modelling the department by facts, named `noExt`. The second way was presented in Section 5 is named `Ext`. Additionally, both ways of modelling the department use the transitive closure to enable a simple path planning approach that is used to check if a room or POI is `reachable` from another room, area, or POI. In Figure 8 the robot's starting position is marked by a circle and the goal is marked by a cross. The path leaves the `studentArea` and follows `mainHallA` to reach `mainHallB` since the door from `mainHallA` to the `offices` is closed. From `mainHallB` the path will enter the `offices` through door `d` and finally reaches the goal, which is situated inside the `utility` area. Door `d` is the solely open door in `mainHallB` and is opened and closed via an External Statement to simulate a change in the environment.

To evaluate the wrapper and the implemented query structure, we use the presented scenarios in the following way: During the first step the logic program is grounded and a navigation query is solved. Hereby two rooms are selected and checked if they are reachable from each other, which is not possible in this step. The second step is a purely solving step to be able to compare the time needed. Step three is used to "open a door", which means either a new grounding step (`noExt`), the change of an External Statement (`Ext`) or to create a new solver instance. In this step the navigation between the rooms is possible. The last step is a purely solving step again.

The evaluation results using model `Ext`, `noExt`, and instantiating a new solver are presented in Figure 9. All measurements include grounding and solving time and show a very low standard deviation of 2.8 ms for the initial grounding with `Ext` and 0.18 ms when using `noExt` or instantiating a new solver, which shows that the measured runtimes are stable and will not show significant variations in time.

The initial grounding and solving is the most time consuming operation and lasts 107.6 ms when using `Ext` and 47.4 ms when using `noExt` or a creating a new solver instance. This is caused by additional steps during the first grounding, which include

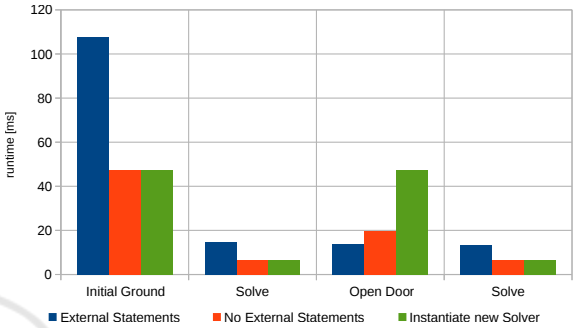

Figure 9: Comparison of different modelling approaches.

the parsing of the ALICA plan tree and grounding the resulting predicates. Additionally, these steps include the verification of the well-formedness of this plan tree by ASP Rules and the parsing and grounding of the corresponding model of the Distributed Systems Department map. The difference in time between these two results is caused by the way the department is modelled. Without the use of External Statements `noExt` each connection between two rooms is modelled by a single predicate. When using `Ext` the connection between two rooms is modelled with an External Statement and two rules expressing that a door can be either opened or closed. This increases the logic program's size which causes a longer grounding and solving time. The test results, when using `Ext`, show no difference in runtime when solving or changing the truth value of an External Statement. Both operations take roughly 13.8 ms. Additionally, the stable model's size is not affected by changing the truth value of an External Statement. In comparison, when using `noExt` or when using a new solver instance, solving and changing a door state differ in runtime. Solving without any change in the model will take 6.5 ms when using `noExt` but changing a part of the model will last 19.6 ms, since a new program section has to be grounded. Furthermore, this increases to size of the stable model's size by 156 facts, which is the number of relations between the rooms of the Distributed Systems Department. In contrast to `noExt` creating a solver differs in runtime regarding a change in the model, since a initial grounding step

has to be performed for every change resulting in an grounding time of 47.4 ms.

Due to the fact that the ALICA framework step frequency is usually set to 30 Hz, an ALICA behaviour can formulate up 30 queries per second. In Figure 10 all three methods for modelling the department are compared with respect to the amount of changes per second.

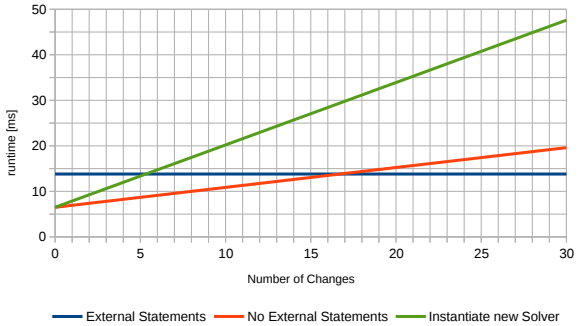

Figure 10: Comparison of measured time regarding changes in the model.

The x-axis shows the number of changes in the model and the y-axis the queries average runtime in 30 ALICA framework steps. Since there is no difference in solving and changing a value when using `Ext` the blue line is constant. In comparison to this the runtime of `noExt` increases when changes are made, since a new program part has to be grounded for each change. Both lines intersect at 17 changes. Corresponding to this result, we may conclude that `noExt` is suitable, if the modelled scenario is stable and does not change often. In dynamic scenarios we suggest the use of External Statements, since changes do not affect the runtime. Furthermore, when using `noExt` the stable model's size increases with every change made. In contrast, when using `Ext`, the models size stays the same. Therefore, we recommend to use `noExt` only for stable scenarios, since the model size will grow rapidly and will slow down the process of handling the models. This could be countered by discarding and creating a new ASPSolver instance after a few changes. This method is depicted by the green line, which intersects the blue line by six changes and is the slowest solution. Therefore, we suggest the use of `Ext`.

Additionally, we evaluated the scalability of the Region Connection Calculus 4. Therefore, we used the navigation scenario presented in 4 and expanded the number of rooms, starting from 100 rooms and ending with 1700 rooms. Furthermore, we tested different percentages of connection between the rooms ranging from 25 % to 100 %. The results are shown in Figure **??** and 12.

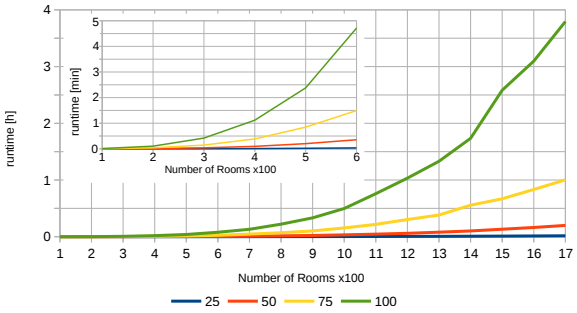

Figure 11: Runtime of the initial grounding.

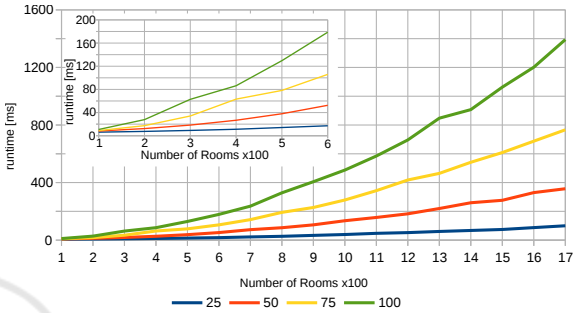

Figure 12: Runtime of the solve step.

As you can see, both runtimes increase exponentially with rising percentage and number of rooms. Hence, the collection of evaluation data was stopped at 1800 rooms with 100 % connection, since the grounding step did not stop after four hours and was aborted. Considering the exponential increase in runtime, we suggest not to use the Region Connection Calculus 4 for transitive closure based path planning in dynamic domains, if more than 600 regions have to be considered. We suggest this number, since after an initial grounding of roughly five minutes, the robot can react to changes in the knowledge base with 5 Hz, which we consider suitable for a dynamic domain.

7 RELATED WORK

Besides the ALICA framework other domain independent frameworks could be used to integrate an ASP solver for the use in different robotics scenarios. These frameworks and further related work are presented in this section.

One of these frameworks is `DyKnow` presented in (Heintz et al., 2010). The ALICA framework provides a domain independent framework to model the behaviour of multiple agents. These agents can collect domain specific information and knowledge in an unspecified way and use these to adapt their behaviours. In comparison to this `DyKnow` focuses on the distributed collection and distribution of varying

kinds of data, that includes raw sensor values, processed sensor values or even predicates, which hold between objects recognised in the world. The needed processes to collect and to process the data are specified with the knowledge processing language provided in (Heintz et al., 2010). These processes provide collected data respectively derived information and knowledge which can be used by multiple agents. Furthermore, these processes can provide knowledge about objects and the relations between them, which could be passed to a solver to allow reasoning about them. This can be compared to the knowledge base given in our wrapper and the domain specific knowledge given in ALICA behaviours. In contrast to the ALICA framework `DyKnow` does not provide the use of hierarchically constrained variables that can be used to formulate queries of increasing complexity to an ASP solver.

Another framework is `KnowRob` presented in (Tenorth and Beetz, 2013). The central part of this framework is the knowledge base. This knowledge base provides a knowledge store and access methods for retrieving information saved in Prolog relations. The "virtual knowledge base" is an extension to a robot's knowledge base by computation of abstract representations only when the corresponding data is queried and by forwarding queries to parts of the robot which can provide better answers. In comparison to the ALICA framework both use a declarative programming language to create the knowledge base. Furthermore, queries can increase the knowledge base during the computation of their answer. But in contrast to the ALICA framework with the included ASP wrapper `KnowRob` does not support multiple agents.

Furthermore, in (Gebser et al., 2015) one-shot solving is compared with multi-shot solving based on External Statements. Hereby they used benchmarks given by the Fifth ASP Competition and support our results regarding External Statements in dynamic domains. Nevertheless, they always investigated External Statements in the context of expanding universes. According to our knowledge, our work is the first investigating the advantages of External Statements in the context of dynamic universes of almost constant size.

8 CONCLUSION AND FUTURE WORK

In this paper we presented the integration of the Clingo ASP solver with the ALICA framework. Therefore, an interface has been created, that can be used with several solvers. Additionally, the integrated ASP solver has been wrapped to enable queries against resulting models and temporarily adding rules and facts. Several tests have been performed in order to evaluate the runtime of a query. The results showed that both ways of modelling (with or without External Statements) respond in less than 20 ms and therefore can be considered as suitable for dynamic domains. Nevertheless, we prefer to use External Statements, since they do not increase the size of the resulting stable models when the environment changes. Another option to avoid an increasing model size is to create a new solver instance whenever something changes. However, this would significantly increase the query runtime, making this option unfavourable starting from 5 changes per second. Therefore, we propose the use of External Statements to model dynamic domains, such as human-populated service robotic domains. Furthermore, we evaluated the scalability of the Region Connection Calculus 4 using the example of determining the transitive closure of the reachability relation. From the results, we may conclude that the calculus scales up to a number of 600 regions, since after an initial grounding time of roughly five minutes an agent can still query its knowledge base at a rate of 5 Hz.

In our future work, we will further investigate the performance of External Statements in dynamic domains by applying the presented approach in various scenarios. This investigation will be joined with knowledge-based collaboration between multiple agents. Currently each agent has its own knowledge base that is independent from all other knowledge bases. We want to allow agents to query knowledge bases of other agents. In the scenario from Section 5, for example, an agent that is unable to reach its destination could ask other agents for open doors, instead of searching for open doors by itself.

Another aspect of knowledge-based collaboration is about global consistent stable models. In ASP, it is possible that several valid models exist, but it is often desirable that a team agrees on the same or a similar model. Therefore, agents could exchange relevant parts or even complete models between each other, in order to choose the local model that is most similar to the parts received from other agents.

REFERENCES

Brachman, R. J. and Levesque, H. J. (2003). *Knowledge Representation and Reasoning*. Morgan Kaufmann, Morgan Kaufmann Series in Artificial Intelligence.

Brewka, G., Eiter, T., and Truszczyński, M. (2011). Answer set programming at a glance. *Communications of the ACM*, 54(12):92–103.

Eiter, T., Ianni, G., and Krennwallner, T. (2009). Answer Set Programming: A primer. In Tessaris, S., Enrico Franconi, Eiter, T., Gutierrez, C., Handschuh, S., Rousset, M.-C., and Schmidt, Renate A. Schmidt, editors, *Reasoning Web. Semantic Technologies for Information Systems*, volume 5689. Springer, Berlin and New York.

Gat, E. (1998). On Three-Layer Architectures. In Kortenkamp, D., Bonasso, R. P., and Murphy, R., editors, *Artificial Intelligence and Mobile Robots: Case Studies of Successful Robot Systems*, pages 195–210. MIT Press, Cambridge, USA.

Gatsoulis, Y., Alomari, M., Burbridge, C., Dondrup, C., Duckworth, P., Lightbody, P., Hanheide, M., Hawes, N., Hogg, D. C., and Cohn, A. G. (2016). QSRlib: A Software Library for Online Acquisition of Qualitative Spatial Relations from Video. In Bredeweg, B., Kansou, K., and Klenk, M., editors, *Proceedings of the 29th International Workshop on Qualitative Reasoning*, pages 36–41.

Gebser, M., Grote, T., Kaminski, R., Obermeier, P., Sabuncu, O., and Schaub, T. (2012). Answer set programming for stream reasoning. In Eiter, T. and McIlraith, S., editors, *Proceedings of the Thirteenth International Conference on Principles of Knowledge Representation and Reasoning (KR'12)*, volume 13, pages 613–617. AAAI Press.

Gebser, M., Janhunen, T., Jost, H., Kaminski, R., and Schaub, T. (2015). ASP Solving for Expanding Universes. pages 354–367. Springer International Publishing.

Gebser, M., Kaminski, R., Kaufmann, B., and Schaub, T. (2014). Clingo = ASP + Control: Preliminary Report. In M. Leuschel and T. Schrijvers, editors, *Technical Communications of the Thirtieth International Conference on Logic Programming (ICLP'14)*, volume 14(4-5).

Gelfond, M. and Kahl, Y. (2014). *Knowledge representation, reasoning, and the design of intelligent agents: The answer-set programming approach*. Cambridge University Press, Cambridge, USA.

Heintz, F., Kvarnström, J., and Doherty, P. (2010). Bridging the Sense-Reasoning Gap: DyKnow – Stream-based Middleware for Knowledge Processing. *Advanced Engineering Informatics*, 24(1):14–26.

Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., and Scarcello, F. (2006). The DLV System for Knowledge Representation and Reasoning. *ACM Transactions on Computational Logic*, 7(3):499–562.

Opfer, S., Niemczyk, S., and Geihs, K. (2016). Multi-Agent Plan Verification with Answer Set Programming. In Aßmann, U., Brugali, D., and Piechnick, C., editors, *Proceedings of the Third Workshop on Model-Driven Robot Software Engineering*. ACM.

Randell, D. A., Cui, Z., and Cohn, A. G. (1992). A Spatial Logic based on Regions and Connection. In Nebel, B., Rich, C., and Swartout, W., editors, *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning*, volume 92, pages 165–176, San Francisco, CA, USA. Morgan Kaufmann.

Skubch, H. (2013). *Modelling and Controlling of Behaviour for Autonomous Mobile Robots*. Springer Vieweg, 1 edition.

Skubch, H., Saur, D., and Geihs, K. (2011a). Resolving Conflicts in Highly Reactive Teams. In Luttenberger, N. and Peters, H., editors, *17th GI/ITG Conference on Communication in Distributed Systems (KiVS 2011)*, volume 17 of *OpenAccess Series in Informatics (OASIcs)*, pages 170–175, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

Skubch, H., Wagner, M., Reichle, R., and Geihs, K. (2011b). A Modelling Language for Cooperative Plans in Highly Dynamic Domains. In van de Molengraft, M. and Zweigle, O., editors, *Special Issue on Advances in Intelligent Robot Design for the Robocup Middle Size League*, volume 21, pages 423–433.

Tenorth, M. and Beetz, M. (2013). KnowRob: A Knowledge Processing Infrastructure for Cognition-Enabled Robots. *International Journal of Robotics Research*, 32(5):566–590.

Wang, J. (2010). PR2 at the Intelligent Autonomous Systems Group, Technical University Munich. https://www.flickr.com/photos/jiuguangw/5136649984 [Online; accessed December 9, 2016].

WillowGarage (2010). Turtlebot Website. http://www.turtlebot.com/ [Online; accessed November 8th, 2016.].

Witsch, A. (2016). *Decision Making for Teams of Mobile Robots*. Dissertation, University of Kassel, Kassel, Germany.