Towards ASP-based Scheduling for Industrial Transport Vehicles *

Felicitas Fabricius
Marco De Bortoli
Gerald Steinbauer
Graz University of Technology
{mbortoli,steinbauer}@ist.tugraz.at

Maximilian Selmair
BMW Group
maximilian.selmair@bmw.de

Michael Reip
Incubed IT
m.reip@incubedit.com

Martin Gebser
Universität Klagenfurt
martin.gebser@aau.at

Abstract.

The increasing number of robots and autonomous vehicles involved in logistics applications leads to new challenges to face for the community of Artificial Intelligence. Web-shop giants, like Amazon or Alibaba for instance, brought this problem to a new level, with huge warehouses and a huge number of orders to deliver with strict deadlines. Coordinating and scheduling such high quantity of tasks over a fleet of autonomous robots is a really complex problem: neither simple imperative greedy algorithms, which compromises over the quality of the solution, nor precise enumeration techniques, which make compromises over the solving time, are anymore feasible to tackle such problems. In this work, we use Answer Set Programming to tackle real-world logistics problems, involving both dynamic task assignment and planning, at the BMW Group and Incubed IT. Different strategies are tried, and compared to the original imperative approach.

1. Introduction

Industry 4.0 is bringing more and more interest toward the digitalization of all productive stages in the industrial field. Even before that, we all have been witnesses of the big impact robotics had in industry, by the automatization of repetitive tasks. In the last years, thanks to the increasing computational power, Artificial Intelligence (AI) is spreading as well, leading to the next step of the integration between robots and production: the automatization of complex tasks requiring reasoning. In this perspective, optimization of logistics is crucial for large companies, in order to save both time and money. Still we are dealing with a production environment which considers a fleet of robots floating around, efficiently performing tasks and carrying goods where to model such NP-hard domains a high number of constraints is needed. For this reason, an imperative approach becomes more and more difficult to maintain, and cannot benefit from the numerous meta-heuristics and optimizations (if not manually implemented) already encoded inside the solvers of other programming paradigms, like declarative programming. Answer Set Programming (ASP) is a fast and intuitive logic language, which already has many applications both in industry and in research (see Section 2). In this paper we are going to investigate the difference between the two paradigms, by replacing the imperative part of the task schedulers used by two companies, the BMW Group and Incubed IT, with an ASP implementation. While classical languages are well suited for greedy algorithms implementation, declarative programming has other advantages: first of all, the focus is on the description of the problem, leaving all the solving details to the external solver. Moreover, most solvers are configurable with a lot of meta-heuristics to cut the search space: the user has only to find the one which fits the problem better, without implementing anything. Then, since logic languages are basically based on enumeration techniques, an ASP solver looks for the best solution, or at least the best one in a given time. Depending

*This paper appeared at the 2020 Joint Austrian Computer Vision and Robotics Workshop.

on the size of the instance, this behaviour leads to huge computational time with respect to a greedy algorithm. We are interested into the analysis of this trade-off between greedy solving time and declarative solution quality. This paper is based on Felicitas Fabricius' Master thesis [6].

2. Related Research and ASP Foundations

The demand for increased complexity and scalability in industry automatization requires more and more powerful techniques and algorithms. Imperative programming is suitable to write a very problem-specific solution. However, the development of such kind of code can be really arduous, time expensive and difficult to maintain. Optimal task scheduling and planning, enriched by domain-based heuristics, requires a huge amount of code lines if written with an imperative language [17].

Answer Set Programming, and logic programming in general, allows to tackle combinatorial problem in a very intuitive way, splitting the work into two phases: the description of the problem and its efficient solving procedure [7]. The programmer has only to care of the former, and this requires just a fragment of the effort required by an imperative language. Then, (s)he can use one of the solvers available in the market, like Clingo or DLV, to find the optimal solution, improving it with a large set of meta-heuristics. Although the most common approach is the imperative one, many use-cases of ASP applied to industry can be found in the literature: in 2017, Dodaro and Maratea designed a shift plan for 164 Italian nurses, calculating the optimal plan for an entire year in just 50 minutes, using the state-of-the-art solver Clingo [3]. Staying in the shift scheduling field, the DLV solver was deployed to find the optimal shift plan for seaport workers [14]. In this case, the problem was complicated by the fact that the employees have different qualifications, and there were different kinds of tasks. Finding of the optimal one month-long plan required 8 minutes. A similar work, considering different demands as well, is described in [2]. Moving to other kind of industrial fields, ASP was used in E-tourism in order to find the travel which suits the user the most [1]. In [12] the authors used ASP for phone routing in call centers. The customer was classified in a category and assigned directly to the human operator. We can find plenty of ASP applications regarding task assignment and routing as well. Examples can be found in [4], [16], [10], [13] and [15].

Answer Set Programming is based on the stable model semantics, presented by Gelfond and Lifschitz in [11] for dealing with logic programs with negation as failure. With the following we give a quick overview of the language semantics [2, 7].

A *rule* r in a logic program is an expression of the form

$$h \leftarrow a_1, \ldots, a_m, \neg a_{m+1}, \ldots, \neg a_n \qquad (1)$$

where $a_1, \ldots, a_n$ are *atoms* of the form $s(t_1, \ldots, t_k)$, in which s is a *predicate* symbol and $t_1, \ldots, t_k$ are *terms*, viz. constants, variables, or functions, and $\neg$ stands for *default negation*. The head h of r is either an atom a, a choice $\{a\}$, or the special symbol $\bot$. If h is an atom and $n = 0$, we call r a *fact*, a *choice rule* if h is $\{a\}$, and an *integrity constraint* if h is $\bot$; we skip $\leftarrow$ or $\bot$, respectively, when writing rules (1) with $n = 0$ and integrity constraints. A *logic program* P is a set of rules and constraints. In the first-order case, terms occurring in P may include arithmetic expressions, and atoms may be based on relational operators like "$<$". On the other hand, a term, atom, rule, constraint, or program is *ground* if it does not include variables, arithmetic expressions, or relational operators. A first-order program P stands for the set $grd(P)$ of all instances of rules and constraints constructible by substituting ground terms for variables and evaluating arithmetic expressions as well as relational operators in the standard way. For details on ground instantiation, we refer the interested reader to [5, 9]. The semantics of a logic program P is given by its stable models, which are particular sets of (true) ground atoms as defined in the following. The *reduct* P^X relative to a set X of ground atoms is the set of all rules and constraints in $grd(P)$ such that $\{a_1, \ldots, a_m\} \subseteq X$, $\{a_{m+1}, \ldots, a_n\} \cap X = \emptyset$, and $a \in X$ if $h = \{a\}$ is a choice for a rule (1). Then, X is a *stable model* of P if it is $\subseteq$-minimal among the sets of ground atoms such that, for all rules in P^X, $\{a_1, \ldots, a_m\} \subseteq X$ implies $h \in X$ or $a \in X$ if $h = \{a\}$. In addition to rules, a logic program can contain `#minimize` statements of the form

$$\texttt{\#minimize}[\ell_1 = w_1@L_1, \ldots, \ell_n = w_n@L_n].$$

Besides literals ℓ_i and integer weights w_i for $1 \leq i \leq n$, a `#minimize` statement includes integers L_i providing priority levels [8]. The `#minimize` statements in P distinguish optimal answer sets of P in

the following way. For any set X of atoms and integer L, let Σ_L^X denote the sum of weights w_i such that $\ell_i = w_i@L$ occurs in some `#minimize` statement in P and ℓ_i holds w.r.t. X. We also call Σ_L^X the utility of X at priority level L. An answer set X of P is dominated if there is an answer set Y of P such that $\Sigma_L^Y < \Sigma_L^X$ and $\Sigma_{L'}^Y = \Sigma_{L'}^X$ for all $L' > L$, and optimal otherwise.

3. ASP and Logistics: Two Cases-Studies

To evaluate ASP in an industrial environment, we discovered two interesting case-studies. Both are related to Fleet Management Systems (FMS) - one at Incubed IT, the other one at the BMW Group. In both cases, the imperatively described task allocation strategy was replaced by an ASP-based program.

3.1. The BMW Use Case: Task Assignment and Charging Management

By the following, the requirements for the FMS at the BMW Group are described. Here, two elemental decisions have to be made. These are on one hand the assignment of tasks to the vehicles and on the other hand the assignment of charging and parking stations to the same vehicles. Both decisions are made online, which means that neither tasks nor the needs for charging (and parking) are known beforehand. With *task* we mean a transportation job of a container, accomplished by a vehicle, from a station to another one. The required time is estimated from the Euclidean distance.

For the task assignment, the standard C# scheduler applies a trivial first-in-first-out (FIFO) strategy, which means that earlier created tasks have to be executed first. By that, the criterion for the selection of tasks, formulated as a constraint, is not to assign a task if there is another appropriate task with earlier creation time assignable. Vehicles on the field must have a battery level at a minimum of 25 %, and charging vehicles a battery level of 40 % to be assigned to tasks. The optimal assignment of vehicles to tasks is based on the traveling costs that are set to be the Euclidean distance between robots and the first goal of the assigned task. The used optimization criterion ensures the lowest traveling cost for the tasks with earliest time of creation.

In ASP a different optimization criterion is used, in order to achieve a better overall quality of the solution. The Euclidean distance for all assignments is summed up and minimized, in order to have a better make-span and save more energy. Considering T and R as the set of tasks and robots respectively, task assignment is encoded by the following logic formulas:

$$\forall t \in T(|\{r \in R|(assign(t, \_)\}| \leq 1)$$
$$\forall t_1, t_2 \in T, \forall r \in R$$
$$(assign(t_1, r) \wedge assign(t_2, r) \wedge t_1 \neq t_2 \Rightarrow \bot)$$

The first formula may (non-deterministically) assign each task to one robot at most. The second one makes sure that two different tasks are not assigned to the same robot. The non-deterministic choice is driven by the optimization algorithm. In ASP, above formulas are encoded as follows (:- stands for $\leftarrow$):

Listing 1 ASP encoding of the task assignment

```
0{assign(T,R):robot(R,_,_,_)}1:-
  task(T,_,_).
:- assign(T,R), assign(T2,R), T != T2.
```

The first rule makes use of both a conditional literal and a cardinality constraint. A conditional literal $a : b_1, \ldots, b_n$ is a nested implication, where a and $b_1, \ldots, b_n$ can be seen as the head and the body of a rule respectively. The cardinality constraint is used to ensure that each task is assigned to one robot at most. Given $x\{head\}y$:- $body$, the meaning is that, for each different $body$ instantiation (for each task T in our case), the $head$ is instantiated from x to y times (from 0 to 1 in our case). In our code this implies that, for each task T, at most one robot R is assigned inside the head. The second rule is an integrity constraint. In the case that, after the task assignment was performed, unassigned vehicles are remaining, these free vehicles are assigned to charging stations and parking places. The rules used for this particular assignment problem are defined separately for vehicles on the field and vehicles currently in charging stations. A charging vehicle can only be assigned to a charging station if the battery level is below 90 %. Vehicles on the field can be sent to charging stations any time, regardless of the current battery level. Charging vehicles can go to a parking place only if the battery level is above or equal 90 %, whereas vehicles on the field can go to parking places independently from the battery level. In the original implementation, priority is given to vehicles with the lowest battery level. Similarly to the FIFO strategy in task assignment, first we assigne the least charged vehicle to the closest station, then the second

least charged one, and so on. However, this implementation shows its limits on circumstances where multiple robots have critical battery levels that differ only in a very small amount. For this reason, and since our goal with the declarative encoding is to improve the overall quality of the assignment, in the ASP implementation we minimize the overall travelling distance, like we do for the task assignment. The rules and constraints needed are very similar to the ones we used before, where PR1, PR2 are some user-defined parameters required for the assignment:

Listing 2 ASP encoding of the charge assignment - assignment rule

```
0{charge(S,R): station(S,PR1,PR2)}1
        :- robot(R,PR1,PR2,_).
```

3.2. The Incubed IT Use Case

Incubed IT is a robotics company focused on software development for smart robots. They typically deal with problems of the same type as the previous use-case we just discussed above. Thus the main topic is multi-robot planning and scheduling. For this reason, programmers at Incubed IT designed a highly parameterized platform which, if configured accordingly, can face a lot of different situations, like warehouses of online traders, logistic centers of supermarkets and car manufacturing plants. Fortunately, this platform is quite modular, partially centralized and partially decentralized, with a main FMS module which is responsible for the coordination of the many parts of the system. Thanks to this design, replacing the old solving module with the ASP solver has been easy to do.

In the imperative implementation, two kinds of optimization costs can be used: FIFO and global optimum. The former does not require more explanations, while the latter considers a priority number associated to each task. Regarding the task assignment, we stick to the important constraint rule in ASP: we can assign only one vehicle to a task, and only one task to a vehicle at a time. The same rules and constraints we used for the BMW use-case thus fit to Incubed IT software as well.

We can now focus on the other problem to solve, the charge assignment. The charging strategy here is more sophisticated than in the BMW-case, and robot can be sent to charge for four different reasons: *fixed time slot charging*: robots are assigned to charging stations due to a reached time slot; *critical charging*: robots are assigned to charging stations due to a battery level below the critical charging limit; *busy charging*: robots are assigned to charging stations due to a battery level below the busy charge limit; *idle charging*: robots are assigned to a charging station due to not enough appropriate assignable tasks. Obviously, all of these parameters (critical and busy charging limit, duration of the time slot) can be customized by the user. We define now the rules and constraints used to implement the third situation:

Listing 3 Assignment of busy charging robots

```
0{charge(S,R busy) : chargingstation(S
    ,_,_,_,_), robot_station(R,S)}1 :-
robot_charge_opt(R,BL,automatic_mode,_
    ,_,_,BCL,CCL), BL <= BCL, BL > CCL.
```

Listing 4 Avoidance of double allocations

```
:- charge(S,R,_),charge(S,R2,_),R!=R2.
:- charge(S,R,_),charge(S2,R,_),S!=S2.
:- assign(T,R),charge(_,R,_).
```

In contrast to the BMW case-study, here we do not handle the two problems of task and charge assignment separately: we optimize two different weighted criteria. The most important one is the minimization of the overall travelling distance of robots assigned to tasks or to charging stations due to forced timeslot, critical or busy charging. Then, the same optimization, with a lower weight, is applied to robots assigned to parking places and charging stations for idle charging.

4. Evaluation of Runtime and Quality for both Case-Studies

In this section, we present a brief evaluation of both case-studies. We designed several instances for each case-study involving different numbers of robots, orders, charging and parking stations to test different scales. Subsequent, the runtime as well as the quality of the solutions for these scenarios are compared. Furthermore, since Clingo can combine different meta-heuristics and parallelization strategies for the solving process, we tested all the combinations between them in order to find, for each case, the best one. As a result of the evaluation of these solving approaches [6], we chose the branch-and-bound-based optimization strategy in combination with splitting-based search multithreading and four threads for the two BMW assignment problems, while for Incubed IT the best approach is the

Vsids Heuristic combined with compete-based multithreading with four threads.

The systems of BMW and Incubed IT have been tested on devices with the following specifications. At BMW an Intel(R) Core(TM) i5 with a 1.70GHz processor and 8GB RAM is used. At Incubed IT an Intel(R) Core(TM) i5- 7200U is used with a 2.50GHz processor and 8GB RAM. On both systems Windows 10 is installed. Clingo is running in version 5.3.0 with Gringo V5.3.0. and Clasp V3.3.4.

4.1. Evaluation at BMW Group

In Table 1, the mean value and the standard deviation of the runtimes for all test scenarios (10 for each scenario) are shown and the number of solved test runs is given. If the optimal solution is not found within the BMW-specific time limit of 60 seconds, the solving process is aborted. Consequently, these aborted test runs are not considered in the calculations for the mean and standard deviation. The mean performance of the imperative method is for every scenario the best. As shown in the tables, two different ways of using ASP were tested. In the first one, the solver is directly called inside C#, while in the second we run ASP standalone. The serious performance issues of the former indicate potential for an improved incorporation of the ASP call in C#.

The instances are formed as follow: for the test scenario 1, we have 5 tasks and 5 robots; for the scenario 2, 20 tasks and 12 robots; finally, scenario 3 has 50 tasks and 30 robots. The positions of the robots and stations of the tasks are randomly placed on a $1000\,\mathrm{m} \times 1000\,\mathrm{m}$ area.

Looking at the results in Table 1 the imperative solution seems the winner, but in ASP not the Euclidean distance for single robot is optimised, but the traveling costs of the whole fleet. So, by using ASP, we are rewarded with far better quality solutions, as witnessed by Table 4, where traveling costs for scenario 3 are shown. This scenario is particularly interesting, since ASP was not able to find the provable optimal solutions within the time limit. Although, while looking for that, solvers like Clingo keep returning the best solution found so far, as soon as it finds a better one. Looking at Table 4, we can see that the best ASP solution found within 1 second considerably beats the C# solution. However, in this scenario we do not get an improvement with higher time limits. Results with the other scenarios are similar, with the imperative implementation never being close to the ASP traveling distance. This particular problem highlights the performance-quality trade-off between the two approaches.

In Table 2 the mean value and the standard deviation of the runtime of every test scenario is shown, considering the charge and park problem. Same rules as before are applied regarding the time limit of 60 seconds. The instances are formed as follow: 2 charging stations (CS), 3 parking places (PP) and 3 robots (R) for scenario 1; 7 CS, 14 PP and 3 R for scenario 2; finally, 17 CS, 33 PP and 30 R for scenario 3.

The imperative C# approach shows for all scenarios a better performance than the ASP-implementations, which, as in the task assignment problem, makes use of a different optimization, minimizing the overall travelling distance between robots and stations, while the C# program prioritizes the robots with the most critical battery level. In contrast to the task assignment, in this case the problem is too complex to ASP, which does not succeed in finding good quality solutions (Table 4) and, in some cases, it does not succeed to find a solution at all. This observation leads to the assumption that the encoding of the park and charge assignment problem in ASP is not optimal, as the performance of the task assignment encoding for similarly scaled instances is significantly better.

4.2. Evaluation at Incubed IT

In the Incubed IT use-case, the two problems, task assignment and park and charge assignment, are handled together, according to our characterization in the previous section. In Table 3, the mean value and standard deviation of the runtimes for the test scenarios solved with the original code and with the in-Java integrated ASP are shown, together with standalone ASP. A timeout is reached when a test run requires more than 30 seconds to find an optimal solution. Test runs that reached the timeout are not considered in the calculation for the mean and the standard deviation. The testing environment has a floor area of $100\,\mathrm{m} \times 86\,\mathrm{m}$ where the robots are freely movable. The three scenarios we are going to test are formed as follows: for scenario 1, we have 5 robots (R), 3 charging stations (CS), 7 parking places (PP) and 5 tasks (T); 10 R, 6 CS, 14 PP and 10 T for scenario 2; finally, for the last scenario we have 30 R, 18 CS, 42 PP and 15 T.

As we would expect from an NP problem solver,

Test Scenario	C# Implementation			ASP within C# Implementation			Standalone ASP		
	μ [ms]	σ [ms]	# TRS	μ [ms]	σ [ms]	# TRS	μ [ms]	σ [ms]	# TRS
1	0.00	0.00	10	415.50	18.16	10	8.30	8.92	10
2	0.30	0.48	10	2,802.20	4,445.21	10	1,428.90	2,746.91	10
3	0.00	0.00	10	/	/	0	/	/	0

Table 1: Runtime and solved test runs (TRS) for the different BMW task assignment implementations

Test Scenario	C# Implementation			ASP within C# Implementation			Standalone ASP		
	μ [ms]	σ [ms]	# TRS	μ [ms]	σ [ms]	# TRS	μ [ms]	σ [ms]	# TRS
1	0.00	0.00	10	473.80	81.24	10	13.90	8.88	10
2	16.20	4.87	10	788.10	350.75	10	341.70	404.17	10
3	1,753.30	127.58	10	/	/	0	/	/	0

Table 2: Runtime and solved test runs (TRS) for the different BMW park and charge assignment implementations

the reader can notice from the results that ASP is faster than the Java program while solving small instances, regardless of the number of constraints. Though, once the size of the problem hits the combinatorial blow-up point, it fails to return an optimal plan within time.

To measure the quality of the solution, we consider the two metrics we described in the previous section: most important are the overall travelling costs for task assignment and critical charging; the travelling distance of the other kinds of assignment (like for parking places) are then considered. Since the optimization strategy adopted with ASP is very similar to the one already used in the original program, in all the scenarios in which the optimal declarative solution is found within time, its quality w.r.t. to these metrics coincides to the Java solution quality. For this reason, like we did for the BMW case, when the ASP solver fails to find the optimal solution within the limit, we are interested in the analysis of the best ASP solution found so far. This situation shows up in the third scenario. Looking at Table 4, where the total cost (which is the weighted sum of the two metrics) is shown, it can be seen that the original implementation in Java provides a significantly faster and better solution than the ASP implementations.

5. Conclusion

The goal of this work is to make a comparison, in different real-world logistics scenarios, between the classic imperative paradigms and the declarative ones. Answer Set Programming was chosen because of its high efficiency, as witnessed by the many applications in industry. To achieve that, the FMS of BMW and Incubed IT were first analyzed, and then integrated with a new scheduler modeled in ASP. In the previous section, results and comparison between the two approaches in both companies are shown and analyzed. As we expected, there is not a clear winner between the two systems, but this comparison highlighted the pros and cons of both languages, whose performance highly depend on the kind and size of tasks to be accomplished. One main quality criterion of the FMS is the performance and the quality of the results. To evaluate the criterion, test scenarios have been set up that are based on typical use-cases of the FMS. Regarding the BMW use-case, the imperative solution is significantly faster than the declarative one, especially for the task assignment problem. However, in ASP we make use of a different optimization technique, which rewards with better solutions . This different strategy led to a trade-off between solving time and solution quality: if the imperative method is faster, ASP finds better solutions. The Incubed IT use-case gave instead different results, making clear how a very specific scenario can benefit from a particular approach rather than a general one. However, a common behavior can be seen from both BMW and Incubed IT, which represents the main weakness of ASP and enumeration tools in general. It does not scale over the size of the problem. Yet in the Incubed IT scenario in which ASP

Test Scenario	Java Implementation			ASP within Java Implementation			Standalone ASP		
	μ [ms]	σ [ms]	# TRS	μ [ms]	σ [ms]	# TRS	μ [ms]	σ [ms]	# TRS
1	278.30	226.64	10	121.30	143.50	10	71.90	14.96	10
2	491.30	223.23	10	495.00	339.16	10	278.67	236.67	10
3	2,572.80	4,894.22	10	/	/	0	/	/	0

Table 3: Runtime and solved test runs (TRS) for the different Incubed IT assignment implementations

Use-case	Imperative	ASP after 1 sec	ASP after 5 sec	ASP after 60 sec
BMW task assign.	7242	3834	3834	3834
BMW park & charge assign.	3217	6979	6530	4669
Incubed IT assign.	186	293	302	257

Table 4: Traveling costs for all the use-cases [m]

does not experience a combinatorial blow-up, it finds the best solution in less time than Java, without compromising over the quality.

To conclude, this work has shown that declarative programming can perform well on real-world logistics scenario, especially when we are interested in the quality of the optimization. Another important advantage of this approach is the separation between the description and the solving of the problem. In fact, performance can be an issue with ASP, especially in a dynamic planning scenario, but fortunately state-of-the-art solvers like Clingo or DLV come with many meta-heuristics and optimizations to play with. Once the proper settings for the specific scenario are found, solving time can improve considerably, without having to modify the code at all. In all the cases in which a greedy algorithm is proven to perform well, in terms of both quality and solving time, imperative programming still remains the best choice.

References

[1] *Reasoning Web. Semantic Technologies for Intelligent Data Access - 9th International Summer School 2013, Mannheim, Germany, July 30 - August 2, 2013. Proceedings*, volume 8067 of *Lecture Notes in Computer Science*. Springer, 2013.

[2] M. Abseher, M. Gebser, N. Musliu, T. Schaub, and S. Woltran. Shift design with answer set programming. *Fundam. Inform.*, 147(1):1–25, 2016.

[3] C. Dodaro and M. Maratea. Nurse scheduling via answer set programming. In M. Balduccini and T. Janhunen, editors, *Logic Programming and Nonmonotonic Reasoning*, pages 301–307. Springer International Publishing, 2017.

[4] E. Erdem, E. Aker, and V. Patoglu. Answer set programming for collaborative housekeeping robotics: representation, reasoning, and execution. 5(4):275–291, 2012.

[5] W. Faber, N. Leone, and S. Perri. The intelligent grounder of DLV. In E. Erdem, J. Lee, Y. Lierler, and D. Pearce, editors, *Correct Reasoning: Essays on Logic-Based AI in Honour of Vladimir Lifschitz*, pages 247–264, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[6] F. Fabricius. ASP-based Task Scheduling for Industrial Transport Robots. Master's thesis, Graz University of Technology, 2019.

[7] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. Answer set solving in practice. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 6(3):1–238, 2012.

[8] M. Gebser, R. Kaminski, A. König, and T. Schaub. Advances in gringo series 3. In J. P. Delgrande and W. Faber, editors, *Logic Programming and Nonmonotonic Reasoning*, pages 345–351, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[9] M. Gebser, R. Kaminski, and T. Schaub. Grounding recursive aggregates: Preliminary report. 2016. Workshop proceeding.

[10] M. Gebser, P. Obermeier, T. Schaub, M. Ratsch-Heitmann, and M. Runge. Routing driverless transport vehicles in car assembly with answer set programming. 18(3-4):520–534, 2018.

[11] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski, Bowen, and Kenneth, editors, *Proceedings of International Logic Programming Conference and Symposium*, pages 1070–1080. MIT Press, 1988.

[12] N. Leone and F. Ricca. Answer set programming: A tour from the basics to advanced development tools and industrial applications. In *Reasoning Web. Web Logic Rules: 11th International Summer School 2015*, pages 308–326, 07 2015.

[13] V. Nguyen, P. Obermeier, T. C. Son, T. Schaub, and W. Yeoh. Generalized target assignment and path finding using answer set programming. *IJCAI International Joint Conference on Artificial Intelligence*, pages 1216–1223, 2017.

[14] F. Ricca, G. Grasso, V. Lio, and S. Iiritano. Team-building with answer set programming in the gioia-tauro seaport. *Theory and Practice of Logic Programming*, 12(03):361–381, 2012.

[15] Z. G. Saribatur, E. Erdem, and V. Patoglu. Cognitive factories with multiple teams of heterogeneous robots: Hybrid reasoning for optimal feasible global plans. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2923–2930, 2014.

[16] S. Schieweck, G. Kern-Isberner, and M. ten Hompel. Using answer set programming in an order-picking system with cellular transport vehicles. *IEEE International Conference on Industrial Engineering and Engineering Management*, pages 1600–1604, 2016.

[17] M. Selmair, S. Hauers, and L. Gustafsson-Ende. Scheduling charging operations of autonomous agvs in automotive in-house logistics. *ASIM*, 2019.