



Reasoning for Autonomous Agents in Dynamic Domains: Towards Automatic Satisfaction of the Module Property

Stephan Opfer^(✉), Stefan Jakob^(✉), and Kurt Geihs^(✉)

Distributed Systems Research Group, University of Kassel,
Wilhelmshöher Allee 73, Kassel, Germany
{opfer,s.jakob,geihs}@uni-kassel.de

Abstract. State-of-the-art service 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. Our long-term objective is an autonomous robotic team that is capable of handling dynamic and domestic environments. Therefore, we combined ALICA – A Language for Interactive Cooperative Agents – with the Answer Set Programming solver Clingo. The answer set programming approach offers multi-shot solving techniques and non-monotonic stable model semantics, but requires to keep the Module Property satisfied. We developed an automatic satisfaction of the Module Property and chose topological path planning as our evaluation scenario. We utilised the Region Connection Calculus as the 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 while automatically satisfying the Module Property.

Keywords: Answer Set Programming · Region Connection Calculus
Module Property · Multi-shot solving

1 Introduction

Due to the development of autonomous vacuum cleaners, lawnmowers and pool cleaners, autonomous robots stepped into our everyday life and also in other areas similar developments are currently taking place. Automated guided vehicles that take care of the logistics in production plants or parcel service centres are already commonly used [1], but also most car manufacturers are developing autonomous cars [2]. Even autonomous and interactive toys become more and more intelligent and conquer our children's rooms [3, 4].

In contrast to these single purpose devices, researchers in the field of service robots focus on multi-purpose robots. Figure 1 shows a state-of-the-art service robot, which can do everyday household tasks. The variety of those tasks pose



Fig. 1. Domestic service robot [5].

new challenges to the research community because the number of elements relevant to the robot's environmental representation is tremendous. Further challenges arise from human populated environments, that require robots to handle very dynamic situations, because human beings insert, remove, and displace objects in their environment. Furthermore, human beings are themselves dynamic obstacles from the robot's point of view. In order to cope with such environments, robots need cognitive capabilities such as learning, planning, and reasoning. Reasoning about such complex and dynamic domains, e.g., requires a suitable level of abstraction in order to make the reasoning more tractable. Consider the example of a robot that should fetch a cup. Reasoning about possible positions of the cup and how the robot could get there, should not be polluted by the cup's exact coordinates or the robot's trajectory planning. The robot should only take its current position in the building, locked doors, obstacles, and the building's topology into account.

Symbolic knowledge representation and reasoning is a common approach for tackling such problems [6]. Nevertheless, most benchmarks present today¹ are designed in a way that prevents the development of solvers that continuously use a changing amount of knowledge while solving different problem instances over time. Our contribution is to enable multi-agent systems to continuously reason about dynamic environments by utilising Answer Set Programming (ASP) – a non-monotonic knowledge representation and reasoning formalism [7], suitable for multi-shot solving [8]. In our case, the multi-agent system is controlled by the ALICA – A Language for Interactive Cooperative Agents [9]. Our preliminary work [10] forms the basis for integrating ASP with ALICA and allows us to extend it with a general solver interface which in turn makes ALICA open for a wider set of application domains. Moreover, we had to extend the utilised ASP solver with query mechanism according to [11]. In contrast to our preliminary work [12], the Module Property is now satisfied automatically. In order to evaluate our query mechanism, we used an ASP-based path planning scenario that utilizes a simplified version of the Region Connection Calculus [13], denoted as RCC-4.

¹ <http://www.satcompetition.org/> [Online; accessed September 12, 2017].
<http://aspcomp2015.dibris.unige.it/> [Online; accessed September 12, 2017].

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 Sect. 3. Furthermore, the query semantics extension including the automatic satisfaction of the Module Property is elaborated in Sect. 4. Section 5 provides the description of our evaluation scenario, whose results are presented in Sect. 6. Finally, we compare our work with other approaches in Sect. 7 and conclude with Sect. 8.

2 Foundations

This section is divided into three subsections. In Sect. 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 Sect. 2.2 that is about the syntax and semantics of ASP. In Sect. 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 Skubch [9] and two supplementary publications [14,15]. 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 itself 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 sequencers [16].

An ALICA program is a special tree, whose interior nodes are plans and its leaf nodes are atomic behaviours. The *CleanUp* plan in Fig. 2 is an example of such an interior node. A plan can include several states ($Z_0 \dots Z_8$) 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 the 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 Fig. 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.

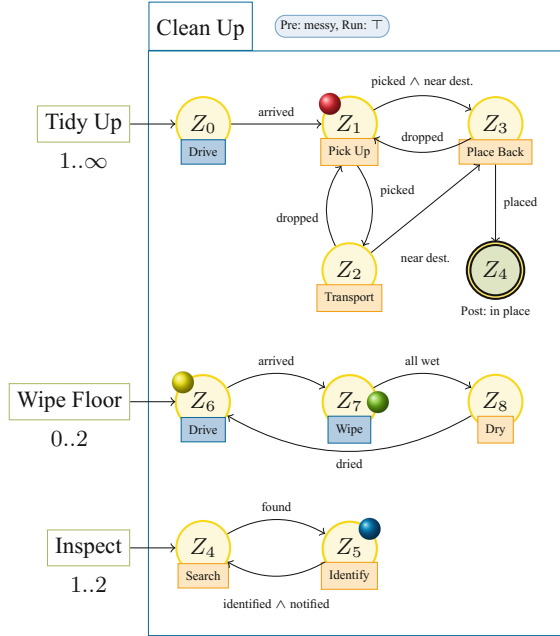


Fig. 2. Simple clean up plan [12]. (Color figure online)

The coloured circles on top of some states in Fig. 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 published 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 Fig. 2 to create a plan on its own.

The purpose of this plan, as shown in Fig. 3, could be the identification of coffee cups that should be cleaned, because they are dirty and not used any more. 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

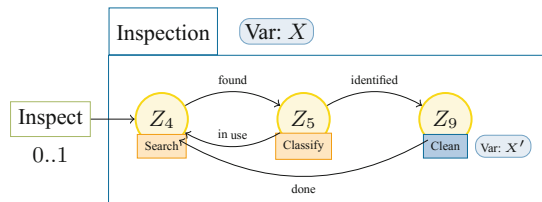


Fig. 3. Inspection plan [12].

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 determining the agent’s behaviour in the *Clean* plan, depending on which cup was found in the *Inspection* plan.

ALICA, as presented in [9], only provided one solver for assigning values to plan variables. The given solver addresses non-linear continuous constraint satisfaction problems and, as it was hard-wired to ALICA, the applicability of ALICA was limited for some domains. Our extension of ALICA with a general solver interface tackles this issue (see Sect. 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 expressiveness, ease of use, and computational effectiveness [17]. An ASP program is specified by a set of rules of the form $a_0 \leftarrow a_1 \wedge \dots \wedge a_m \wedge \sim a_{m+1} \wedge \dots \wedge \sim a_n$. Each a_i denotes a predicate $p(t_1, \dots, t_k)$ with terms t_1, \dots, t_k build from constants, variables, and functions. Rules consist of three parts, namely the head a_0 , the positive part $a_1 \wedge \dots \wedge a_m$ and the negative part $\sim a_{m+1} \wedge \dots \wedge \sim a_n$ 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.

The example in Listing 1.1 is a syntactically correct ASP program. In order to create syntactically correct 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 \neg in front of `robot(X)` stands for classic negation \neg and `not` in front 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 a capital 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 1.1. Robots can drive, as long as they are not broken.

The Herbrand universe of the program in Listing 1.1 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 [7,18] 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 1.1 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 rule's head ($a_0 \in M$), or does not include all predicates of the positive part of the rule's body ($\{a_1, \dots, a_m\} \not\subseteq M$), or contains predicates from the negative part of the rule's body ($\{a_{m+1}, \dots, a_n\} \cap M \neq \emptyset$). Informally speaking, a stable model is as small as possible and contains predicates, only if they are justified by facts. For a detailed introduction to the stable model semantics, see [19].

In our approach, we choose the Clingo 4.5.3 ASP solver [7], which introduces the notion of *External Statements* to ASP [8,20]. *External Statements* in combination with *Program Sections*, explained later on, are the key concepts for enabling the query semantics described in Sect. 4. The *External Statements* are predicates annotated with `#external` (see Listing 1.2). Those predicates are not removed from the body of a rule during grounding, even if they do not appear in the head of any rule, because of this annotation. Furthermore, it is possible to set them explicitly to `true` or `false` without an extra grounding step.

```

1  #external closed(lab, hall).
2  connected(lab, hall) :- not closed(lab, hall).
3  disconnected(lab, hall) :- closed(lab, hall).

```

Listing 1.2. Modeling a door using an External Statement.

The example, given in Listing 1.2, is part of our evaluation scenario presented in Sect. 5 and will also be used in ASPEXtensionQueries presented in Sect. 4.1. In this example `closed(lab, hall)` is marked as an *External Statement* and is therefore set to `false` by default. If the *External Statement* `closed(lab, hall)` is set to `true`, the head of Rule 3 holds and the predicate `disconnected(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*. 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 `disconnected(lab, hall)` no longer holds but `connected(lab, hall)` can be derived, thus the rooms are connected. Furthermore, changing the *External Statement* does not change the size of the resulting stable model.

Additionally, Clingo introduces *Program Sections* [7]. *Program Sections* are used to divide a logic program into different parts, which can be grounded separately. An example is given in Listing 1.3.

```

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

Listing 1.3. 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 *Program Section* `rcc4_composition_table` is grounded before the *Program Section* `rcc4_facts`, the resulting model would only contain the facts `properPart(office1,offices)` and `disconnected(studentArea, offices)`. If *Program Section* `rcc4_facts` is grounded first, the model will contain these facts before the *Program Section* `rcc4_composition_table` is grounded. Once this *Program Section* is grounded the model will additionally contain the fact `disconnected(office1, studentArea)`, since this rule's body 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 [13]. 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 [21] since the majority of the RCC-8 relations were not used for modelling the Distributed Systems Department (see Sect. 6). These calculi are commonly used in qualitative spatial reasoning and will be used to model our evaluation scenarios in Sect. 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

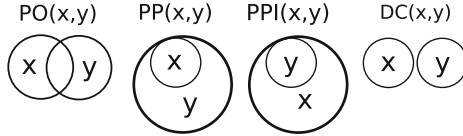


Fig. 4. RCC-4 base relations [12].

reflexive and symmetric. By using the $C(x,y)$ relation four base relations are defined and shown in Fig. 4.

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 [12].

	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 for using RCC-4 to model a building is given in Sect. 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.

We created an abstract solver interface that allows solving very different problem classes. A solver that is compatible with this interface, has to adhere to four corresponding concepts: *solver*, *variable*, *term*, and *problem descriptor*. The *solver* is expected to solve the problem contained in a *problem descriptor*. A *problem descriptor* encapsulates the other two concepts that describe the actual problem. Solving a problem means to assign values to *variables* in a way that the values fulfil constraints described by a set of *terms*. The ALICA engine only understands these basic relations between the four concepts. The type of values that can be assigned to a specific *variable* and the format of the *terms* is unknown to the ALICA engine. Only the domain specific parts of an ALICA program should be able to understand the actual meaning of a described problem and be able to interpret the solver’s results. Furthermore, the *variable* concept is identical to that of ALICA plan variables (see Sect. 2.1). Therefore, the ALICA engine is able to collect all relevant constraints (*terms*) over *variables* in the plan hierarchy, when a solution for a specific *variable* is required and can pass it via a *problem descriptor* to the responsible *solver*.

There are two methods for interacting with the solver interface. The first method checks if a solution for a given set of *variables* can be found. This method only determines, whether a solution exists and does not provide the solution itself. Usually, the complexity of providing a solution is the same as for proofing only the existence of a solution, but in case of an optimisation problem, it is not the same. Determining, for example, whether there is a cup available, is much easier than to determine the closest available cup. Therefore, whenever it is enough to check the existence of a solution the ALICA engine can save some time doing so.

With the second method, it is possible to actually determine the solution itself. The solver assigns its result to the corresponding variables and the ALICA engine hands them, via the *problem descriptor*, back to the calling domain-specific part of its program. The gradient solver, for example, returns continuous values and an ASP solver could return a set of predicates.

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 [22] 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

The extension of Clingo with query semantics is twofold. On the one hand, there is the structure of the query itself and on the other hand there is the processing of such a query. The query structure holds all information necessary for processing the query, e.g., ASP rules that change the ASP program during the query process. The details are explained in Subsect. 4.1. The workflow during the processing of

a query is described in Subsect. 4.2. This subsection places the focus especially on the requirements and techniques for changing the ASP program only during the query process and how it is possible to reestablish the programs state without the changes inflicted due to the query.

4.1 Query Structure

The central part of the query structure, further denoted as ASPQuery, is the ASPTerm. In compliance with the *Term* and *Variable* concepts, described in Sect. 4, the ASPTerm constraints a set of ASPVariables with a set of user defined rules. The interpretation of these rules depends on the type of the ASPQuery. It is either an ASPFilterQuery or an ASPEExtensionQuery (see Fig. 5).

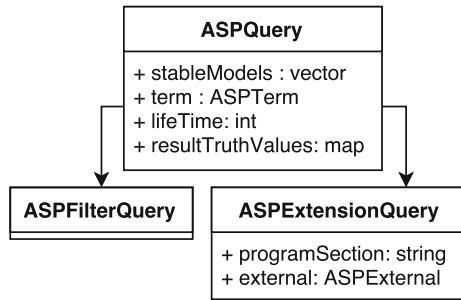


Fig. 5. Classmodel of ASPQueries.

The ASPQuery does, apart from the ASPTerm, also include a lifetime, a list of stable models, and a map of results with its corresponding truth values. The lifetime determines the number of solving operations the query should be processed with the current ASP program. The list of stable models includes all stable models from the last solving operation. Finally, the map of results and truth values contains one entry per queried predicate. The queried predicate is mapped to **true** if it is part of all stable models, **false** if its classic negation is part of all stable models, and **unknown** in all other cases.

The ASPFilterQuery is a simple ASPQuery that only filters the stable models by a set of given predicates. In the opposite to the ASPEExtensionQuery, the ASP program is not altered due to ASPFilterQueries. Gelfond et al. [11] present a similar query mechanism, that checks whether a grounded predicate is part of the solver’s stable models. In extension to this query formalism, we allow wildcards as parts of the filters. The filter `robot(wildcard)`, e.g., filters for all grounded `robot/1` predicates, including classic negated predicates. The truth value of this filter is set to **true** if at least one grounded `robot/1` predicate exists in all stable models, **unknown** if no grounded `robot/1` predicate appears in any stable model, and **false** otherwise.

The `ASPExtensionQuery` is a much more sophisticated way to query an ASP program. During its lifetime, it changes the ASP program and therefore its stable models. Afterwards, the ASP program is reverted to its original state. Consider, for example, an ASP program that identifies cups as free to use, when they are on the cupboard. With the `ASPExtensionQuery`, it is possible to temporarily declare all cups in the dishwasher as free to use, too. In general this is done by adding an arbitrary set of rules, given by the `ASPTerm`, to the ASP program. One rule in the set is specially handled and therefore further denoted as query rule. The head of the query rule defines, similar to the filter of the `ASPFILTERQuery`, the crucial predicates to look for in the stable models.

For each `ASPExtensionQuery` a unique *External Statement* and *Program Section* is created. Both are necessary to revert the ASP program back to its original state after the end of the query's lifetime. The *External Statement* is added to the body of each rule and all rules are added to the ASP program as part of the *Program Section*. The *Program Section* is grounded one time before the solving operations and as soon as the lifetime of the query is expired, the *External Statement* is released. Releasing the *External Statement* automatically removes all added rules from the ASP program and therefore their influence on the stable models.

Consider the following example, given in Listing 1.4. It demonstrates the setup of an `ASPExtensionQuery` in pseudo code.

```

1 term->addQueryRule("goalReachable(X):-reachable(X,Y),
                    goal(X),start(Y).")
2 term->addRule("goal(r1405B).")
3 term->addRule("start(r1411).")
4 query->setTerm(term)
5 query->programmSection("distributedSystems")
6 query->external(distributedSystemsExt)
7 query->lifeTime(1)

```

Listing 1.4. Example of an `ASPTerm`.

The example is part of an ASP navigation, which will be explained in detail in Sect. 5. Here it is queried whether room `r1405B` can be reached from room `r1411`. Therefore, the query rule and two facts are added to the *Program Section* `distributedSystems` and guarded by the *External Statement* `distributedSystemsExt`. Finally, the query's lifetime is set to one, which means that the query will be removed from the ASP program directly after it has been answered one time.

Since the `ASPExtensionQuery` is used to alter an ASP program, it can violate the Module Property, whose satisfaction guarantees that two *Program Sections* can be combined without rendering the ASP program unsolvable. A Module \mathcal{P} is defined as a triple of sets (P, I, O) . P is a ground program over the universe $\text{ground}(A)$ and both, I and O , are disjoint subsets of $\text{ground}(A)$. Furthermore, all *atoms* appearing in P are either part of I or O and all rule heads are part of O . I and O are denoted as input and output, respectively. Given this definition of a Module, two Modules \mathcal{P} and \mathcal{Q} are compositional, meaning their join will

not violate the Module Property if the following two conditions hold. 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 [23]. 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 \mathcal{P} and \mathcal{Q} (SCC) have to be considered. If any strongly connected component in SCC has a non-empty intersection with both output sets ($O(\mathcal{P}) \cap \text{SCC} \neq \emptyset$ or $O(\mathcal{Q}) \cap \text{SCC} \neq \emptyset$), this condition is violated. Violating any of these two conditions would result in a recursion between both Modules and therefore violating the Module Property. The given definition is based on the lecture material from Schaub [24].

Expanding an ASP program with a *Program Section* can violate of the Module Property, which causes that the ASP program can no longer be grounded and solved. Therefore, the *ASPExtensionQuery* has to guarantee unique rule heads for every rule in the query. This can be done by enclosing the rule heads inside a unique predicate, which in this case is realised by a constant string and a counter provided by the *ASPSolverWrapper* (see Sect. 4.2). This way of satisfying the Module Property does not change the arity of the query rule head but replaces it with a new predicate, which has to be considered when the result is returned to the user. In order to cope with this problem and making the Module Property transparent for the user, the *ASPExtensionQuery* is expanded by an automatic satisfaction of the Module Property. The pseudo code in Algorithm 1 describes our approach to guarantee the satisfaction of the Module Property.

This algorithm uses an *ASPExtensionQuery* q and the value of the counter c , which is maintained by the *ASPSolverWrapper*, as inputs and returns a modified unique *ASPExtensionQuery*. 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’s facts appearing in the query rule’s bodies are encapsulated in a new predicate $pred$, which is a combination of ps and c . After this step is completed, the query rule is expanded by the unique *External Statement*, which allows the removal of the query after it has been answered. Step 4 duplicates the query rule q' , which will be explained based on an example in the next paragraph. Furthermore, in Step 5 (Lines 8–10) the occurrences of all rule heads are encapsulated in the predicate $pred$, as well, which marks the end of altering the query rule. The additional rules and facts have to be adapted since they can still violate the Module Property. Therefore, Step 6 (Lines 11–14) alters the rule heads by encapsulating them in $pred$ and by adding ex to the body of every rule. This creates unique rules, which can later be removed from the ASP program by releasing the *External Statement*. The same process is used in Step 7 for altering the query’s facts, which results in a unique query.

An example for the application of this algorithm is presented in Listing 1.5. This example is part of the evaluation scenario presented in Sect. 5. In this example, the counter c , managed by the *ASPSolverWrapper*, is set to 1.

Algorithm 1: Automatic satisfaction of the Module Property.

Input : ASPEExtensionQuery q , Counter c
Output: ASPEExtensionQuery with unique rules

- 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 $q' = \text{duplicate } q.queryRule$
- 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 | replace $rule.head$ by $ps(rule.head)$
- 13 | $add(rule.body, ex)$
- 14 **end**
- 15 **foreach** $fact$ in $q.facts$ **do**
- 16 | replace $fact$ in $q.queryRule$ by $ps(fact)$
- 17 | $expand(fact, ex)$
- 18 **end**
- 19 **return** q

```

1 //ASPEExtensionQuery before applying the algorithm
2 goalReachable(X) :- reachable(X, Y), goal(X), start(Y),
   room(X), room(Y).
3 reachable(r1405, r1406) :- room(r1405), room(r1406).
4 goal(r1405B).
5 start(r1411).
6
7 //ASPEExtensionQuery after applying the algorithm
8 #program query1.
9 #external extQuery1.
10 query1(goalReachable(X)) :- reachable(X, Y), query1(goal(X)),
   query1(start(Y)), room(X), room(Y), extQuery1.
11 query1(goalReachable(X)) :- query1(reachable(X, Y)),
   query1(goal(X)), query1(start(Y)), room(X), room(Y), extQuery1
   .
12 query1(reachable(r1405, r1406)) :- room(r1405), room(r1406),
   extQuery1.
13 query1(goal(r1405B)) :- extQuery1.
14 query1(start(r1411)) :- extQuery1.

```

Listing 1.5. Application of the automatised satisfaction of the Module Property.

The example consists of the query rule in Line 1, an additional rule stating that if room 1405 and room 1406 exist they are reachable by each other, and two facts defining the start and goal position of the navigation. These facts can

cause a violation of the Module Property since they form two Strongly Connected Components between two Modules if this query is used twice. Therefore, the automatized satisfaction of the Module property is used. The result of this process is shown in Lines 8 till 14 of Listing 1.5. Hereby, the Lines 8 and 9 have already been added in the previous version of the *ASPExtensionQuery* presented in [12], which is a unique *Program Section* and *External Statement*. This part of the query is followed by the query rule and its duplicate in Line 10 and 11. 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 13 and 14 of this example. The only difference between the duplicates is the handling of the rule heads of additional rules like Line 12. In order to satisfy the Module Property, the head of this rule has to be expanded as well. This change has to be conducted in the query rule, 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 `reachable(r1405, r1406):- room(r1405), room(r1406)`. Therefore, the occurrence of the rule head in the query rule has to be replaced. This causes that the `reachable` predicates in the knowledge base are no longer appearing in the query rule. Thus, leaving out possible solutions for the query. To cope with this problem, the duplicated query rule is used. The appearance of the additional rule heads is not replaced, leaving the possibility to use the knowledge base. After solving the query the unique predicate is removed from the results.

To sum up, the presented approach of an automatized satisfaction of the Module Property allows the user to create *ASPExtensionQueries*, without the opportunity to violate it. Furthermore, this approach allows to expand predicates appearing in the knowledge base without losing the possibility to use the knowledge base. 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.

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 the ASPQueries have to be forwarded to the Clingo ASP solver. Therefore, a wrapper has been created, that encapsulates the Clingo ASP solver and provides access to the query structure. This wrapper, named *ASPSolverWrapper*, is registered at the ALICA framework by using the interface explained in Sect. 3 and can be used by ALICA behaviours. The interaction workflow between the ALICA Engine and the wrapper are depicted in Fig. 6.

As mentioned in Sect. 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 (an example is given in Listing 1.4), it can be used by an ALICA behaviour to formulate a query. Thereby, the *External Statements'* truth values are given by a `worldmodel` class,

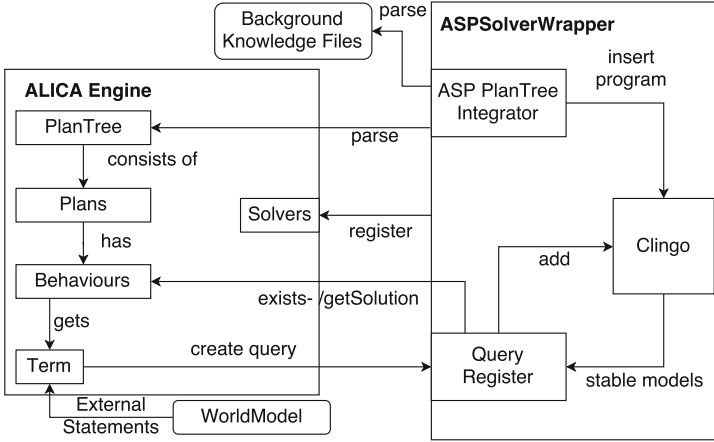


Fig. 6. Query workflow [12].

which encapsulates data that was perceived by the local 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, which passes it to the ASP solver Clingo. In case of an ASPExtensionQuery, the automatized satisfaction of the Module Property (Sect. 4) is applied, which encapsulates the query in a unique *Program Section*. Afterwards, the wrapper adds the query rule, the set of additional 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 the background knowledge files. These files, for example, contain rules to detect malformed ALICA programs. The corresponding ASP rules are presented in [10]. The ASPPlanTreeIntegrator is only used during 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 anymore. Once all program parts (queries, background knowledge, and plan structure) have been added, Clingo grounds and solves the program. The in this process 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 be used in combination with an `ASPFilterQuery`, 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 `ASPExtensionQuery`, where a query rule, a set

of additional rules, and facts are used to alter the ASP program. Since the `ASPExtensionQuery` modifies the ASP program the resulting stable models and the queried rule heads are returned to the querying ALICA behaviour. Since the rule heads have been altered by Algorithm 1, the encapsulating unique *predicate* is removed before the results are returned to the caller. After the results are returned, the ALICA behaviour can react to the changes in the model or to the resulting rule heads. By returning the calculated results to the ALICA behaviour, the workflow of a query is finished and the queries lifetime is reduced by one and the ALICA behaviour can create the next query.

5 Evaluation Scenario

Our approach for handling dynamic domains will be evaluated using the scenario presented in this section. The base of this scenario is a map of the Distributed Systems Department of the University of Kassel and was created by using a TurtleBot [25]. A TurtleBot is a small service robot equipped with a 3D camera and was in our case extended with a 2D laser scanner. The customised version of a TurtleBot is depicted in Fig. 7 alongside the resulting map, which is shown in Fig. 8.

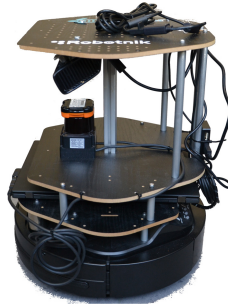


Fig. 7. Adapted version of a TurtleBot.

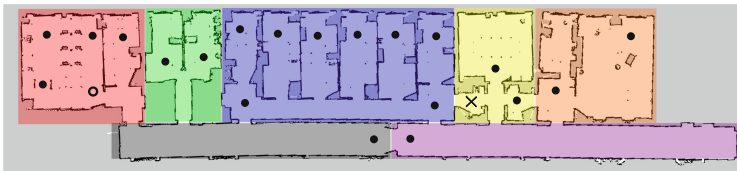


Fig. 8. Map of the distributed systems department [12]. (Color figure online)

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. For example, a POI was placed on different workplaces, the coffee machine or the conference room. Furthermore, the robot’s position is marked with a circle and the navigation goal used in this scenario is marked by a cross. The relations between the areas, rooms and POIs are modelled using the Region Connection Calculus 4 (Sect. 2). 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**. Additionally, doors have been modelled utilising *External Statements*, as shown in Listing 1.2. This enables our approach to use different parts of the logic program without an additional grounding step and thus allows to change the stable models. 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, which could lower the performance of the ASP program. In order to use this scenario, we implemented an ALICA behaviour, which uses an ASPFilterQuery to check if the robot’s goal position (cross) is reachable from its current position. In this case, we use the transitive closure defined by the predicate `reachable(X,Y)`, which is presented in Listing 1.6 as a simple path planning approach.

```

1 reachable(X,Y) :- reachable(Y,X),X!=Y.
2 reachable(X,Z) :- reachable(X,Y),reachable(Y,Z),
                    X!=Y,Y!=Z,X!=Z.
3 reachable(X,Y) :- partialOverlapping(X,Y),room(X),room(Y).
4 reachable(X,Y) :- partialOverlapping(X,Y),area(X),area(Y).
5 reachable(X,Y) :- properPart(X,Y),room(X),area(Y).

```

Listing 1.6. Transitive closure of reachable relation.

Rules 1 and 2 of the listing express, the `reachable` relation is symmetric and transitive. Furthermore, Rules 3 and 4 state, that a room `room(X)` or area `area(X)` is `reachable` from another room `room(Y)` or area `area(Y)` if they are partial overlapping `partialOverlapping(X,Y)`. As a last point, a room is `reachable` from an area if the room is a `properPart` of the area (Rule 5).

6 Evaluation

In this section, we present the revised and updated evaluation results from [12]. There are three main differences to our preliminary evaluation. We utilise a more current version of the ASP solver Clingo, the Module Property is satisfied automatically, and we evaluated the influence of adding *External Statements* to our ASP program.

6.1 Dynamic Changes

The evaluation scenario has been modelled in two different ways. The first way, denoted as **Ext**, makes use of *External Statements* in its ASP rules, as described in Sect. 5. The second, denoted as **noExt**, purely relies on facts describing the connections between rooms of the department. Both ways utilize the transitive closure for their path planning approach, as presented in Listing 1.6. In Fig. 8 the robot’s starting position is marked by a circle and the goal is marked by a cross. The path the robot is supposed to follow 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 a door and finally reaches the goal, which is situated in the **utility area**. This door is the solely open door in **mainHallB** and is opened and closed via an *External Statement* to simulate a change in the environment.

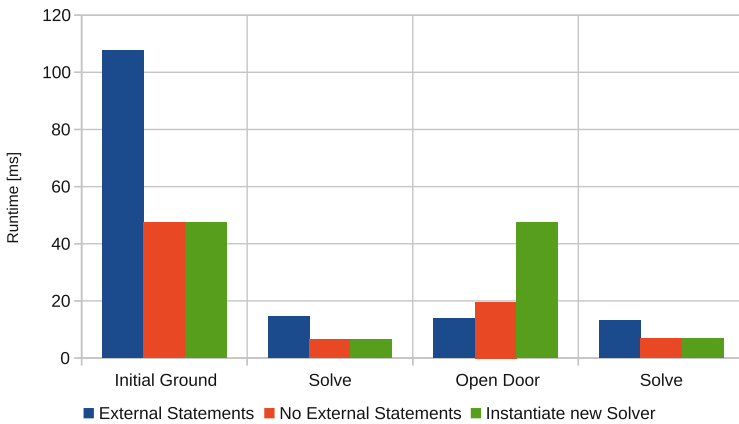


Fig. 9. Comparison of different modelling approaches [12].

The evaluation mainly consists of four steps. As a first step the solver is initialized and after transforming the ALICA plantree into corresponding ASP rules, the correctness of the plantree is checked as well as a navigation query is solved. In this step, the selected goal is not reachable, due to closed doors. The second step is purely solving the composed ASP program again, in order to be able to compare the time needed. In step three a door is opened, making the navigation between both rooms possible. This either means a new grounding step (**noExt**), the change of the truth value of an *External Statement* (**Ext**), or the creation of a new solver instance, which uses the **noExt** model. The last step is purely solving the ASP program again.

The evaluation results of our preliminary evaluation are presented in Fig. 9. In contrast to these former results, all steps were roughly halved in runtime, independent from the way changes are handled (see Fig. 10).

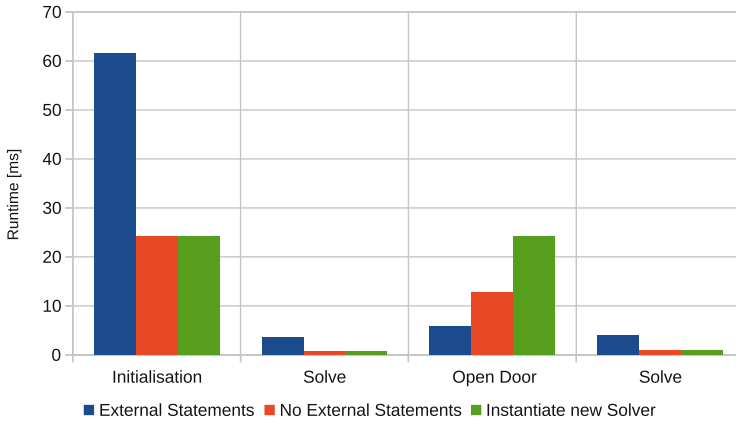


Fig. 10. Comparison of different modelling approaches using Automatic Module Property satisfaction.

The first and most time-consuming step lasts 61.5 ms for *Ext* and 24.2 ms for *noExt* or creating a new solver instance. The difference of 37.3 ms is caused by the way the department is modelled. In *noExt*, connections between rooms are expressed by single facts. In *Ext*, a connection between two rooms consists of an *External Statement* and two rules, which express the state of a door. This increases the size of the ASP program, and therefore, the initialization time. The second and the fourth step have similar results. Hereby, the usage of *Ext* results in an average runtime of 3.9 ms, while the usage of *noExt* results in an average runtime of 0.8 ms. In step three a door is opened or closed. This represents a dynamic change in the environment and is, therefore, more critical for our investigation than the other steps. The usage of *Ext* results in a runtime of 5.8 ms, *noExt* lasts 12.8 ms, and the instantiation of a new solver takes 24.2 ms. This is caused by the way a change in the model is performed. In *Ext*, the truth value of an *External Statement* is changed and thereby slightly influences the runtime. In *noExt*, a *Program Section* has to be grounded, which increases the runtime by roughly 12 ms. The highest increase of 23.6 ms is caused by a new solver instance since the initialization step has to be performed again.

As the ALICA framework usually runs at 30 Hz, an ALICA behaviour can query the ASP solver at most 30 times per second, i.e., each iteration. In Fig. 11, all three methods for modelling the department are compared with respect to the number of changes per 30 iterations.

The x-axis shows the number of changes per 30 iterations and the y-axis shows the average runtime of a query. The runtime for 30 changes per 30 queries, i.e., opening or closing a door each iteration, corresponds to the runtime of step three in Fig. 10. 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.

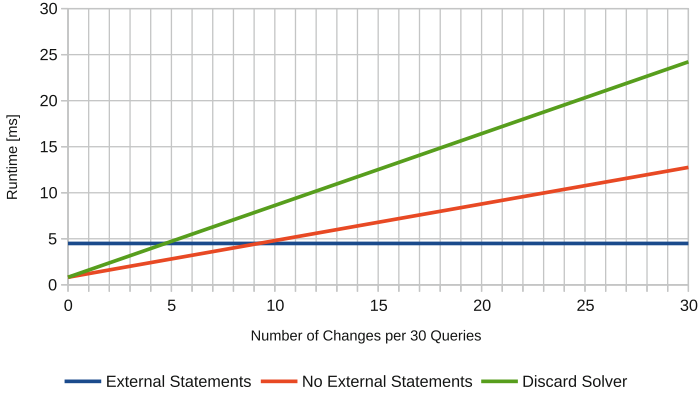


Fig. 11. Comparison of measured time regarding changes in the model.

As a result, *External Statements* are superior in runtime, when in 10 out of 30 iterations a door is closed or opened. This is due to the fact that, when using `noExt`, the stable models' sizes increase with every change made and slows down the process of handling the models rapidly. In contrast, when using `Ext`, the model's size stays the same. A third alternative is to discard the current and create 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` in dynamically changing environments.

6.2 Scalability of *External Statements*

Our approach to automatically satisfy the Module Property makes use of *External Statements*. Each time a query is formulated that was never queried before, a new *External Statement* is added to the ASP program. Although it is possible to deactivate old queries by releasing their corresponding *External Statements*, we encountered that old `ASPExtensionQueries` have a significant influence on the runtime. `ASPFilterQueries` do not suffer from this issue, as they do not alter the ASP program.

In Fig. 12 the runtime for an increasing number of *External Statements* is shown. It is important to note that there is always only one *External Statement* activated and all other *External Statements*, originating from former queries, are released. The average increment per additional *External Statement* is 0.1 ms. Therefore, the use of at most 1800 *External Statements* are allowed, without dropping below a query frequency of 5 Hz.

6.3 Scalability of the Region Connection Calculus

Following our preliminary evaluation, we re-evaluated the scalability of the Region Connection Calculus 4 using a more current version of Clingo. Utilising

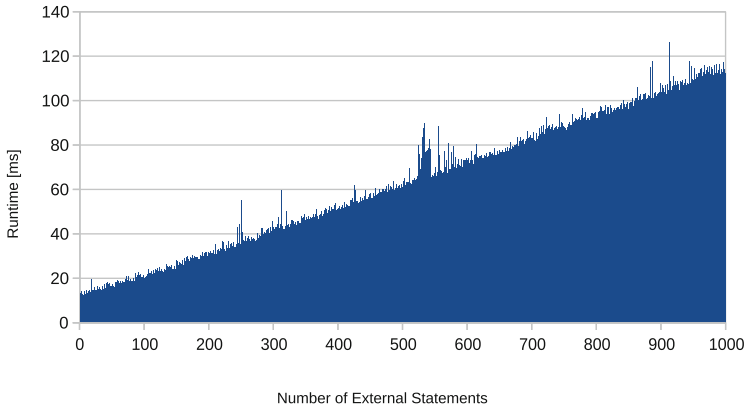


Fig. 12. Influence of increasing External Statements on the query runtime.

the navigation scenario from Sect. 4, we expanded the number of rooms, starting from 500 rooms and increasing up to 2000 rooms. Furthermore, we tested different connection densities between the rooms ranging from 25% to 100% connected rooms. Hereby, 100% means that every room has at least one connection to another room. The results are given in Figs. 13 and 14.

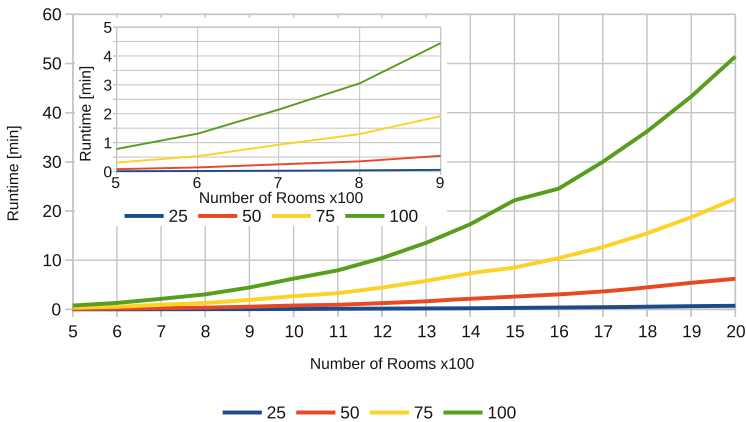


Fig. 13. Runtime of the initial grounding.

We stopped the evaluation at a number of 2000 rooms, as both runtimes increase exponentially with rising connection density and number of rooms. At 2000 rooms, the grounding already lasted 51 min. As a result, we only suggest using the Region Connection Calculus 4 in combination with transitive closure based path planning in dynamic domains if only less than 900 regions need to be considered. We consider this number suitable for dynamic domains since, after

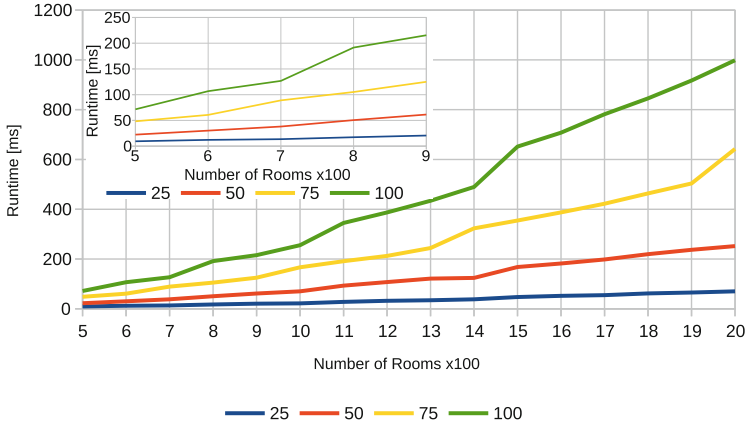


Fig. 14. Runtime of the solve step.

an initialization phase of roughly 4.5 min, a robot can still change the knowledge base 5 times per second by an `ASPExtensionQuery`.

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 include `DyKnow` [26] and `KnowRob` [27], which are both presented in this section. Furthermore, papers utilising ASP for multi-shot solving and the application of ASP in household scenarios are shown.

`DyKnow`, which is presented in [26], sets its focus on distributed collection and the distribution of data. This includes raw sensor values, processed sensor values or even predicates, which hold between objects recognised. Both, the collection and the distribution, are a set of processes specified in the knowledge processing language provided in [26]. These processes provide collected data, derived information, and knowledge about objects and their relation to multiple agents and allows them to reason about the received data. In comparison to this, the ALICA framework provides a domain independent framework, which is used to model and control the behaviour of agents. By expanding ALICA with the solver interface and the query mechanism presented, the agents are able to reason about the relations of objects in their environment and the domain specific knowledge, which is given in ALICA behaviours. Furthermore, ALICA provides the functionality to hierarchically constraint variables, which allows the formulation of queries utilising the ALICA plan tree structure.

`KnowRob`, presented in [27], is a framework utilising Prolog to build a knowledge base that provides access methods to retrieve information stored in the knowledge base. This knowledge base is extended by a “virtual knowledge base”, which is used to compute the abstract representation of data when the data is

queried. This is done by forwarding the query to another part of the robot, which has free resources or can provide better results. Furthermore, this framework supports only one agent, which is a contrast to ALICA which is used for teams of agents. In comparison to KnowRob framework, ALICA uses a declarative programming approach (ASP and Prolog), too. Additionally, both frameworks use this programming approach to formulate knowledge bases, which can be accessed by the supported agents.

Furthermore, in [8] 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.

Erdem et al. present in [28] a framework utilising ASP and ConceptNet [29] for representing commonsense knowledge in ASP, which is then 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. Therefore, the possible actions of a robot and the estimated locations of objects are modelled in ASP. This can be compared to our evaluation scenario (Sect. 5) and the application of the developed query mechanism. A household as presented in [28] is a highly dynamic and human populated environment and therefore is suited for the presented query mechanism, which has been proven in Sect. 6.

Finally, we want to state the difference to our preliminary work [12]. In [12] the user is required to manually guarantee the satisfaction of the Module Property. In this extended version, we introduce an approach to automatically guarantee the satisfaction of the Module Property. Whenever the user formulates an *ASPExtensionQuery* (in [12] denoted as *ASPVariableQuery*) our approach transforms the given query rules into unique rules without any interaction from a user. This makes the otherwise tedious satisfaction of the Module Property transparent to the user and releases him from this responsibility. Additionally, we re-evaluated our approach with a more current version (5.2.0) of the ASP solver Clingo that gives us a significant runtime improvement, as shown in Sect. 6.

8 Conclusion and Future Work

In extension to our preliminary work [12], we presented an automatic satisfaction of the Module Property and re-evaluated our scenarios with a current version of the ASP solver Clingo. The integration of Clingo with the ALICA framework clearly benefits from the current version of Clingo. All runtime evaluations show improved results. In order to prevent a violation of the Module Property, we automatically create a unique rule head for every part of the *ASPExtensionQuery*. As we showed before, both ways of creating unique rule heads (with or without *External Statements*) respond appropriately fast (less than 20 ms) for dynamic

domains. Nevertheless, the runtime advantage with *External Statements* further improved, compared to the results in [12], due to the current Clingo version. We observed one disadvantage of *External Statements*. Whenever it is necessary to create a new query that is different from all other queries before, a new *External Statement* needs to be added to your ASP program. Although it is absolutely possible to do this automatically, we encountered an increase in solving time by 1 ms per created query. The amount of 1 ms depends on our modelling and can probably be further reduced, but the number of *External Statements*, even when they were already released, is a limiting factor with regard to scalability. However, we still propose the use of *External Statements* for modelling dynamic domains, such as human-populated service robotic domains. Only the number of different queries is limited to 1800 in order to remain reasonably low in runtime. As in [12], we evaluated the scalability of the Region Connection Calculus 4 by determining the transitive closure of the reachability relation. With the current Version of Clingo, we may conclude that the calculus scales up to a number of 900 instead of only 600 regions under premises that an agent can still query its knowledge base at a rate of 5 Hz after an initial grounding time of roughly five minutes.

In our future work, we will increase the variety of our scenarios in order to get a more profound impression of the validity of *External Statements* as a solution to reasoning in dynamic domains. Furthermore, this investigation will be joined with knowledge-based collaboration between multiple agents. In the current scenarios, the knowledge bases are independent of each other and the agents do not utilise knowledge from other agents knowledge bases. Implementing this feature, e.g., would allow an agent to ask another agent for open doors, instead of searching for open doors by itself.

Another aspect of knowledge-based collaboration is about global consistent stable models. Furthermore, it is possible in ASP 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.

Besides the knowledge-based collaboration of agents, another part of our future work will be the provision of ASP based commonsense knowledge, which will enable agents to solve everyday tasks. A promising approach for this is the combination of Clingo with ConceptNet 5 [29], which represents commonsense knowledge as a hypergraph. This hypergraph consists of weighted edges connecting concepts with a given set of relations. These edges will be translated into ASP and will provide a commonsense knowledge base that can be accessed by an agent with the presented query structure.

References

1. Wurman, P.R., D'Andrea, R., Mountz, M.: Coordinating hundreds of cooperative, autonomous vehicles in warehouses. *AI Mag.* **29**, 9–19 (2008)
2. Hars, A.: Driverless Car Market Forecasts (2017). http://www.driverless-future.com/?page_id=384. 9 May 2017
3. Tomizawa, F.: Who is NAO? (2017). <https://www.ald.softbankrobotics.com/en/cool-robots/nao>. 9 May 2017
4. Sullivan, A., Elkin, M., Umaschi Bers, M.: KIBO robot demo: engaging young children in programming and engineering. In: *Proceedings of the 14th International Conference on Interaction Design and Children*, pp. 418–421. ACM (2015)
5. van Overbeeke, B.: Service Robot Amigo at the RoboCup Dutch Open 2012 (2011). <https://www.flickr.com/photos/robocup2013/9015099760>. 19 May 2017
6. Brachman, R.J., Levesque, H.J.: *Knowledge Representation and Reasoning*. Morgan Kaufmann Series in Artificial Intelligence. Morgan Kaufmann, Los Altos (2003)
7. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Clingo = ASP + Control: Extended Report (2014). https://www.cs.uni-potsdam.de/wv/publications/TEMP_journals/corr/GebserKKS14x.pdf. 10 Oct 2017
8. Gebser, M., Janhunen, T., Jost, H., Kaminski, R., Schaub, T.: ASP solving for expanding universes. In: Calimeri, F., Ianni, G., Truszczyński, M. (eds.) *Proceedings of the 13th International Conference on Logic Programming and Non-monotonic Reasoning*, pp. 354–367. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-319-23264-5_30
9. Skubch, H.: *Modelling and Controlling of Behaviour for Autonomous Mobile Robots*, 1st edn. Springer Vieweg, Heidelberg (2013). <https://doi.org/10.1007/978-3-658-00811-6>
10. Opfer, S., Niemczyk, S., Geihs, K.: Multi-agent plan verification with answer set programming. In: Aßmann, U., Brugali, D., Piechnick, C. (eds.) *Proceedings of the Third Workshop on Model-Driven Robot Software Engineering*. ACM (2016)
11. Gelfond, M., Kahl, Y.: *Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The Answer-Set Programming Approach*. Cambridge University Press, Cambridge (2014)
12. Opfer, S., Jakob, S., Geihs, K.: Reasoning for autonomous agents in dynamic domains. In: van den Herik, J., Rocha, A.P., Filipe, J. (eds.) *Proceedings of the 9th International Conference on Agents and Artificial Intelligence*, pp. 340–351. ScitePress Digital Library (2017)
13. Randell, D.A., Cui, Z., Cohn, A.G.: A spatial logic based on regions and connection. In: Nebel, B., Rich, C., Swartout, W. (eds.) *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning*, vol. 92, San Francisco, CA, USA, pp. 165–176. Morgan Kaufmann (1992)
14. Skubch, H., Saur, D., Geihs, K.: Resolving conflicts in highly reactive teams. In: Luttenberger, N., Peters, H. (eds.) *17th GI/ITG Conference on Communication in Distributed Systems (KiVS 2011)*. OpenAccess Series in Informatics (OASISs), vol. 17, pp. 170–175, Dagstuhl, Germany, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2011)
15. Skubch, H., Wagner, M., Reichle, R., Geihs, K.: A modelling language for cooperative plans in highly dynamic domains. In: van de Molengraft, M., Zweigle, O. (eds.) *Special Issue on Advances in Intelligent Robot Design for the Robocup Middle Size League*, vol. 21, pp. 423–433. Elsevier (2011)

16. Gat, E.: On three-layer architectures. In: Kortenkamp, D., Bonasso, R.P., Murphy, R. (eds.) *Artificial Intelligence and Mobile Robots: Case Studies of Successful Robot Systems*, pp. 195–210. MIT Press, Cambridge (1998)
17. Brewka, G., Eiter, T., Truszczyński, M.: Answer set programming at a glance. *Commun. ACM* **54**, 92–103 (2011)
18. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Logic* **7**, 499–562 (2006)
19. Eiter, T., Ianni, G., Krennwallner, T.: Answer set programming: a primer. In: Tessaris, S., Franconi, E., Eiter, T., Gutierrez, C., Handschuh, S., Rousset, M.-C., Schmidt, R.A. (eds.) *Reasoning Web 2009. LNCS*, vol. 5689, pp. 40–110. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03754-2_2
20. Gebser, M., Grote, T., Kaminski, R., Obermeier, P., Sabuncu, O., Schaub, T.: Answer set programming for stream reasoning. In: Eiter, T., McIlraith, S. (eds.) *Proceedings of the Thirteenth International Conference on Principles of Knowledge Representation and Reasoning (KR 2012)*, vol. 13, pp. 613–617. AAAI Press (2012)
21. Gatsoulis, Y., Alomari, M., Burbridge, C., Dondrup, C., Duckworth, P., Lightbody, P., Hanheide, M., Hawes, N., Hogg, D.C., Cohn, A.G.: QSRlib: a software library for online acquisition of qualitative spatial relations from video. In: Bredeweg, B., Kansou, K., Klenk, M. (eds.) *Proceedings of the 29th International Workshop on Qualitative Reasoning*, pp. 36–41 (2016)
22. Witsch, A.: *Decision Making for Teams of Mobile Robots*. Dissertation, University of Kassel, Kassel, Germany (2016)
23. Sharir, M.: A strong-connectivity algorithm and its applications in data flow analysis. *Comput. Math. Appl.* **7**, 67–72 (1981)
24. Schaub, T.: *Module Composition*, 13 October 2016. <https://www.cs.uni-potsdam.de/~torsten/ijcai15tutorial/asp.pdf>. 9 May 2017
25. Foote, T., Wise, M.: *TurtleBot Website* (2010). <http://www.turtlebot.com/>. 9 May 2017
26. Heintz, F., Kvarnström, J., Doherty, P.: Bridging the sense-reasoning gap: DyKnow - stream-based middleware for knowledge processing. *Adv. Eng. Inform.* **24**, 14–26 (2010)
27. Tenorth, M., Beetz, M.: KnowRob: a knowledge processing infrastructure for cognition-enabled robots. *Int. J. Robot. Res.* **32**, 566–590 (2013)
28. Erdem, E., Aker, E., Patoglu, V.: Answer set programming for collaborative house-keeping robotics: representation, reasoning, and execution. *J. Intell. Serv. Robots* **5**, 275–291 (2012)
29. Speer, R., Havasi, C.: ConceptNet 5: a large semantic network for relational knowledge. In: Gurevych, I., Kim, J. (eds.) *The People’s Web Meets NLP. Theory and Applications of Natural Language Processing*, pp. 161–176. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35085-6_6