Automated Planning and BDI Agents: a Case Study

Rafael C. Cardoso[1], **Angelo Ferrando**[2], **Fabio Papacchini**[3]

[1]Department of Computer Science – The University of Manchester
Manchester M13 9PL, UK

[2]Department of Computer Science – University of Genova
Genova 16145, Italy

[3]Department of Computer Science – University of Liverpool
Liverpool L69 3BX, United Kingdom

rafael.cardoso@manchester.ac.uk

angelo.ferrando@unige.it

fabio.papacchini@liverpool.ac.uk

Abstract. *There have been many attempts to integrate automated planning and rational agents. Most of the research focuses on adding support directly within agent programming languages, such as those based on the Belief-Desire-Intention model, rather than using off-the-shelf planners. This approach is often believed to improve the computation time, which is a common requirement in real world applications. This paper shows that even in complex scenarios, such as in the Multi-Agent Programming Contest with 50 agents and a 4 second deadline for the agents to send actions to the server, it is possible to efficiently integrate agent languages with off-the-shelf automated planners. Based on the experience with this case study, the paper discusses advantages and disadvantages of decoupling the agents from the planners.*

1. Introduction

Automated (or also referred to as Artificial Intelligence) planning consists of using a search algorithm to find a solution to a problem [Nau et al. 2004]. The planner receives as input information about the domain (predicates and actions that can be applied) and the problem (initial state of the environment and goals) and will apply a search algorithm to transition between states until the goals have been achieved. The output of a planner is a plan containing a sequence of actions that achieves the goals of the problem. The action theory in STRIPS (STanford Research Institute Problem Solver) [Fikes and Nilsson 1971] is the backbone of later formalisms such as the widely used PDDL (Planning Domain Definition Language) [Mcdermott et al. 1998]. PDDL has been the de-facto standard formalism for representing classical planning problems, but it has also been extended to support temporal, probabilistic, and other types of planning. In this paper, we focus on classical PDDL/STRIPS task planning.

Autonomous agents and Multi-Agent Systems (MAS) have vast and diverse subareas of research. In particular, we focus on Agent-based programming [Cardoso and Ferrando 2021], which uses agent-oriented languages to implement some of the concepts found in MAS. The Belief-Desire-Intention model [Bratman 1987, Rao and Georgeff 1995] is used in most agent-based programming languages [Bordini et al. 2020], as well as in research in the area of agent programming [Logan 2018]. The BDI model consists of a reasoning cycle based on three main concepts: *beliefs* – represent knowledge about the world, *desires* – goals to achieve, and *intentions*, steps that can be made towards achieving something. A simple reasoning trace starts with changes in the belief base which can trigger plans to achieve a goal, which if selected for execution will go through the stack of intentions of the plan and execute them one by one.

Most agent-based programming languages have some form of inherent task planning. Agents have access to a plan library and based on certain triggering events a search is made to find applicable plans from their library. The difference here is that interpreting actions (i.e., their pre-conditions and effects) is delegated to the environment and is not a concern of the agent. This kind of planning is more similar to the non-primitive tasks that can be found in Hierarchical Task Planning (HTN) [Nau et al. 2003]. Even though action descriptions are still needed for HTN planning, the non-primitive tasks (also called methods) are very similar to what plans represent in BDI programming in terms of search, that is, they allow the search space to be effectively pruned. Note that this is different from adding explicit support for automated planning in agent-based programming languages. We discuss approaches that do that later on in the Related Work section.

However, plans in the agents' libraries are usually pre-designed by a developer. Complex case studies and unpredictable environments may result in a plan library that does not contain the necessary plans to solve some of the problems that may appear. This is where using automated planners to generate those missing plans (either at runtime as we will demonstrate in this paper, or offline at design time) may remove the burden from the developer of trying to encode every possible solution.

In this paper we use an off-the-shelf planner in combination with our agents that were programmed using a BDI agent-based language to solve some complex problems in the 14th and 15th Multi-Agent Programming Contest[1] scenarios [Ahlbrecht et al. 2020]. The main problem we wanted to solve with this integration is to plan optimal movement of the agents in a grid environment with optional actions for clearing obstacles. Agents plan individually because they have partial view of the environment (i.e., they can only see a few squares around them), thus, a centralised approach would not be very effective. It is also important to note that since we have a grid environment with extra actions (i.e., state-based), task planners are much more suited to tackle this problem rather than traditional path planning algorithms, since there are no real actuators, effectors, or sensors in play.

We explain how we have made this integration work and what had to change between the 14th and 15th editions to still make this strategy effective. In particular, we have used the Fast Downward[2] [Helmert 2006, Helmert 2009] planner for the automated planning, and the JaCaMo[3] [Boissier et al. 2013, Boissier et al. 2020] multi-agent oriented programming framework.

The rest of this paper is organised as follows. In the next section we discuss some

[1]`https://multiagentcontest.org/`
[2]`http://www.fast-downward.org/`
[3]`http://jacamo.sourceforge.net/`

approaches that have tried to explicitly incorporate planning in BDI languages, as well as BDI applications that have used off-the-shelf planners in the past. Section 3 briefly explains our case study, with a particular focus on the elements that were advantageous to apply automated planning. In Section 4, we describe in detail how we have used an off-the-shelf planner to solve the problems described in the previous section. A discussion about our experience with the combination of automated planning and BDI agents is presented in Section 5. We conclude the paper and list some future directions in Section 6.

2. Related Work

Automated planning and BDI programming have been used individually to solve an assortment of different tasks in the past. An example of the former is [Borgo et al. 2016], where an off-the-shelf planner is used in a manufacturing plant for logical reconfiguration of the control nodes after changes in the environment (production change, fault in physical components, or changes in the production goals). An example of the latter is [Cardoso et al. 2021], where agents are implemented in a BDI-based programming language to support ethical reasoning by adding agents that can recommend ethical actions to be executed.

Some approaches have been developed which aim to integrate BDI programming languages with off-the-shelf automated planners. An example of such approach is the work in [Cardoso and Bordini 2019], which uses the SHOP2 HTN planner [Nau et al. 2003] without requiring any modifications to the planner itself. Instead, the agents in the JaCaMo framework call an individual instance of the planner when required to perform multi-agent planning, with planning being coordinated and tasks allocated at runtime using agent communication protocols and techniques. There are several differences between our work and theirs: (a) we do not perform coordinated multi-agent planning, our agents plan individually to each other; (b) we use a PDDL planner while they used HTN; and (c) we focus on a more practical and complex case study.

Conversely, there have been many approaches that tried to integrate automated planning directly into BDI programming languages. In [de Silva et al. 2009] first principles classical planning is introduced to a theoretical (never implemented) BDI language through the derivation of abstract planning operators from BDI programs. A further extension of this work is reported in [Sardiña and Padgham 2011], which adds failure handling and declarative goals on top of the planning, however even though the theoretical contributions were important at the time, the main issue still remains that the language described was never implemented. A more practical approach with a mapping between classical planning formalisms and traditional BDI agent languages is presented in [Meneguzzi and Luck 2013], describing a formal translation process from BDI plans to classical planning operators. We believe there are a number of fundamental differences in using off-the-shelf planners and integrating automated planning directly into BDI programming languages. We discuss what these differences are in Section 5.

3. Multi-Agent Programming Contest: Agents Assemble

The Multi-Agent Programming Contest (MAPC) is an annual competition with complex multi-agent oriented challenges (such as communication, coordination, and interaction between agents) that aims to promote and strengthen the use of multi-agent programming frameworks, tools, and methodologies. By applying these techniques to difficult scenarios it is possible to identify which features are missing, can be improved, or have to be fixed in agent-based technologies. The scenarios change every few years, with extensions being done to the previous scenarios on off years. The core structure of the simulations remain mostly the same: it is a synchronous step-based simulation wherein clients (the teams) have to send actions to the server under a certain deadline (usually 4 seconds per step), and a match is composed of 3 rounds (each with a different map).

The 14th edition of the MAPC [Ahlbrecht et al. 2020] introduced the "Agents Assemble" scenario. In this scenario, two teams of 10 agents each compete to accumulate the most amount of currency by assembling block structures to match tasks announced by the server in a grid environment. We participated as team LFC (Liverpool Formidable Constructors) [Cardoso et al. 2020] and achieved first place. We believe one of the main factors in our exemplary performance during that edition of contest was due to our strategy in using an off-the-shelf planner to generate plans for movement at runtime. The scenario has many other details which required an assortment of different strategies that the interested reader can find more information about in [Ahlbrecht et al. 2020, Cardoso et al. 2020]. For the remaining of this paper, we focus only on the elements of the contest that were relevant for the use of the automated planner and its integration with the agents.

The 15th edition of the MAPC [4] used the same scenario but extended it in many interesting ways. Of relevance to our planning strategy was the change from the static 10 agents to 15 agents in round 1, 30 agents in round 2, and 50 agents in round 3. This required us to adapt our strategy, as calling 30 and 50 instances of the planner would slow down the reasoning of our agents and make them unable to send an action before the deadline. Our solution to this was developing a plan cache, which we go into detail later on in Section 4.2. We achieved second place in the 15th edition of the MAPC, however we believe our planning strategy performed very well and we only lost due to the lack of optimisation in some of our other strategies (such as task assembly).

In this paper, we focus on the challenge of performing efficient movement in the grid. Map grids during the 14th and 15th MAPC ranged from 50x50 (2500 cells) to 100x100 (10000 cells). Each agent has a local view of 61 cells around them. An example of the local view of an agent is shown in Figure 1. Initially, we tried to encode the agents' movements directly into the agent program, however, we soon realised that there were too many edge cases to consider and that our solution resulted in longer routes and even a few deadlocks. Using an automated off-the-shelf planner allowed us to focus on the other strategies while knowing that the movement of our agents were very efficient.

The main challenge for planning the agents' movement (either directly in the agents' program or in an automated planner) is the dynamic nature of the environment and the lack of knowledge of the agent about cells outside its local vision. For example, the following dynamic events can happen during a step: (a) an agent from the other team (or from our own team, if we do not disclose movement information to other agents) may move into one of the cells that are a part of our agent's route, generating a conflict; (b) a special event called a *clear* event has a random chance of occurring each step which may

[4] https://multiagentcontest.org/2020/

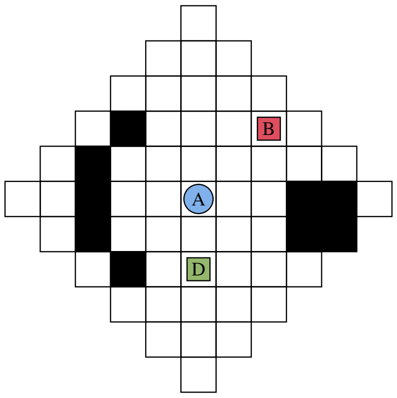


Figure 1. Example of local view of an agent (A). Black squares are obstacle cells, green squares with a (D) are dispensers that can spawn blocks when requested by an agent, and red squares with a (B) are blocks.

remove obstacle cells, create obstacle cells, disable agents, and remove blocks; and (c) agents also have access to a *clear* action, however compared to the environment event this action has a reduced radius and can not create obstacles.

4. Automated Planning and BDI Agents

Our team[5] was programmed in the JaCaMo multi-agent oriented development framework using its Eclipse plugin. JaCaMo has three main programming dimensions (agent, environment, and organisation), each handled by a different tool that has been developed over many years and then integrated to work well with each other. Jason [Bordini et al. 2007] is a popular BDI-based agent programming language that has been shown to be one of the most efficient agent-based languages in several benchmarks [Cardoso et al. 2013, Mohajeri Parizi et al. 2020]. We used Jason to define our agent programs. CArtAgO [Ricci et al. 2009] is based on the notion of artifacts that are used to represent and interact with the environment, and in our case this meant interfacing with the server of the MAPC as well as interfacing with the off-the-shelf planner. Finally, Moise [Hübner et al. 2007] provides organisation constructs to determine roles and norms in groups of agents, which our team used to aid in the coordination of certain tasks.

Fast Downward (FD) is a well-established planner which has been used several times in the International Planning Competition (IPC). For example, different variations of the planner were used in the IPC in 2018[6] and performed well in the classical, satisficing, and bounded-cost tracks[7]. Clear actions take three steps to be successfully used by an agent, and we needed to encode it as a planning operator because clearing obstacles can make much more optimal routes. However, FD does not support numeric planning. To workaround this problem we made use of action costs where all movement actions have a cost of 1 and the clear action has a cost of 3, and then the planner is asked to find plans that can minimise the cost.

[5]Our team code for the 14th MAPC:
`https://github.com/autonomy-and-verification-uol/mapc2019-liv`
Our team code for the 15th MAPC:
`https://github.com/autonomy-and-verification-uol/mapc2020-lfc`

[6]`https://ipc2018.bitbucket.io/`

[7]`https://ipc2018-classical.bitbucket.io/#results`

Next, we describe our planning strategies for the 14th MAPC (Local Vision Planning) as well as the necessary extensions for it to work well in the 15th MAPC (Plan Cache).

4.1. Local Vision Planning

We limited the use of the planner only for performing the movement of the agents in the grid (which includes the clear action for more efficient routes). If the grid cells were static and the map was completely observable then it would have made more sense to try to plan the entire route of the agent at once. However, apart from the fact that this solution would likely result in performance problems, the cells in the grid were dynamic and could change because of clear actions or agents' actions. The local vision and the lack of knowledge of the remaining cells in the map also made it pointless to plan too far in advance. Therefore, we call the planner with a target cell that is inside the local vision of the agent. An example of how this works is given later on.

Planning is performed at runtime and an instance of the FD planner is invoked by the agents when they required planning. The workflow is defined as follows:

1. The agent wants to move to a target position in the grid
2. Is the target position inside the agent's vision? If yes jump to step 4
3. Select the closest cell to the final target position that is inside the agent's vision
4. Determine what type of domain will be used, planning operators change depending on specific circumstances:
    - if the agent has enough energy to perform clear actions enable clear as a planning operator
    - if the agent has a block attached (maximum one block attached for planning movement) enable operators to move with one block
5. Call a CArtAgO artifact which will generate the problem file based on the local vision of the agent (obtained through the perceptions observed in that step from the server), with the target cell as a goal, and the appropriate domain (based on the flags determined in previous steps)
6. Call an instance of the FD planner with the generated problem file and the appropriate pre-generated domain file
7. The solution plan is saved onto a file which the CArtAgO artifact reads and then translates it back to the agent
8. If a plan was found, the agent simply executes each action of the plan at each step (if yes jump to step 10)
9. If a plan is not found, then the agent tries to move (without the planner) in a direction that would bring it closer to the final target, and then calls the planner again (back to step 2)
10. If the agent is not in the final target cell after executing all actions in the plan, the process goes back to step 2, otherwise planning has finished

The 14th edition of the MAPC had 10 agents per team. Initially, we thought that if all of our 10 agents started an instance of the planner at the same step, then we may have encountered some performance problems. To circumvent this, we introduced a planning counter which would limit the number of agents allowed to start an instance of the planner at any given step when the counter hits the maximum number allowed. Setting this number varied depending on the processing power of the computer that was

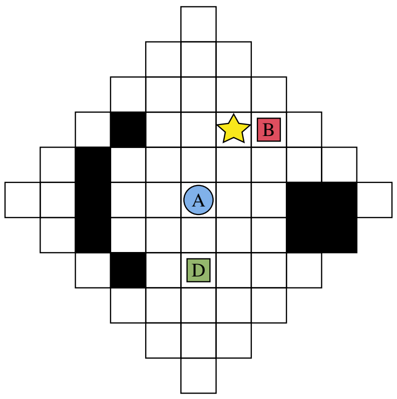
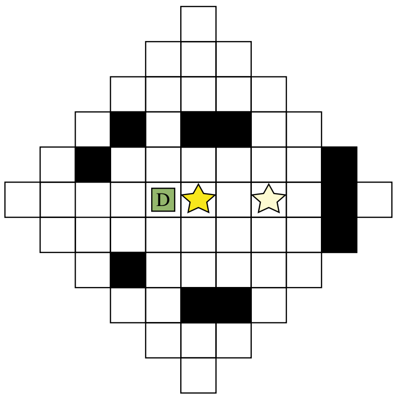
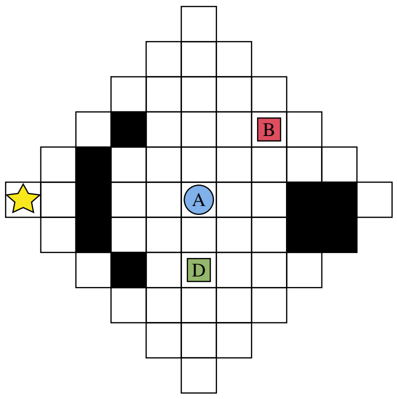


Figure 2. Example of a final target position (star) inside the vision of the agent.



(a) Example of where the final target position would be with respect to Figure 3b. The faded star represents the expected position of the agent after Figure 3b.



(b) Example of a partial target position (star) with a final target outside the vision of the agent.

Figure 3. Examples of planning targets (destination cells) for when the target is outside the vision of the agent.

being used. During the contest, we disabled this feature, as the computer we used to run was powerful enough to comfortably handle the maximum number of agents.

Example.
To better explain the difference between local and final target selection we exemplify some scenarios in Figure 2 and Figure 3.

Figure 2 shows an example of when the final target position is inside the vision of the agent. Since we are dealing with local vision the coordinates are all local with respect to the agent, that is, the agent is in cell (0,0). Therefore, the target position in this example is set to cell (1,-2), one cell to the right and two cells above. This is the simplest scenario, and when the agent arrives at the destination planning is concluded.

Figure 3a has a target cell that is outside the local vision of the agent. It would be inefficient to try to plan the whole route since things could change and then it would require replanning the route multiple times. Therefore, first we must find a target cell inside the agent's vision that would bring it closer to the final target, which in this case is shown in Figure 3b to be cell (-5,0). Upon arriving at the local target the agent recalculates the difference to its final target, which in this example would result in cell (-2,0), and then calls the planner once again.

4.2. Plan Cache

Using automated planning at runtime is far from being an easy task. The entire process, from encoding the state of the agent as a problem file, to solving it with a planner, is computationally demanding. This can be reasonable for small and simple applications, but it becomes an issue when applied to large and complex systems, such as a MAS. Indeed, even though the planning problem for a single agent is feasible, it might not be for a coalition of agents as well. Let us assume we have N agents, we would need to call the planner N times (one per agent). Considering the MAPC scenario, this could happen in each step of the simulation.

Since physical resources (CPU, memory) and time (how long an agent can wait) are finite, it is always possible to pick a number N of agents for which it is not possible to solve the planning problem in less than a certain amount of time (4 seconds for each simulation step in the MAPC). In particular, as we mentioned previously the 15th edition of the contest extended the scenario to have 15 agents in round 1, 30 agents in round 2, and 50 agents in round 3. Through testing, we noticed that our previous strategy managed to hold up for 15 agents, but it did not work for 30 and 50 agents. Because of this, alternatives to speed the planning process up must be considered.

We investigated alternatives to speed up the process, and found that one possible way to make the planning process faster is by *caching* the plans. By caching, we refer to the act of storing previously generated plans, instead of forgetting about them after execution. When an agent asks the planner to solve a problem, if such a problem has been already solved in the past, there is not really the necessity for the planner to waste time in solving it again. This can be achieved by keeping a mapping between $Problem \rightarrow Plan$, which given a problem file, returns its corresponding plan (if present in the cache). When a problem file does find a match in the cache, it means that it has never been solved before (cache miss), in which case the execution continues by calling the planner and then updating the cache. Note that this cache is saved independent of the size of the grid, number of agents, or any other parameter. This means that the cache can be used in any configuration to speed up the planning process.

The first aspect we have to consider about caching is how to encode a problem file, so it can be straightforwardly retrieved from the cache. Such an encoding must uniquely identify a problem file. Thus, all information which describe the agent's local view needs to be considered. In Figure 4, we report an example of how we generate such encoding.

Starting from the agent's local view (left side of Figure 4), a possible encoding can be obtained by unrolling the grid as a one-dimensional array (top right of Figure 4). The unrolling starts from the upper most cell, and then all rows are appended one by one from left to right. Finally, in the bottom right of Figure 4, we can see a possible encoding of the unrolling; where empty spaces are mapped[8] to 0 (dispensers and the agent current position are considered empty), obstacles to 1, blocks to 2[9], and the goal to 3.

[8]The mapped values are not semantically relevant, as long as is preserved for all mappings.

[9]After the contest we realised that since blocks are considered as obstacles for the planner, we could have set their value to 1, which would have decreased the number of cached plans at least by a small margin.

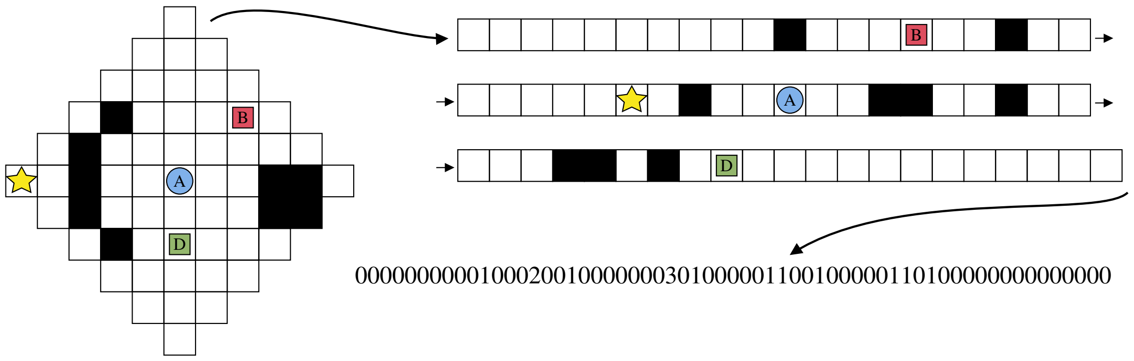


Figure 4. Example of the step by step process of converting from a problem description to an encoding that can be easily read.

Once the encoding is created, it can be used to query the cache. Since we want to use the cached plans amongst different executions, the cache is stored in the secondary memory. More in detail, for each cached plan, a file is stored. The encoding of the agent's local view is used to name such a file. Consequently, to check if a certain planning problem has already been solved, it is enough to look if there exists a file named as the encoding of the problem. If such a file exists, there is no need to call the planner since the corresponding plan is already available inside the file, otherwise, the planner is called and a new file is stored (named after the encoding of the problem).

For instance, in the example of Figure 4, the plan returned and stored as a file (named using the encoding) will contain the sequence of actions reported in Listing 1.

```
clear(-3,0)
move(w)
move(w)
move(w)
move(w)
move(w)
```

Listing 1. Plan returned by the planner for Figure 4, and stored in the cache.

As we have seen previously, it is possible to call the planner in two different settings. The first one is calling for a plan to the agent by itself, which we have already covered how to encode it. The second is that we may call for a plan to the agent with a block currently attached. The encoding for this is almost identical, except that in this case we append the local view coordinates of the block that is attached (note that our planning domain only supports movement with up to one attached block). For example, if there is a block attached to the agent and the block is located one cell below the agent, then we would append 01 to the beginning of the encoding.

5. Discussion

The most glaring benefit of using an off-the-shelf planner is that we are able to take advantage of the many sophisticated and efficient tools that have been developed over the years in the community of automated planning. Re-implementing the features provided by these tools in an agent programming language can take a lot of effort. A more subtle advantage is that if the application domain requires distinct planning techniques for different types of problems in the system then we can use multiple off-the-shelf planners that are most appropriate for each problem. For example, we may need at one point to perform probabilistic planning, and at another stage we require temporal planning.

A limiting factor in using off-the-shelf planners with autonomous agents at runtime in time critical applications is that the solution may not be produced inside acceptable time bounds. Initially, we did not consider using automated planners to be feasible because we thought we would run into this issue. However, after testing we noticed that for 10 agents we could comfortably perform planning under a four seconds restriction. When the number of agents increased to 30 and 50, then we had to modify our solution by performing plan caching which provided excellent results. Our initial analysis indicates that using off-the-shelf planners in domains with loosely coupled agents (low or no interactions between agents) and/or low number of agents should generate results very quickly. Note that determining if the result can be produced inside the time constraint depends specifically on the application domain. Alternative solutions such as plan caching can also make previously unfeasible planning scenarios work well, but the size of the cache must be controlled since it can eventually increase up to a point where it is no longer worth using.

The seamless integration of a planner in an agent programming language has the advantage that the computing performance can be vastly superior. This is not only because the agents themselves can perform planning directly without having to call a separate process, but also because *lookahead* online planning can be used, wherein only a subset of the problem is planned for and then executed before proceeding to plan for the rest.

The direct integration of planning in agents can restrict the type of planning supported, which can limit its applicability to certain domains. Furthermore, it can make the code difficult to port to other agent languages. For example, in the MAPC it is required for all teams to make their source code available after the contest. This can help new teams (or under performing teams) to look for inspiration in strategies that have worked well in the past. Because we used an off-the-shelf planner, other teams can easily make similar use of it without being tied to the agent development framework we used (JaCaMo).

6. Conclusion

In this paper, we have described our experience in applying an off-the-shelf automated planner at runtime for a very complex problem requiring a solution under a strict deadline. We have integrated this solution with agents that were developed in a BDI agent programming language. Our case study was based on the scenario first introduced in the 14th edition of the MAPC (and then later extended on the 15th edition). There are advantages and disadvantages to either using off-the-shelf planners or integrating automated planning directly in an agent programming language. Ultimately, the best choice is going to depend on the complexity and limitations imposed by the application/case study as well as the tools and languages that are available to solve the problem.

Future work involves trying to categorise different classes of applications and case studies to understand at a more general level when can an off-the-shelf planner be used alongside an agent programming language with minimal effort, as well as listing possible combinations of planners and agent-based languages that could work well together.

Acknowledgements

Work supported by UK Research and Innovation, and EPSRC Hubs for "Robotics and AI in Hazardous Environments": EP/R026092 (FAIR-SPACE) and EP/R026084 (RAIN). Cardoso's work is also supported by Royal Academy of Engineering.

References

Ahlbrecht, T., Dix, J., Fiekas, N., and Krausburg, T. (2020). The multi-agent programming contest: A résumé. In Ahlbrecht, T., Dix, J., Fiekas, N., and Krausburg, T., editors, *The Multi-Agent Programming Contest 2019*, pages 3–27, Cham. Springer International Publishing.

Boissier, O., Bordini, R., Hubner, J., and Ricci, A. (2020). *Multi-Agent Oriented Programming: Programming Multi-Agent Systems Using JaCaMo*. Intelligent Robotics and Autonomous Agents series. MIT Press.

Boissier, O., Bordini, R. H., Hübner, J. F., Ricci, A., and Santi, A. (2013). Multi-agent oriented programming with JaCaMo. *Science of Computer Programming*, 78(6):747–761.

Bordini, R. H., Seghrouchni, A. E. F., Hindriks, K. V., Logan, B., and Ricci, A. (2020). Agent programming in the cognitive era. *Auton. Agents Multi Agent Syst.*, 34(2):37.

Bordini, R. H., Wooldridge, M., and Hübner, J. F. (2007). *Programming Multi-Agent Systems in AgentSpeak using Jason*. John Wiley & Sons.

Borgo, S., Cesta, A., Orlandini, A., and Umbrico, A. (2016). A planning-based architecture for a reconfigurable manufacturing system. In *Proceedings of the 26th International Conference on International Conference on Automated Planning and Scheduling*, ICAPS'16, pages 358–366, London, UK.

Bratman, M. E. (1987). *Intentions, Plans, and Practical Reason*. Harvard University Press.

Cardoso, R. C. and Bordini, R. H. (2019). Decentralised planning for multi-agent programming platforms. In *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems*, AAMAS '19, pages 799–807, Richland, SC. International Foundation for Autonomous Agents and Multiagent Systems.

Cardoso, R. C. and Ferrando, A. (2021). A review of agent-based programming for multi-agent systems. *Computers*, 10(2):16.

Cardoso, R. C., Ferrando, A., Dennis, L. A., and Fisher, M. (2021). Implementing ethical governors in bdi. In *9th International Workshop on Engineering Multi-Agent Systems*.

Cardoso, R. C., Ferrando, A., and Papacchini, F. (2020). Lfc: Combining autonomous agents and automated planning in the multi-agent programming contest. In *The Multi-Agent Programming Contest 2019*, pages 31–58, Cham. Springer International Publishing.

Cardoso, R. C., Zatelli, M. R., HÃ¼bner, J. F., and Bordini, R. H. (2013). Towards benchmarking actor- and agent-based programming languages. In *Workshop on Programming based on actors, agents, and decentralized control*, pages 115–126, Indianapolis, Indiana, USA.

de Silva, L., Sardiña, S., and Padgham, L. (2009). First principles planning in bdi systems. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems - Volume 2*, AAMAS '09, pages 1105–1112, Richland, SC. International Foundation for Autonomous Agents and Multiagent Systems.

Fikes, R. E. and Nilsson, N. J. (1971). Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3):189 – 208.

Helmert, M. (2006). The fast downward planning system. *J. Artif. Intell. Res.*, 26:191–246.

Helmert, M. (2009). Concise finite-domain representations for PDDL planning tasks. *Artif. Intell.*, 173(5-6):503–535.

Hübner, J. F., Sichman, J. S., and Boissier, O. (2007). Developing organised multiagent systems using the MOISE+ model: programming issues at the system and agent levels. *Int. J. Agent-Oriented Software Engineering*, 1(3/4):370–395.

Logan, B. (2018). An agent programming manifesto. *International Journal of Agent-Oriented Software Engineering*, 6(2):187–210.

Mcdermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., Weld, D., and Wilkins, D. (1998). PDDL - The Planning Domain Definition Language. Technical Report TR-98-003, Yale Center for Computational Vision and Control.

Meneguzzi, F. and Luck, M. (2013). Declarative Planning in Procedural Agent Architectures. *Expert Systems with Applications*, 40(16):6508 – 6520.

Mohajeri Parizi, M., Sileno, G., van Engers, T., and Klous, S. (2020). Run, agent, run! architecture and benchmarking of actor-based agents. In *Proceedings of the 10th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE 2020, pages 11–20, New York, NY, USA. Association for Computing Machinery.

Nau, D., Ghallab, M., and Traverso, P. (2004). *Automated Planning: Theory & Practice*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Nau, D. S., Ilghami, O., Kuter, U., Murdock, J. W., Wu, D., and Yaman, F. (2003). SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research*, 20(1):379–404.

Rao, A. S. and Georgeff, M. (1995). BDI Agents: From Theory to Practice. In *Proc. 1st Int. Conf. Multi-Agent Systems (ICMAS)*, pages 312–319, San Francisco, USA.

Ricci, A., Piunti, M., Viroli, M., and Omicini, A. (2009). Environment programming in CArtAgO. In *Multi-Agent Programming: Languages, Tools and Applications*, Multiagent Systems, Artificial Societies, and Simulated Organizations, chapter 8, pages 259–288. Springer.

Sardiña, S. and Padgham, L. (2011). A BDI Agent Programming Language with Failure Handling, Declarative Goals, and Planning. *Autonomous Agents and Multi-Agent Systems*, 23(1):18–70.