28th International Conference on
Automated Planning and Scheduling

June 24-29, 2018, Delft, the Netherlands

KEPS 2018

Proceedings of the Workshop on

Knowledge Engineering for Planning and Scheduling

Edited by:

Lukáš Chrpa, Ron Petrick,
Mauro Vallati, Tiago Vaquero.

Organization

Lukas Chrpa, Czech Technical University & Charles University in Prague, Czech Republic

Ron Petrick, Heriot-Watt University, UK

Mauro Vallati, University of Huddersfield, UK

Tiago Vaquero, CalTech, USA

Program Committee

Roman Barták, Charles University, Czech Republic

Daniel Borrajo, Universidad Carlos III de Madrid, Spain

Amedeo Cesta, National Research Council of Italy (CNR), Italy

Susana Fernandez, Universidad Carlos III de Madrid, Spain

Simone Fratini, European Space Agency - ESA/ESOC, Germany

Alan Lindsay, University of Huddersfield, UK

Lee Mccluskey, University of Huddersfield, UK

Andrea Orlandini, National Research Council of Italy (ISTC-CNR), Italy

Simon Parkinson, University of Huddersfield, UK

Patricia Riddle, University of Auckland, New Zealand

Shirin Sohrabi, IBM Research, USA

Dimitris Vrakas, Aristotle University of Thessaloniki, Greece

Contents

Evaluating a Knowledge-Based Scheduling Assistant

Neil Yorke-Smith
Delft University of Technology, Netherlands, and
American University of Beirut, Lebanon
n.yorke-smith@tudelft.nl

Abstract

We summarize a recent article that studies the evaluation of a knowledge-based scheduling system. The article considers a user-adaptive personal assistant agent designed to assist a busy knowledge worker in time management. We examine the managerial and technical challenges of designing adequate evaluation and the tension of collecting adequate data without a fully functional, deployed system. The PTIME agent was part of the CALO project, a seminal multi-institution effort to develop a personalized cognitive assistant. The project included a significant attempt to rigorously quantify learning capability in the context of automated scheduling assistance. Retrospection on negative and positive experiences over the six years of the project underscores best practice in evaluating user-adaptive systems. Through the lessons illustrated from the case study, the article highlights how development and infusion of innovative technology must be supported by adequate evaluation of its efficacy.

Evaluation of the Personalized Time Management (PTIME) Agent

The case study article by Berry et al (2017) reports and critiques the *evaluation* of a knowledge-based scheduling system that learns preferences over an extended period. The domain of application is personal time management, in particular, providing assistance with arranging meetings and managing an individual's calendar. The *Personalized Time Management* (PTIME) calendaring assistant agent increased in usefulness as its knowledge about the user increases. The enabling technologies involved were preference modelling and machine learning to capture user preferences, natural language understanding to facilitate elicitation of constraints, and constraint-based reasoning to generate candidate schedules (Berry et al. 2011). Human-computer interaction (HCI) and interface design played central roles.

The PTIME system was part of a larger, seminal project, *Cognitive Assistant that Learns and Organizes* (CALO), aimed at exploring learning in a personalized cognitive assistant. Thus, the primary assessment of PTIME was in terms of its adaptive capabilities, although such a knowledge-based system must necessarily have a certain level of functionality to assist with tasks in time management, in order to provide a context for learning.

Figure 1: Screenshot of the PTIME system.

At the commencement of the project, however, the degree of robustness and usability required to support evaluation was not immediately obvious. Evaluation was focused almost exclusively on the technology; experiments were designed to measure performance improvements due to learning within a controlled test environment intended to simulate a period of real-life use—rather than in a genuinely 'in-the-wild' environment. Technologists such as the majority of the authors are trained primarily to conduct such 'in-the-lab' evaluations, but—as argued in the article—many situations require placing the technology into actual use with real users in a business or personal environment, in order to provide a meaningful assessment. In retrospect, the authors suggest that the evaluation methodology of CALO gave too little attention to the usefulness and usability of the technology.

Scheduling in PTIME

We briefly review the role of automated scheduling in the PTIME system. PTIME consists of four main components: user interface, calendar proxy, constraint reasoner, and preference learner. In its main mode of operation, PTIME elicits an event request: for the user, this corresponds to stating details on the desired event to be arranged; these details correspond to constraints.

PTIME computes preferred candidate schedules (possi-

bly relaxations) in response to the request and presents a ranked subset of the candidate schedules to the user. Note that, because PTIME will consider moving existing events if necessary, the options presented to the user are schedules rather than single events. The number of such candidate schedules presented depends on the number of feasible schedules. PTIME will typically display 10 candidate schedules, including a mix of more optimal and more diverse options.

PTIME accepts the user's selection from among the presented candidate schedules. PTIME updates the preference model accordingly, based on the implicit feedback of the selected versus non-selected options. The updated model is used in the subsequent interactions.

These steps repeat as necessary, with the system presenting new or refined options after each new detail is entered or modified by the user. Through a collaborative negotiation process, event invitees comment, respond, and counter-propose to reach agreement over the event.

At the heart of the scheduling is the constraint reasoner. This component generates scheduling options in response to new or revised details and constraints from the user, using the current preference model to generate preferred options. The reasoner translates requests such as *"next tues afternoon with nigel and kim"* into a set of soft constraints, and solves a soft constraint problem with preferences (Moffitt, Peintner, and Yorke-Smith 2006). Soft constraints allow all aspects of the user's request—including times, location, and participants—to be relaxed in the case where the request cannot be satisfied, i.e., when the scheduling problem is over-constrained. Details of the constraint solving, and the other aspects of the system, are given in Berry et al. (2006; 2009; 2011).

Lessons Learned

The six lessons that emerged from the evaluation journey with PTIME are not unfamiliar from other experiences of evaluating (non-adaptive) systems (Cohen and Howe 1989; Nielsen and Levy 1994; Chen and Pu 2014):

1. The contexts of the use of technology, and the competing interests of the stakeholders, must be a primary focus in designing an evaluation strategy.
2. Evaluating one component based on an evaluation of a whole system can be misleading, and vice versa.
3. User-adaptive systems require distinct evaluation strategies.
4. In-the-wild evaluation is necessary when factors affecting user behaviour cannot be replicated in a controlled environment.
5. In-the-wild evaluation implies significant additional development costs.
6. Ease of adoption of the system by users will determine the success or failure of a deployed evaluation strategy.

Our hope is that, since the conclusion of the CALO project, these lessons are increasingly understood in Artificial Intelligence and its constituent communities, including the automated planning/scheduling community. Indeed, Foster and Petrick (2017) contrast differences between the latter community and the dialogue systems community. They discuss the overhead of integration, deployment in real-world environments, and the need to evaluate certain types of systems in-the-wild—as all encountered in the case of PTIME.

Summarizing the article, the main lesson from this case study of evaluation of a knowledge-based scheduling system is obvious but under-valued: researchers and project managers benefit from familiarity with and adoption of best practice in evaluation methodologies from the start of a technology project.

Acknowledgements Thanks to the KEPS workshop reviewers. This material is based in part upon work supported by the US Defense Advanced Research Projects Agency (DARPA) Contract No. FA8750-07-D-0185/0004. Views are the author(s) and do not necessarily reflect the views of DARPA. A shorter version of this article abstract was presented at the 29th Benelux Conference on Artificial Intelligence (BNAIC'17).

References

Berry, P. M.; Conley, K.; Gervasio, M.; Peintner, B.; Uribe, T.; and Yorke-Smith, N. 2006. Deploying a personalized time management agent. In *Proc. of AAMAS'06*, 1564–1571.

Berry, P. M.; Donneau-Golencer, T.; Duong, K.; Gervasio, M. T.; Peintner, B.; and Yorke-Smith, N. 2009. Mixed-initiative negotiation: Facilitating useful interaction between agent/owner pairs. In *Proc. of AAMAS'09 Workshop on Mixed-Initiative Multiagent Systems*, 8–18.

Berry, P. M.; Gervasio, M.; Peintner, B.; and Yorke-Smith, N. 2011. PTIME: Personalized assistance for calendaring. *ACM Transactions on Intelligent Systems and Technologies* 2(4):40:1–40:22.

Berry, P. M.; Donneau-Golencer, T.; Duong, K.; Gervasio, M.; Peintner, B.; and Yorke-Smith, N. 2017. Evaluating intelligent knowledge systems: Experiences with a user-adaptive assistant agent. *Knowledge and Information Systems* 52:379–409.

Chen, L., and Pu, P. 2014. Experiments on user experiences with recommender interfaces. *Behaviour & IT* 33(4):372–394.

Cohen, P., and Howe, A. E. 1989. Toward AI research methodology: Three case studies in evaluation. *IEEE Transactions on Systems, Man, and Cybernetics* 19(3):634–646.

Foster, M. E., and Petrick, R. P. A. 2017. Separating representation, reasoning, and implementation for interaction management: Lessons from automated planning. In Jokinen, K., and Wilcock, G., eds., *Dialogues with Social Robots: Enablements, Analyses, and Evaluation*, volume 427 of *Lecture Notes in Electrical Engineering*. Springer. 93–107.

Moffitt, M. D.; Peintner, B.; and Yorke-Smith, N. 2006. Multi-criteria optimization of temporal preferences. In *Proc. of CP'06 Workshop on Preferences and Soft Constraints*, 79–93.

Nielsen, J., and Levy, J. 1994. Measuring usability: Preference vs. performance. *Communications of ACM* 37(4):66–75.

On Learning from Human Expert Knowledge for Automated Scheduling

Neil Yorke-Smith
Delft University of Technology, Netherlands, and
American University of Beirut, Lebanon
n.yorke-smith@tudelft.nl

Abstract

Automated scheduling systems and decision support tools require at least four kinds of knowledge: 1) domain knowledge, 2) problem instance knowledge, 3) control knowledge, and 4) solving knowledge. This short paper draws attention to learning from human experts for these different kinds of knowledge, and advocates a complementarity of knowledge acquisition by automated techniques and by human knowledge engineers.

Introduction

Knowledge – computational knowledge – is the fulcrum of Artificial Intelligence. Whether hand-coded in a logical formalism, or extracted from data by a deep learning network, knowledge is the basis for computation. AI-based scheduling and planning is no different. Take a now-ubiquitous 'intelligent' personal assistant agent. One of the pain points helped by an assistant like Siri is scheduling meetings and managing your calendar (Berry et al. 2011). This assistance is based on knowledge of your calendar, to-do list, emails, location – and learned preferences.

If knowledge for computation is the fulcrum, then AI rests on its acquisition. As the KEPS workshop organizers put it, automated planning and scheduling systems "still need to be fed by carefully engineered domain and problem descriptions, and fine tuned for particular domains and problems." This position paper draws attention to learning from human experts as a way to accelerate the knowledge engineering process, which we take to comprise both elicitation and encoding.

We briefly discuss four of the kinds of knowledge required for automated scheduling:

1. **Domain knowledge**. What is the 'physics' of the problem domain?
2. **Problem knowledge**. What are the particulars of the problem instance, including its data and objectives?
3. **Control knowledge**. How does the system go about deciding how to solve the problem instance, and manage the solving process?
4. **Solving knowledge**. What solving approaches and heuristics can be used?

Information science has for decades distinguished between data, information, knowledge, and wisdom (Ackoff 1989). According to Ackoff's oft-quoted taxonomy, starting from the broad base layer of a pyramid and progressing to its narrow pinnacle layer[1], we have:

- Data: raw symbols ('know-nothing' (Zeleny 1987))
- Information: data that is processed to be useful; provides answers to 'who', 'what', 'where' and 'when' questions (know-what)
- Knowledge: application of data and information; answers 'how' questions (know-how)
- Wisdom: evaluated appreciation of 'why' (know-why)

Seen with this lens, the activity of 'knowledge engineering' – such as for an automated scheduling system – aims to apply raw data and processed data in order to support the answering of 'how' questions. For example, the calendaring assistant can (has the know-how to) arrange a meeting with Alice and Bob for next week. While not dwelling on nuances of terminology, we can see domain and problem knowledge as fitting closer to Ackoff's Information level, and control and solving knowledge as fitting closer to his Knowledge level.

We advocate a complementarity of knowledge acquisition by automated techniques and by human knowledge engineers, for the purpose of automated scheduling.

Learning for Domain and Problem Acquisition

The power of automated planning and scheduling systems comes from the combination of the model of the problem and the problem-solving techniques applied to that model. Two elements comprise the former: the model of the domain, and the model of the problem instance. The domain model tells us what is possible, while the problem instance models the questions we want to answer.

As surveyed by Vaquero et al. (2013), real-world problems require detailed knowledge elicitation, encoding and management. These authors' methodology, itSIMPLE, strives to use common notations such as UML in a process of moving from requirements analysis all the way to an input-ready model for solving algorithms.

This kind of methodology, which starts from a graphical representation used to represent statements from subject matter experts (SMEs), is found not only in AI planning and

[1] We follow a number of authors and join Ackoff's Understanding and Wisdom layers.

scheduling, but across other areas of AI such as organizational modelling and agent-based simulation (van Putten et al. 2008) and goal-oriented programming (Abushark et al. 2017).

In line with the rise of machine learning (ML), we can undertake automated acquisition of domain and problem models from data, as Celorrio et al. (2012) survey for AI planning. Since that survey, there has followed much more work on learning domains and problem instances.

Pushing further with learning from data, Lombardi, Milano, and Bartolini (2017) propose a strongly empirical approach to model learning for general combinatorial optimization problems, using ML to construct components of a model from data. The model encompasses both problem domain and instance. The data is obtained either from a relaxed version of the model (a form of boot-strapping), or if applicable and possible, from the modelled system itself. The SME could be involved in creating the initial approximate model; otherwise this approach is driven by data.

A key question is *representation*: how do we formulate the domain and problem models? There are various representations for AI planning, including standard languages such as PDDL, and ML approaches to acquire models into these representations are actively researched.

By contrast to planning, quite commonly scheduling problems have a fixed structure of domain and data, such as flow shop scheduling and other classical Operational Research (OR) scheduling problems. Learning into these representations is more straightforward, and it can suffice to learn from data – since the human expertise has already been put into defining the problem structure. For example, a knowledge engineer encodes the problem as a flow shop with a cyclic job structure, and obtains the data from instrumentation embedded in the manufacturing process. We note that a difficulty, however, given classical OR models is that the modeller can be tempted to coerce the actual problem at hand into one of the standard models, for the sake of convenience, tractability, or the assurance of familiarity.

A more general model for scheduling problems is based on Constraint Satisfaction Problem (CSP): see Salido et al. (2007) for a typical example. ML approaches to acquire (general) CSPs are also actively researched (O'Sullivan 2010; Beldiceanu and Simonis 2016; Bessiere et al. 2017). Bessiere et al. (2017) exemplify this line of work, in deriving CSP models – which like Lombardi, Milano, and Bartolini (2017)'s approach encompass both domain and instance – from a user; both passive and active elicitation are supported.

We advocate for a position that uses data-driven methods as much as possible, and hand-engineered methods in all other aspects. The advantage is to attempt to gain the best from both types of methods: automation and parsimony, and judgement and completeness. In some cases, the two can be used together to triangulate certain knowledge. In other cases, the knowledge acquired with ML can form the starting point for the knowledge engineer's refining of models. In still other cases, manual knowledge engineering can provide or structure data far enough so that ML can then be used.

Learning for Control and Problem Solving

Control knowledge decides what search and reasoning strategies to apply in a problem-solving process, and can adjust the strategies as solving proceeds. Problem solving knowledge comprises of the available strategies, in particular those suited to the domain or to the problem instance. Hence control knowledge is predicated on having problem solving knowledge available to it.

A potent recipe for control knowledge consists of portfolio solving approaches, in which control knowledge is acquired in the form of selection among solving algorithms for a problem instance. Portfolio approaches have proved successful in several subareas of AI, such as SAT (Xu et al. 2008) and automated planning (Gerevini, Saetti, and Vallati 2014). Beck and Freuder (2004) is one example of a portfolio approach specifically for a scheduling problem.

We identify problem solving knowledge for scheduling as 'heuristics'. The literature is substantial on learning how to solve a particular scheduling problem or class of problems (e.g., (Li, Pan, and Mao 2015)). Perhaps because scheduling problems tend to have structure – and at that often a variant of a standard structured problem class, as we have noted – it is easier for scheduling problems than for planning problem to hand-code the problem domain, and extract instance data from some book-keeping system or instrumentation.

Hence the focus of ML for scheduling is drawn to solving problem instances. In a now-classic paper, Gratch, Chien, and DeJong (1993) learn control knowledge for an aerospace scheduling problem; a whole literature on learning metaheuristics is now known. Shahrabi, Adibi, and Mahootchi (2017) is a recent example of learning control knowledge for scheduling, using reinforcement learning. Examples of heuristic learning are many (Russell et al. 2009; Braune and Doerner 2017).

In contrast to this kind of work, which focuses on learning from data, Alzugaray and Sanfeliu (2016) learn path planning heuristics from human expert problem solvers. The point here is that the experts may not be cognisant of their own strategies: they cannot articulate them fully.

Similarly, Berry et al. (2011) learn users' calendering preferences (heuristics) from user actions, with explicit elicitation being optional. These authors found that users' stated preferences often differ from their preferences exposed by their actions. Generalizing, Gombolay et al. (2016a; 2016b) coin the term 'apprenticeship scheduling' for apprenticeship learning techniques applied to scheduling problems. These authors learn heuristics from experts' actions, and in two domains show how an automated scheduling system performs at or exceeds a level that satifices for human approval.

We note that a difficulty of learning from a scheduling problem can arise when the data is gathered under a certain policy (objective, schedule) that we are now trying to optimize. This situation amounts to the classic exploration/exploitation dilemma in reinforcement learning.

We advocate for a position that uses data-driven methods inasmuch as data is available; using learning from experts, particularly to acquire expert knowledge that the SMEs cannot articulate; and using "fine tuned" interventions of knowl-

edge engineers in synergy with the ML techniques. Indeed, it has not passed un-noticed that the engineering of a ML model can be as much effort as manually engineering solving strategies and heuristics (Domingos 2012).

Outlook

We have considered four types of knowledge of automated scheduling, and drawn attention to learning from human experts for these different kinds of knowledge. We note that SME knowledge, and certainly human decision-making, may be imperfect; both (semi-)automated and manual knowledge engineering must recognize this. In advocating a complementarity of knowledge acquisition by automated techniques and by human knowledge engineers, we anticipate a growth in the KEPS sub-community and fruitful interactions with the ML community, including reinforcement learning. Let us take up this opportunity.

Acknowledgements Thanks to the KEPS workshop reviewers for their suggestions. The author also thanks J. Shah, M. Spaan, E. Walraven, M. de Weerdt and C. Witteveen.

References

Abushark, Y.; Miller, T.; Thangarajah, J.; Winikoff, M.; and Harland, J. 2017. Requirements specification via activity diagrams for agent-based systems. *Autonomous Agents and Multi-Agent Systems* 31(3):423–468.

Ackoff, R. 1989. From data to wisdom. *Journal of Applied Systems Analysis* 16:3–9.

Alzugaray, I., and Sanfeliu, A. 2016. Learning the hidden human knowledge of UAV pilots when navigating in a cluttered environment for improving path planning. In *Proc. of IROS'16*, 1589–1594.

Beck, J. C., and Freuder, E. C. 2004. Simple rules for low-knowledge algorithm selection. In *Proc. of CP-AI-OR'04*, 50–64.

Beldiceanu, N., and Simonis, H. 2016. ModelSeeker: Extracting global constraint models from positive examples. In *Data Mining and Constraint Programming – Foundations of a Cross-Disciplinary Approach*, volume 10101 of *Lecture Notes in Computer Science*. Springer. 77–95.

Berry, P. M.; Gervasio, M. T.; Peintner, B.; and Yorke-Smith, N. 2011. PTIME: Personalized assistance for calendaring. *ACM Transactions on Intelligent Systems and Technology* 2(4):40:1–40:22.

Bessiere, C.; Koriche, F.; Lazaar, N.; and O'Sullivan, B. 2017. Constraint acquisition. *Artifical Intelligence* 244:315–342.

Braune, R., and Doerner, K. F. 2017. Real-world flexible resource profile scheduling with multiple criteria: Learning scalarization functions for MIP and heuristic approaches. *Journal of the Operational Research Society* 68(8):952–972.

Celorrio, S. J.; de la Rosa, T.; Fernández, S.; Fernández, F.; and Borrajo, D. 2012. A review of machine learning for automated planning. *Knowledge Engineering Review* 27(4):433–467.

Domingos, P. M. 2012. A few useful things to know about machine learning. *Communications of the ACM* 55(10):78–87.

Gerevini, A.; Saetti, A.; and Vallati, M. 2014. Planning through automatic portfolio configuration: The PbP approach. *Journal of Artificial Intelligence Research* 50:639–696.

Gombolay, M. C.; Jensen, R.; Stigile, J.; Son, S.; and Shah, J. A. 2016a. Apprenticeship scheduling: Learning to schedule from human experts. In *Proc. of IJCAI'16*, 826–833.

Gombolay, M. C.; Yang, X. J.; Hayes, B.; Seo, N.; Liu, Z.; Wadhwania, S.; Yu, T.; Shah, N.; Golen, T.; and Shah, J. A. 2016b. Robotic assistance in coordination of patient care. In *Proc. of RSS'16*.

Gratch, J.; Chien, S. A.; and DeJong, G. 1993. Learning search control knowledge for deep space network scheduling. In *Proc. of ICML'93*, 135–142.

Li, J.; Pan, Q.; and Mao, K. 2015. A discrete teaching-learning-based optimisation algorithm for realistic flowshop rescheduling problems. *Engineering Applications of AI* 37:279–292.

Lombardi, M.; Milano, M.; and Bartolini, A. 2017. Empirical decision model learning. *Artifical Intelligence* 244:343–367.

O'Sullivan, B. 2010. Automated modelling and solving in constraint programming. In *Proc. of AAAI'10*, 1493–1497.

Russell, T.; Malik, A. M.; Chase, M.; and van Beek, P. 2009. Learning heuristics for the superblock instruction scheduling problem. *IEEE Transactions on Knowledge and Data Engineering* 21(10):1489–1502.

Salido, M. A.; Abril, M.; Barber, F.; Ingolotti, L. P.; Tormos, M. P.; and Lova, A. L. 2007. Domain-dependent distributed models for railway scheduling. *Knowledge-Based Systems* 20(2):186–194.

Shahrabi, J.; Adibi, M. A.; and Mahootchi, M. 2017. A reinforcement learning approach to parameter estimation in dynamic job shop scheduling. *Computers & Industrial Engineering* 110:75–82.

van Putten, B.; Dignum, V.; Sierhuis, M.; and Wolfe, S. R. 2008. OperA and Brahms: A symphony? In *Proc. of AAMAS'08 Workshop on Agent-Oriented Software Engineering (AOSE'08)*, 257–271.

Vaquero, T. S.; Silva, J. R.; Tonidandel, F.; and Beck, J. C. 2013. itSIMPLE: Towards an integrated design system for real planning applications. *Knowledge Engineering Review* 28(2):215–230.

Xu, L.; Hutter, F.; Hoos, H. H.; and Leyton-Brown, K. 2008. SATzilla: Portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research* 32:565–606.

Zeleny, M. 1987. Management support systems: Towards integrated knowledge management. *Human Systems Management* 7(1):59–70.

Improving Planning Performance in PDDL+ Domains via Automatic Predicates Reformulation

Santiago Franco, Mauro Vallati, Alan Lindsay, Thomas L. McCluskey
School of Computing and Engineering, University of Huddersfield, UK

Abstract

In the last decade, planning with domains modelled in the hybrid PDDL+ formalism has been gaining significant research interest. PDDL+ models enable the representation of problems that involve both continuous processes and discrete events and actions, which are required in many real-world applications. A number of approaches have been proposed that can handle PDDL+, and their exploitation fostered the use of planning in complex scenarios.

In this paper we introduce a PDDL+ reformulation method that reduces the size of the grounded problem, by reducing the arity of *sparse* predicates, i.e. predicates with a very large number of possible groundings, out of which very few are actually exploited in the planning problems. Arity is reduced by merging suitable objects together, and partially grounding the operators, processes and events in which reformulated predicates are involved. Our experiments show that these methods can substantially improve performance of domain-independent planners on PDDL+ domains.

Introduction

The growing number of domain-independent PDDL+ planners is fostering the exploitation of planning in complex real-world applications, where notions of continuous processes and discrete events and actions are needed (Fox and Long 2006). Since they accept the domain and problem description in a standardized interface language and return plans using the same syntax, they can now be exploited as embedded components within larger frameworks, as they can be interchanged without modifying the rest of the system.

This modular approach supports the use of reformulation and configuration techniques, which can automatically re-represent the planning model in order to increase efficiency and enable a scale up in size of applications that can be handled. Types of reformulation of PDDL models include macro-learning (Botea et al. 2005; Newton et al. 2007), bagged representation (Riddle et al. 2015), action schema splitting (Areces et al. 2014) and entanglements (Chrpa and McCluskey 2012; Chrpa, Scala, and Vallati 2015): here the domain model is transformed to a more efficient form that is fed into the planner. Recent work (Vallati et al. 2015) also highlights the importance of domain model configuration.

Hybrid PDDL+ models are amongst the most advanced models of systems and the resulting problems are notoriously difficult for planners to cope with due to non-linear behaviours and immense search spaces. Complexity is exacerbated by the potentially huge size of the fully grounded problems, which can make some problems impossible to tackle. Particularly, grounding is also strongly affected by predicates' instances that will not be reachable in any state of the problem.

In this paper we introduce a PDDL+ reformulation method that allows to drastically reduce the size of the grounded problem, by reducing the arity of *sparse* predicates, i.e. predicates with a very large number of possible groundings, out of which very few are actually exploited in the planning problems. Arity is reduced by merging suitable objects together, and partially grounding the operators, processes and events in which reformulated predicates are involved. In a broader sense, our work is inspired by the general use of dimensionality reduction to make processes more efficient in AI. For example, in Machine Learning, dimensionality reduction has been to reduce the size of hypothesis space (Srinivasan and Kothari 2005). Our experimental analysis shows that the proposed reformulation technique can substantially improve the performance of PDDL+ planning engines, by allowing problems to be grounded and by constraining the search space.

The Proposed Reformulation Approach

In this section we describe the proposed reformulation step that collapses variables within sparsely instantiated predicates. Our approach relies on identifying sparse predicates that are partially constrained by a static predicate. Through combining the sparsity measure for dynamic predicates with a constraining static predicate, the approach is able to better identify predicates for which the reformulation will have significant impact. We first provide a motivating example from the Rovers domain and then present our algorithm.

A Motivating Example from the Rovers Domain

Let us consider an hybrid version of the well-known Rovers domain model, where movements and energy generation via solar are modelled as continuous processes, triggered by actions under the control of a planner, and constrained by appropriate events. In the Rovers domain, rovers are used to make soil and rock samples and to take pictures for various objectives. This requires that the rovers are moved between waypoints in order to establish shots and collect samples

Algorithm 1 Reformulation for flattening sparse predicates

Input: D_o, I_o, s^t, a^t
Output: D_r,I_r

1: $SP = statics(D_o); P = predicates(D_o) \cup functions(D_o)$
2: $D_r = D_o; I_r = I_o$
3: **for all** p_j **in** P, **where** $arity(p_j) > 2$ **do**
4: **if** $sparsity(p_j, I_o) > s^t$ **then**
5: $p_{stat} = findConstrainingStatic(p_j, SP)$
6: **if** $p_{stat} \neq None$ **then**
7: $T_{p_{stat}} = getSparseVariables(p_{stat}, I_o, a^t)$
8: $C^{new} = makeConstants(T_{p_{stat}}, s_o)$
9: $D_r = addAsConstants(D_r, C^{new})$
10: $D_r = updateOpProEv(D_r, T_{p_{stat}}, C^{new})$
11: $I_r = updatePredsFuncs(D_r, I_r, T_{p_{stat}}, C^{new})$
12: **end if**
13: **end if**
14: **end for**

Predicate	**CP**	$s_0 \wedge$ **CP**	Sparsity
can_traverse	32	10	0.3125
have_image	18	0	0.0
energy	2	2	1.0
recharges	1	1	1.0
rover-movement	2	2	1.0
movementduration	1	1	1.0

Table 1: The size of the cross product (**CP**), initial state ($s_0 \wedge$ **CP**) and sparsity score for the predicates or numeric predicates with arity > 2 for a Rovers problem.

from certain positions. The constraints establishing the properties of the rovers and the relationships between waypoints (e.g., that a rover can traverse between waypoints) are encoded as static facts. As with many network based relationships, only a fraction of the potential connections are made available in any particular problem model. Of course, as the number of waypoints (and rovers) grows this fraction will reduce. We have observed that sparsely instantiated predicates like these can lead to poor performance in PDDL+ planners. The problem model reformulation reported in this paper collapses the variables of predicates, creating a model with fewer sparsely instantiated predicates. This procedure can be applied to Rovers, for example, consider replacing the (arity 2) predicate: (`visible` *?waypoint1 ?waypoint2*) with an alternative (arity 1) encoding: (`visible` *?visible_waypoint1_waypoint2*). Whereas in the original version, the domain of possible instantiations is every pair of waypoints; in the second approach, we can reduce the domain by only defining constants for the combinations of waypoints for which the relation holds.

The Reformulation Algorithm

Algorithm 1 shows how the reformulation of a domain model, D_o, and a problem model, I_o, is performed. Beside the models to be reformulated, the algorithm requires as input a sparsity threshold s^t, which is used to decide whether or not it is useful to perform the reformulation and a parameter, and a^t, which sets the maximum number of variables considered in the reformulation (how these parameters are set is explained below).

In the algorithm (see Algorithm 1) the sparsity of the predicates (boolean or numeric) with arity greater than 2 are assessed in turn (line 3) to determine if they are suitable for the reformulation step. As a measure of sparsity we compare the set of propositions in the initial state with the possible set of all propositions for the predicate. For example, if we consider a specific example Rovers problem from our benchmark problems, with 4 waypoints and 2 rovers, we can calculate the total set of possible propositions as: $4 \times 4 \times 2 = 32$. In this example, there are 10 instances of `can_traverse` in the initial state and so the sparsity for this predicate is $10/32$. Table 1, presents the sparsity values for `can_traverse` and the other predicates with a higher arity than 2 for the same example Rovers problem.

In the case of a sparse predicate, p_j, the procedure attempts (line 5) to find a static predicate, p_{stat}, such that p_j is only used in transition schemas (that is in the action, process or event schemas) with p_{stat}. We consider predicates as static if instances of the predicate can not be deleted or created during the planning process but, in the case of numeric predicates, their value can be changed. If there is more than one constraining static predicate then one is selected heuristically by selecting the predicate that occurs the most in transition schemas. There are two static facts that constrain the `can_traverse` predicate: `can_traverse` itself and `visible`. The algorithm selects `visible` as it appears in more transition schemas.[1]

Reformulating the domain model

In the case that p_{stat} exists (e.g., `visible`), a reformulation step is applied using p_{stat} as the basis. In our current system, we have considered subsets of the variables of the static facts and so we add a parameter, a^t, to determine the maximum arity of the reformulation. The best $\texttt{max}(a^t)$ variables are selected (line 7) using the sparsity of the tuple for p_{stat} in Io. We use T_p to denote a subset of the variables of p. In our example, `visible` has arity 2 and therefore $T_{\texttt{visible}}$ would contain both its variables. The variables in T_{stat} are then combined to form a set of constants, C^{new}, of type, t^{new}, which are added to the domain model. One constant is defined for each distinct combination of these variables for the instances of the predicate in I_o. For example, a new constant is generated for each distinct combination of the waypoints in the instances of the `visible` predicate in the initial state. For instance, `(visible waypoint3 waypoint0)` leads to a new constant `waypoint3_waypoint0` (using the new type).

At this stage (line 10) each of the transition schemas that refer to p_{stat} are reformulated. For example, in Rovers, the transitions with `visible` as a precondition are identified (e.g., `start-navigate`, `communicate_soil_data`). For each predicate (dynamic, static or numeric) in these transitions the algorithm tests to determine if it can be part of this reformulation step. If the predicate is only used in transition schemas with p_{stat}, and $T_{p_{stat}}$ is a subset of the parameters of the predicate then it is selected for reformulation. In the case of `visible`, only the `can_traverse`

[1]Notice that although the `have_image` predicate is sparsely defined in the initial state (see Table 1), there is no constraining static predicate, hence it does not satisfy the condition in line 6.

predicate is constrained by the `visible` predicate and so only these two are selected for reformulation. For each selected predicate, p, (including p_{stat}) a new predicate, p', is made by replacing the variables that are in $T_{p_{stat}}$ with a single variable of type, t^{new} and retaining the other variables (e.g., $arity(p') = arity(p) - |T_{p_{stat}}| + 1$). For example, (`can_traverse` *?rover ?waypoint1 ?waypoint2*), is reformulated as (`can_traverse` *?rover ?new-type*). The original predicate, p, is omitted from the new model, Dr.

Each transition schema that depends on p_{stat} is partially grounded so that the variables corresponding to those in $T_{p_{stat}}$ are grounded and constants added as necessary (i.e., for referencing the individual objects). This allows the relation between the new constants and the original objects to be maintained. For example, there are new `start-navigate` operators for each of the new constants, e.g., `start-navigate-waypoint3-waypoint2`. In `start-navigate`, each matching of *?waypoint* is added as constant as it is referred to in other predicates.

Reformulating the problem model

Finally, the problem model is reformulated (line 11) by changing those predicates involved in the reformulation to use the constants in C^{new} in the initial state and goal, using a similar approach as described above. For example, the proposition:

```
(can_traverse rover0 waypoint3 waypoint0)
```

is reformulated as :

```
(can_traverse rover0 waypoint3_waypoint0).
```

Of course, after this step has been applied once, the procedure can be repeated on the reformulated model supporting further combining of variables as appropriate. In cases where action parameter lists have been modified, a simple post-process can be used to resolve plan steps correctly.

Experimental Analysis

Our experimental analysis aims at assessing the usefulness of the proposed reformulation approach in improving the performance of domain-independent planners able to handle PDDL+ hybrid domains.

Four PDDL+ planners at the state of the art are included in the evaluation: ENHSP (Ramírez et al. 2017; Scala et al. 2016), UPMurphi (Della Penna et al. 2009), DINO (Piotrowski et al. 2016), and SMTPlan (Cashmore et al. 2016). Planners have been run using default heuristics, unless differently specified.

For a fair comparison, all reported results were achieved by running the planners on a machine equipped with i7-4750HQ CPU, 16 GB of memory, running Ubuntu 16.10 OS. 4 GB of memory were made available for each run, and a 15 CPU-time minutes cut-off time limit was enforced.

We observed that traditional PDDL+ benchmarks usually include a very limited number of objects and restricted number of predicates and operators / processes / events. For this reason, the experimental evaluation is performed by considering three benchmark domains, namely Hybrid Rover, Urban Traffic Control, and Baxter, –which have been recently introduced or have been designed on the basis of existing domains– and five instances per domain.[2] The domains have been selected for their complexity, in terms of involved objects and predicates, and due to the fact that they represent interesting applications of planning in real-world scenarios. For the sake of this analysis, here we considered for reformulation the predicates with the largest arity in each domain.

Baxter. The Baxter domain, recently introduced in (Capitanelli et al. 2017), exploits planning for dealing with articulated objects manipulation tasks. The available "simplified" domain model[3] has been extended by allowing continuous movements of a joint, modelled via actions and process envelope, on different axis, and by adding events for preventing movements wider than 360 degrees. Problems consider articulated objects composed by between 2 and 5 links. In this model, the objects of type `link` has been merged into a new type, and four predicates have then been reformulated: `connected`, `increasing_angle`, `decreasing_angle`, and `use`. Our reformulation approach has been applied following the fact that the `connected` predicate is static, and is exploited in operators and processes to control all the other mentioned predicates.

According to the results shown in Table 2 UPMurphi, DINO, and SMTPlan grounded and explored the search space of all the considered problems but only ENHSP solved most of the problems using the original representation. This domain gives a high degree of freedom to the planner to decide how to manipulate the articulated object. ENHSP solved all but one of the problems using the original models. Remarkably, the use of reformulated models did lead to a significant search speedup, and allows ENHSP to solve all the considered benchmarks. Empirical evidence indicates that the reformulation allows to improve the pruning done by the reachability analysis of ENHSP, leading to a faster expansion and evaluation of the search states. Interestingly, DINO works better with the original representation. According to our observations, in that specific problem, the DINO heuristic expanded twice the number of states compared to its use with the reformulated model, but to find the same solution.

Hybrid Rover. We extended the well-known Rover domain model introduced in IPC-3 (Long and Fox 2003) by modelling as continuous processes the movements of the rovers, and the energy generation via solar power. Each of the mentioned process can be controlled by the planner using two appropriate actions (one for starting the process, and one for terminating it), and is constrained, where appropriate, via events. In this domain, the predicate `can_traverse` has been reformulated by merging the objects of type `waypoints`, as shown in the previous section.

The use of reformulated models allows ENHSP to solve a larger number of problem instances. However, in few cases, the reformulation negatively affected search performance, once the grounding is completed.

SMTPlan is not significantly affected by the different domain models. This seems to be related to the compilation of the PDDL+ model into an SMT encoding that allows to reduce grounding-related issues.

[2] Benchmarks will be made available in the CRC, via link.

[3] `https://github.com/EmaroLab/paco_synthetic_test`

Planner		Baxter					Hybrid Rover					UTC				
		1	2	3	4	5	1	2	3	4	5	1	2	3	4	5
ENHSP	O	0.40	X	26.8	13.5	335.7	1.3	18.9	58.54	77.33	–	5.3	10.8	–	–	–
	R	0.45	151.7	23.5	15.2	17.9	1.1	35.0	60.28	65.5	87.5	2.7	3.2	12.5	6.9	30.8
UPMurphi	O	X	X	X	X	X	X	X	X	X	X	7.26	X	–	148.78	X
	R	X	X	X	X	X	X	X	X	X	X	0.62	5.02	29.34	5.2	49.4
DINO	O	12.0	X	X	X	X	116.24	X	X	X	X	X	X	–	X	X
	R	27.6	X	X	X	X	X	X	X	X	X	X	X	X	X	X
SMTPlan	O	0.01	X	X	X	X	0.5	1.8	X	X	X	NA	NA	NA	NA	NA
	R	0.01	X	X	X	X	0.5	1.8	X	X	X	NA	NA	NA	NA	NA

Table 2: For each considered problem, the CPU-time seconds needed by the planners to find a satisficing solution. O (R) rows show the results achieved when running the Original (Reformulated) model. X indicates grounded but not solved. ”–” means crashed during grounding. NA indicates that the planner is not able to handle the domain model.

	Baxter					Hybrid Rover					UTC				
Problem	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5
Ratio	2.42	3.98	3.98	3.13	4.76	1.00	1.05	1.88	2.86	4.18	18.59	37.08	–	18.58	21.05

Table 3: Ratio of maximum search space sizes of original vs reformulated representations for the UPMurphi and DINO planners. ”–” is used to indicate cases where one of the approaches lead planners to crash during grounding.

Urban Traffic Control (UTC). This domain has been introduced in (McCluskey and Vallati 2017). It models the use of planning for generating traffic light signal plans, in order to de-congest an area of a urban region. In this analysis we considered the problems introduced by McCluskey and Vallati, which involved a network of 10 junctions, and we extended it by considering problems with 20 and 30 junctions, obtained by connecting identical regions. Problems 1–3 have 10, 20 and 30 junctions respectively and only one goal. Problems 4 and 5 have 10 and 20 junctions respectively, both of them have 3 goals.

In this domain, the predicate `flowrate` has been reformulated by merging the objects of type `link` into a new type, which represents road links which are connected via a junction. In this domain, ENHSP and UPMurphi were run using the heuristic proposed by McCluskey and Vallati.

Results presented in Table 2 indicate that reformulation has a strong beneficial impact on planners' performance. ENHSP and UPMurphi are able to quickly solve problems involving large networks as they can manage to ground the problem. In ENHSP most of the improvement is due to the faster grounding, and on the largest 30 junction problem 3, to be able to ground it at all. On the contrary, in the case of UPMurphi and DINO, the reformulation boosts also the search performance, as –given the data structure used by those planners– also the size of each state is significantly reduced by the reformulation. Unsurprisingly, the heuristic exploited by DINO is not very informative in UTC problems, since it is domain-independent, but the planner is able to ground and to explore a large area of the search space when run on reformulated models.

Finally, Table 3 shows how planners DINO and UPMurphi benefit from the reformulation of PDDL+ models, in terms of state size. Results are presented in terms of ratio of maximum space sizes between original and the corresponding reformulated representation; for instance, a value of 2.0 means reformulated search can create twice the number of states before running out of memory. In almost every instance, reformulation greatly increases the maximum state space. An outstanding ratio has been achieved in the UTC domain, where for problem 2 the reformulation allows to raise the maximum number of states by a factor of 37. We expect the ratio for problem 3 to be even better as it has an even larger number of junctions, unfortunately a direct comparison can not be made because the original representation is too large for UPMurphi/DINO parsers to process.

Conclusion

Despite the importance of hybrid planning for real-world applications, there is a lack of knowledge engineering techniques for PDDL+ models, that would allow to improve the performance of any domain-independent planner on the problems of interest.

In this paper, we introduced a reformulation approach that allows to reduce the size of a PDDL+ grounded problem by tackling the arity of sparse predicates. Our experimental analysis showed that: (i) the proposed reformulation technique can effectively reduce the grounding size of hybrid problems, hence allowing planners to deal with them; (ii) the reformulation can also positively affect the size of each search state, leading to a faster and more effective exploration of the search space; and (iii) merged objects can positively affect heuristics. As an example of the importance of the proposed technique, in the UTC domain, the reformulation enables two planners (ENHSP and UPMurphi) to produce strategies in urban regions with 30 junctions (containing over 120 road links) which gives it a clear advantage in scale over conventional connected road traffic management systems such as SCOOT (Taale, Fransen, and Dibbits 1998). This suggests that PDDL+ reformulation techniques, by allowing larger problems to be reasoned upon by planning engines, can lead to a new class of heuristics and foster the exploitation of planning in real-world applications. Future work will be focused on extending the number of objects that can be merged, and an in-depth study of the importance of the sparsity threshold.

References

Areces, C.; Bustos, F.; Dominguez, M.; and Hoffmann, J. 2014. Optimizing planning domains by automatic action schema splitting. In *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling, ICAPS*, 11–19.

Botea, A.; Enzenberger, M.; Müller, M.; and Schaeffer, J. 2005. Macro-FF: improving AI planning with automatically learned macro-operators. *Journal of Artificial Intelligence Research* 24:581–621.

Capitanelli, A.; Maratea, M.; Mastrogiovanni, F.; and Vallati, M. 2017. Automated planning techniques for robot manipulation tasks involving articulated objects. In *Proceedings of the XVIth International Conference of the Italian Association for Artificial Intelligence*, 483–497.

Cashmore, M.; Fox, M.; Long, D.; and Magazzeni, D. 2016. A compilation of the full PDDL+ language into SMT. In *Proceedings of the Twenty-Sixth International Conference on Automated Planning and Scheduling, ICAPS*, 79–87.

Chrpa, L., and McCluskey, T. L. 2012. On exploiting structures of classical planning problems: Generalizing entanglements. In *Proceedings of the European Conference on Artificial Intelligence ECAI*, 240–245.

Chrpa, L.; Scala, E.; and Vallati, M. 2015. Towards a reformulation based approach for efficient numeric planning: Numeric outer entanglements. In *Proceedings of the Eighth Annual Symposium on Combinatorial Search, SOCS*, 166–170.

Della Penna, G.; Magazzeni, D.; Mercorio, F.; and Intrigila, B. 2009. UPMurphi: A tool for universal planning on PDDL+ problems. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling, ICAPS*, 106–113.

Fox, M., and Long, D. 2006. Modelling mixed discrete-continuous domains for planning. *Journal of Artificial Intelligence Research* 27:235–297.

Long, D., and Fox, M. 2003. The 3rd international planning competition: Results and analysis. *Journal of Artificial Intelligence Research* 20:1–59.

McCluskey, T. L., and Vallati, M. 2017. Embedding automated planning within urban traffic management operations. In *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling, ICAPS*, 391–399.

Newton, M. A. H.; Levine, J.; Fox, M.; and Long, D. 2007. Learning macro-actions for arbitrary planners and domains. In *Proceedings of the International Conference on Automated Planning and Scheduling, ICAPS*, 256–263.

Piotrowski, W. M.; Fox, M.; Long, D.; Magazzeni, D.; and Mercorio, F. 2016. Heuristic planning for PDDL+ domains. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI*, 3213–3219.

Ramírez, M.; Scala, E.; Haslum, P.; and Thiébaux, S. 2017. Numerical integration and dynamic discretization in heuristic search planning over hybrid domains. *CoRR* abs/1703.04232.

Riddle, P. J.; Barley, M. W.; Franco, S.; and Douglas, J. 2015. Automated transformation of PDDL representations. In *Proceedings of the Eighth Annual Symposium on Combinatorial Search, SOCS*, 214–215.

Scala, E.; Haslum, P.; Thiébaux, S.; and Ramírez, M. 2016. Interval-based relaxation for general numeric planning. In *ECAI – 22nd European Conference on Artificial Intelligence*, 655–663.

Srinivasan, A., and Kothari, R. 2005. A study of applying dimensionality reduction to restrict the size of a hypothesis space. In Kramer, S., and Pfahringer, B., eds., *Inductive Logic Programming: 15th International Conference, ILP*, 348–365.

Taale, H.; Fransen, W.; and Dibbits, J. 1998. The second assessment of the SCOOT system in Nijmegen. In *IEEE Road Transport Information and Control*, number 21-23.

Vallati, M.; Hutter, F.; Chrpa, L.; and McCluskey, T. L. 2015. On the effective configuration of planning domain models. In *Proceedings of the Twenty-Fourth international Joint Conference on Artificial Intelligence, IJCAI*, 1704–1711.

Domain Model Analysis using Static Graphs

Rabia Jilani
School of Computing and Engineering
University of Huddersfield
United Kingdom

Abstract

Automated Planning is a systematic search for a range of actions to reach some desired goals. Automated Planners must have a model of the dynamics of the domain to reason with. Domain Models can be encoded by human experts or, as required by autonomous systems, automatically acquired from observation. Domain model correctness is a crucial factor in the overall quality of the planning process. This leads to the increased need to measure and analyze domain models, in particular, to evaluate Knowledge Representation (KR) and Knowledge Engineering (KE) techniques.

In this paper, we present a static domain analysis method by the combined use of our previously published work on static graphs learning systems ASCoL with the system of Wickler to understand domain dynamics. A fundamental difference between the two systems is that the Wickler's system depends on the manual extraction of domain features, whereas ASCoL can extract automatically. The resulting technique builds on the static domain analysis of automated planning domain models in terms of automatic identification of static graph relations, Extended Static Relations (ESRs) and Static Modifier Operators (O_{SM}). The method has been evaluated using 13 planning domains drawn from the international planning competition.

Introduction

In 2003 PLANET Roadmap (Biundo et al. 2003), the authors generally describe the steps for domain model development that covers the complete definition of Knowledge Engineering for Planning and Scheduling (KEPS). According to the report, the following are general steps within the domain modelling process:

1. Early Analysis
2. Knowledge Acquisition
3. Domain Design
4. Domain Validation
5. Maintenance

Although AI planning technology is mature, the research in the KEPS community mostly relies on the use of experimental domains for bench-marking the KE experiments. This is also because KE has no standard evaluation and analysis methods, and just like the requirements specification, it cannot be objectively assessed and proved correct.

The evaluation of KE tools and methodologies is challenging when compared to other tools and methods, such as planning algorithms and planners. Moreover, knowledge engineers of domain models use planners to design, develop and debug the domain model, and the planners were not primarily developed for this purpose. This problem of methods/tools validation and verification (V&V) in KE inclines the research community to avoid paying much attention to knowledge engineering for AI planning and it becomes harder to foster the encoding of sophisticated domain models. Indeed, Bensalem et al (Bensalem, Havelund, and Orlandini 2014) state that domain models present the biggest V&V challenge to the Planning and Scheduling (P&S) community. Moreover, for real-time applications of P&S inside critical area e.g. space missions, the domains turn out to be very complex and it takes hard work for domain engineers to validate such domain models. The source of complexity in the planning descriptions arises from the highly declarative form they take and also the complex interactions between the behaviours of different component subsystems within a domain (Long, Fox, and Howey 2009). Such complex domains express the need for automatic analysis, evaluation, V&V techniques to be integrated as parts of KE and P&S systems.

Despite the efforts in the VVPS workshops in ICPAS 2009 and 2011, there are few tools that strive to provide some kind of validation analysis of domain models. Some KE tools such as GIPO (Simpson, Kitchin, and McCluskey 2007) and itSIMPLE (Vaquero et al. 2007) allows users to develop and validate domain model design. (Giunchiglia and Traverso 2000) presents Planning as Model Checking paradigm, a verification system for a domain by exploring its state space. Model Checking has been utilised in the validation of domain models more safety-critical environments such as in Remote Agent Experiment (Khatib, Muscettola, and Havelund 2001). VAL (Howey, Long, and Fox 2004) is another well-developed tool for PDDL domain encoding and provides insight into the behaviour of a domain encoding by evaluating plans. Some authors (Shoeeb 2012)(González Ferrer 2012) have also proposed varying factors that identify the quality of a domain model from a KR and KE point-of-view. Regardless of such efforts of researchers, the scope of domain validation remains limited.

(Biundo et al. 2003) classify process of domain model validation in the process of promoting domain quality in

terms of internal and external criteria by the identification and removal of errors in the model. Internal criteria include properties such as syntactic correctness and logical consistency. External criteria include properties such as accuracy, correctness and completeness.

Our work concentrates on the fourth stage of domain modelling process i.e. Domain validation in terms of static analysis. More specifically to check logical consistency, identify bugs and to check the accuracy of the static domain model structure. As the sources of knowledge elicitation (i.e. plan traces in our case) and domain model development are not mathematical procedures, it is challenging for them to be measured on a correctness scale. Its quality must be checked not only in terms of syntactic and semantics' accuracy but it also includes checking the completeness of the generated domain through static and dynamic testing (Vaquero, Silva, and Beck 2011). One aim of this piece of publication is to stress the importance and need of KE validation tools. We have extended our work on previously published static constraints learning system to use it alternatively as static domain analysis system along with another published analysis technique. The overall aim is to automate and simplify the availability and use of domain analysis systems by domain designers and research community for easy, elaborated and error-free design. The research contributes a visualization tool to help the modeller observe, analyse and validate the static constraints and features of the domain model as a whole.

We use ASCoL (Jilani et al. 2015) -a general system that learns static constraints from previous experience. The system supports domain-independent planning and knowledge engineering in the modelling phase of planning domains. It learns domain-specific static constraints or invariants for domain models in the form of a static graph of ordered static rules (positive preconditions). It demonstrates the feasibility of automatically identifying static relations for domain models by considering application knowledge in the form of training plans for a range of application areas. ASCoL exploits the information which is implicit in the log of action sequences (also called plan traces). It exploits graph analysis for automatically identifying static relations and static graphs from example training plans, in order to enhance planning domain models. ASCoL has been evaluated on domain models from the international planning competition benchmarks, using differently-shaped plan traces.

Wickler's method (Wickler 2013) of domain model analysis uses static graphs of a planning domain. We use this method to identify objects or properties related to the nodes of the static graph that may be modified by the operators in ways restricted by the graph. The formal definition of *shift operators* over static graphs as a domain feature is the main contribution of Wickler's method.

The remainder of this paper is organised as follows. Section 2 includes preliminary definitions and the necessary background. Section 3 describes the ASCoL + Wickler 2013 method, with the *Freecell* and *Grid* domain (from IPC) operators as our running examples. Section 4 includes discussion and results of the empirical evaluation. Finally, Section 5 discusses some applications and concludes the paper.

Preliminaries

In this section, we provide the necessary background and define the key terms that are proposed and exploited by both ASCoL and Wickler's system.

Graphs

A graph is an abstract representation of data in the form of a set of vertices or nodes that are connected either by directed or undirected edges and is used to model multiple types of relationships between objects.

Graph: $G = (V, E, \mu, v)$

- V: finite set of nodes.
- $E \subseteq V \times V$ denotes a set of edges.
- μ: $V \rightarrow L_V$ denotes a node labelling function
- v: $E \rightarrow L_E$ denotes an edge labelling function

Graphs are a popular way to formulate representations and provide a visual way for easily explaining to the user the varying type and complexity problems, including mathematical and computational problems, as well as reasoning about action. Due to the visual representations of complex interrelationships between entities graphs enable the human brain to hold facts and records for longer as compared to the logical representation of the same data.

Static constraints or relations

Domain classes and objects inside them interrelate in different ways. Some interactions are dynamic and others are static. These interrelations are modelled as relationships. Static relationships or sometimes called invariants are implicit in the set of operators and are not directly expressed in the plans. Static relationships enforce constraints in dynamic relationships and restrict the traversal of unnecessary reachable states in solution search. Without using static constraints planners are slower because they have wider search space options to search for a solution to a problem and most of the times produce invalid plans. These can be seen as predicates that appear in the preconditions of operators only, and not in the effects. Therefore, the truth value of static facts and objects never change in the world from state to state under closed world assumption, and that is why they do not appear in state transitions and cannot be relocated. Many domains contain static background information that remains unchanged with the given (problem) task changes, for example, in transport domains the map of an area is the static fact whereas, the routes of a traveller might change depending on the purchases she needs to carry out each day. Other examples of static constraints include the connections of roads in the logistics domain, the level of floors in the Miconic domain and the fixed stacking relationships between specific cards in the Freecell domain.

In the ontology, domain invariants are inter-relationship constraints. In particular, they constrain instance relationships. Since instance relationships are the elements in modeling object states, domain invariants enforce constraints in state-space.

Let $Op = op_1, op_2, ..., op_n$ be a set of operators and let $P = P_1, P_2..., P_n$ is a set of all the predicate symbols that occur in these operators. A predicate P_i in P is fluent iff there is an operator op_j in Op that has an effect that changes the value of the predicate P_i. Otherwise, the predicate is static. We call the static predicate, Main Static Relation (MSR) of the operator.

Gregory et al in (Gregory and Cresswell 2015) define static relations as restrictions on groundings of operators in the domains. In other words, a static relation for each operator is a collection of all the valid groundings for that operator.

Static graph relations Wickler in (Wickler 2013) defines a static graph relation as follows:
If $a(P_i)$ are the arities of predicates P_i, then P_i is a static graph relation if and only if:

- P_i is a Main Static Relation (MSR);
- P_i is a binary relation i.e. $a(P_i)$ = 2; and
- The two arguments of P_i are of the same type T = $argP_i(1) = argP_i(2)$.

Static Graph

ASCoL identifies static graphs and MSRs automatically from plan traces $Pl = (pl_1, pl_2, ..., pl_n)$. Each plan contains an action sequence of N actions on numerous objects, i.e. each $Pl_i \in Pl$ has the form:

$$pl_i(a_1, a_2, ..., a_m) \quad \text{for} \quad i = 1, ..., n$$

Where a_i is an action of the plan trace pl_i. Each action has a format which is made up of an identifier (the name of the action), and the names of objects that it affects, in order of occurrence, which all have the form:

$$a_i(o_{i1}, ..., o_{ij})$$

Where o_{i1} represents the object provided as first parameter for the action a_i.

In ASCoL, all kinds of static relations are represented as graphs, where the graph vocabulary can be seen as the object instances of a particular type t which can be taken from a live activity record(s) or from a manually recorded log of an action sequence. It parses the input plan sequences to induce the universe of same-typed object instances from plan traces. The identified same-typed object instances make the elements of the graph vocabulary. It then identifies all the possible combination pairs of same-type objects in each unique action. The formula to calculate the total number of parameter pairs for each action header is:

$$(n-1) + (n-2) + ... + (n-n)$$

Where n represents the total number of arguments in an action that belongs to same type *t*. For instance, if in an action, the number of available same-typed arguments is four then the total number of parameter pairs would be $3 + 2 + 1 + 0 = 6$. The next step is to generate the embedding and analysis of graphs *G = (IDs, Conn, μ, v)* by considering all the pairs involved in the matching actions from the complete input set of plans (Pl).

Graph: *G = (IDs, Conn, μ, v)*

- IDs: $\{o_{ij} \in O | t(O) = t_i\}$ is a finite set of nodes
- Conn $\subseteq$ IDs $\times$ IDs denotes a set of edges
- μ: IDs $\rightarrow L_{IDs}$ denotes a node labelling function
- v: Conn$\rightarrow L_{Conn}$ denotes an edge labelling function

IDs is the set of vertices of the graph *G* which are labelled by elements in the vocabulary and are observed from plan traces in the form of object instances or action parameters. *O* is the set of object instances from all matching action instances in plan traces and *t(O)* is the type of the object instances *O*. *Conn* is the finite set of edges for each of a particular pair of action instances across all the plan traces available.

Extended Static Relations (ESRs)

Given an automatically extracted MSRs, there could be other relations in the operator that connect the static feature to other objects of different types and which have a fixed relation to the MSR in the form of a graph too. We call such relations Extended Static Relations (ESRs). The object type due to which ESR is associated with MSR, Wickler refers to this object type as a Node-Fixed Type.

Let $Op = op_1, op_2, ..., op_n$ be a set of domain operators and P_i be the MSR learnt from the static graph, $G_{Pi} = (IDs, Conn, \mu, v)$ consisting of nodes called *IDs* and the finite set of directed edges called *Conn* where μ is a node labelling function and *v* is an edge labelling function. t_i is the object type of P_i. A type $t_j \neq t_i$ is called a node-fixed type if the following conditions hold:

- there exists a static binary relation P_j ; and
- P_j has one argument of type t_i and the other of type t_j .

ESR P_j is the relation/predicate that is not dynamic but its arguments are of two different types including the overlapping type (t_i) with MSR P_i. These relations identify the functional properties of the defining static relation, e.g. a value, cost, colour or physical location of the object. The objects in ESRs cannot move between nodes in the static graph and have some fixed relationship with nodes in the static graph. This can be stated as Whole-part relationship or aggregation as mentioned by (Biundo et al. 2003).

Each ESR can further be extended if it fulfils the above-mentioned conditions for producing further extension in the static graph by considering ESR's node-fixed static type t_j in place of MSR's static type t_i. We call such relations as *Level-Two ESRs (ESR_{L2})*

Shift Operators or Static Modifier ($\mathbf{O}_{SM}$)

Given a domain model that exhibits static aspects in addition to dynamic behaviour in operator definition, there are often dynamic first-order relations that support the transitions of object states by representing the dynamicity or movement of static graph objects in the dynamic domain scenario. Such

first-order dynamic predicates carry different functional properties with the same semantics and define the relationship between static and dynamic aspects of objects in the pre- and post-execution of that action. Wickler calls operators that encompass such a property, *Shift Operators* and defines them as follows:

If O be a particular planning operator having preconditions $P_1, ..., P_n$, positive effects+ $(e_1^p, ..., e_n^p)$ and negative effects- $(e_1^n, ..., e_n^n)$ where each precondition and positive/negative effect is a first-order atom, and

If P_i = static graph relation for the O,
t_i = static type for P_i
t_j = node-fixed type for P_i.

Then, O is a shift operator wrt t_j iff:

- O has a MSR $P_i(v, v')$;
- O has a precondition p_s^p with argument v (or v');
- O has a negative effect- = p_s^p;
- O has a positive effect+ = p_s^p but the argument v (or v') must alternate with v'(or v), respectively.

We altered the definition by also including unary predicates with an only static object of type t_i (and no necessity of t_j as a second argument), in addition to binary predicates with reference to t_j. Because only few domains exist that contain this shifting property with respect to t_j, we extended the definition in order to bring a larger range of domains in the application focus of this analysis in addition to transport domains. We call such operators Static Modifiers (O_{SM}). Biundo et al. in (Biundo et al. 2003) refer to such relationships as cardinality relationships.

Figure 1 shows the pseudo code for shift operator identification presented by Wickler. The pseudo code attempts to find preconditions and effects that satisfy all the conditions. It accepts an operator and an MSR as inputs. As output, it indicates if the input operator is Shift Operator or not wrt. the argument type of the given predicate. The algorithm loops through all the precondition to finding one that represents an edge in the graph. Then it checks the preconditions again to find one that represents a candidate for a shifted property. A necessary condition here is that the node from which we are shifting occurs in the property precondition. Given such a candidate, the algorithm tests whether the property is deleted by the operator, which is another necessary condition. Finally, the algorithm tests whether a positive effect exists that represents the shifted property, which is true if it agrees with the property precondition in all arguments except for the one representing the static graph node, which must be the node to which we are shifting to the effect.

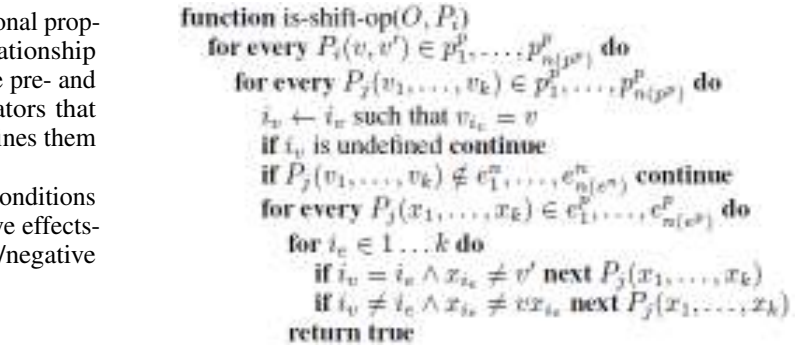


Figure 1: Pseudo Code for Shift Operators.

ASCoL + Wickler's Method

Static analysis is an analysis method known for the helpful internal validation of a domain description, such as to examine the effects of static knowledge or to discover state invariants. This section presents a method for the visual investigation and validation of a Planning Domain Definition Language (PDDL) domain model through finding a static graph and related properties from the training plans and domain operators. This uses the Freecell and Grid domains as running example in order to demonstrate the combined usage of both techniques (ASCoL + Wickler's). This is because the Freecell domain is comprehensive and provides a suitable framework on which to visualise the effectiveness of the approach. All of the ten operators of the domain satisfy the condition (of having the same types) and it is rich in terms of MSRs, ESRs and Shift operators. It encodes a network of static constraints which includes the allowed sequential arrangement of cards in the free cells, the home cells and among the card columns, used within ten operators of the domain model. We randomly chose Grid domain as our second example.

In the combined use of both the methods, we first learn the MSRs using ASCoL. Based on those MSRs, Wickler's method then analyse the domain definition based on static analysis of ESRs and Shift Operators. Together both systems produce the static graph structure with nodes as objects of certain type from the domain and edges as relationships between the nodes.

Following is the operator *homefromfreecell* from the Freecell domain. This operator sends cards from freecells to the top of the home cells based on the ascending order of the playing cards. Here (successor ?vcard ?vhomecard) and (successor ?ncells ?cells) are two MSRs in the operator.

```
(:action homefromfreecell
:parameters (?card - card ?suit - suit
             ?vcard - num ?homecard
             - card ?vhomecard - num
             ?cells ?ncells - num)
:precondition(and
   (incell ?card)
   (home ?homecard)
   (suit ?card ?suit)
   (suit ?homecard ?suit)
   (value ?card ?vcard)
   (value ?homecard ?vhomecard)
   (successor ?vcard ?vhomecard)
   (cellspace ?cells)
   (successor ?ncells ?cells))
:effect (and
   (home ?card)
   (cellspace ?ncells)
```

```
    (not (incell ?card))
    (not (cellspace ?cells))
    (not (home ?homecard)))
)
```

Main Static Relation (MSR)

ASCoL successfully learns static graph G for both the MSRs from operator *homefromfreecell*. For each of the identified pairs of arguments, ASCoL generates a directed graph by considering the objects used in the plan traces. In order to exemplify how directed graphs are generated, consider the following instances of the *homefromfreecell* action which are collected from the input set of plan sequences.

homefromfreecell(club4,club,n4,club3,n3,n0,n1)
homefromfreecell(diamond4,diamond,n4,diamond3,n3,n0,n1)
homefromfreecell(diamond5,diamond,n5,diamond4,n4,n1,n2)
homefromfreecell(spade2,spade,n2,spadea,n1,n0,n1)
homefromfreecell(spade3,spade,n3,spade2,n2,n1,n2)
homefromfreecell(heart3,heart,n3,heart2,n2,n0,n1)
homefromfreecell(heart4,heart,n4,heart3,n3,n1,n2)
homefromfreecell(spade7,spade,n7,spade6,n6,n0,n1)
homefromfreecell(heart6,heart,n6,heart5,n5,n1,n2)
homefromfreecell(spade8,spade,n8,spade7,n7,n2,n3)
homefromfreecell(heart8,heart,n8,heart7,n7,n3,n4)

For type *num*, the following six vertex pairs are identified for the action *homefromfreecell*:

pair1 (?vcard, ?vhomecard), pair2 (?vcard, ?cells), pair3 (?vcard, ?ncells), pair4 (?vhomecard, ?cells), pair5 (?vhomecard, ?ncells), pair6 (?cells, ?ncells).

In order to generate the directed graph for the pair1 arguments of type *num*, i.e., pair1 (?vcard, ?vhomecard), ASCoL considers all of the objects used as the third and fifth arguments of the *homefromfreecell* action instances. Given our example, all the unique IDs include:

IDs = { n4, n3, n5, n2, n1, n7, n6, n8 }.

The *Conn* set includes the following unique edges:

Conn = [(n4,n3),(n5,n4),(n2, n1),(n3, n2),(n7, n6),(n6, n5),(n8, n7)].
Figure 2 represents *IDs* and *Conn* in the form of a graph embedding. This linear order in the third and fifth arguments of the *homefromfreecell* action instances suggests that there is an important one-to-one relationship between the two positions. This linear relationship is implicit in the plan traces and cannot be captured by assembling the transition behaviour of an individual type of objects.

ASCoL draws a graph structure based only on the available number of edges to learn the relationship. In principle, relations can be predicted by only looking at value-pairs of parameters but the difficulty level rises with the increase in the number of parameters of the same type. Using a graph structure makes it easy to analyse such situations where an action has more than two same typed parameters. It also makes

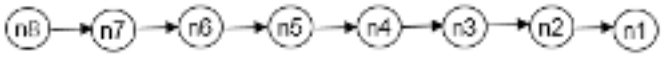


Figure 2: Example of a directed graph with a linear structure (Pair 1: type num).

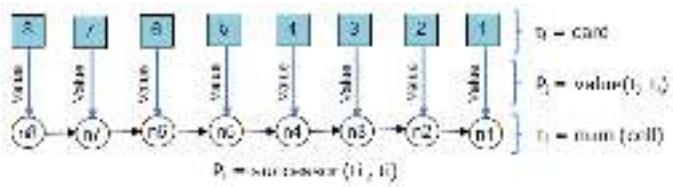


Figure 3: ESRs over a linear graph for homefromfreecell.

the system visually explicit for human users when generating or debugging domains. To determine whether to define the relation, it uses a specific graph structure to decide the correct pair of arguments for producing a MSR.

Extended Static Relations (ESRs)

We continue with our example of operator *homefromfreecell* from the benchmark Freecell domain. There are two ESRs in the benchmark domain for (successor ?vcard ?vhomecard) predicate:

- (value ?card ?vcard)
- (value ?homecard ?vhomecard)

Both these above mentioned ESRs explain the objects of (successor ?vcard ?vhomecard) in terms of face value that both card objects contain.

- MSR = P_i = (successor ?vcard ?vhomecard)
- ESR = P_j = (value ?card ?vcard) and (value ?homecard ?vhomecard)
- Static type of MSR = t_i = num
- Node-fixed type = t_j = card

Each identified ESR in an operator explains the MSR to further level of details in different ways depending upon the nature of the domain e.g. ESR relates some physical objects to one node in the graph in transportation domains. Similarly, in skill-based games, it explains the value, cost or property of the node. The same property or ESR may repeat for more than one node of MSR graph. ESRs P_j with the node-fixed type are represented using rectangular nodes in figure 3 over totally ordered static graph.

Level-Two ESRs (ESR_{L2})

Each ESR can further be extended if it fulfills the conditions for ESR for producing further extension in the static graph with respect to the node-fixed type, i.e. considering ESR's node-fixed static type t_j in place of MSR's static type t_i. We call it Level-two ESRs (ESR_{L2}).

To explain this, the same example can further be expanded in the form of (suit ?card ?suit) and (suit ?homecard ?suit)

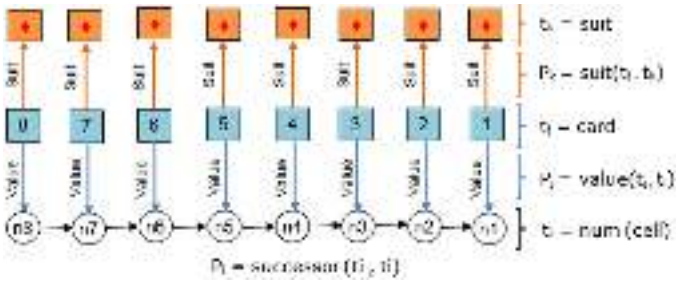


Figure 4: ESRs and ESRL2 over a linear graph with MSR.

considering previous $t_j = t_i$ = card and level-two $t_j = t_k$ = suit as demonstrated in figure 4. ESR_{L2} P_k are represented using rectangular nodes with marked label P_k.

The extended part of the graph contains directed edges with node-fixed type as a source of each edge. This is because the ESR P_j could possibly be functional in both directions. This mainly depends upon the order of predicate arguments in the domain under verification.

Grid Domain Example: The strips version of Grid domain was used in the first planning competition. Without going into the details of the domain description, we just describe how the following *Unlock* operator of this domain can be used as an example and the source of finding ESR and ESR_{L2}.

```
(:action unlock
 :parameters (?curpos ?lockpos ?key ?shape)
 :precondition(and
    (place ?curpos)
    (place ?lockpos)
    (key ?key)
    (shape ?shape)
    (conn ?curpos ?lockpos)
    (key-shape ?key ?shape)
    (lock-shape ?lockpos ?shape)
    (at-robot ?curpos)
    (locked ?lockpos)
    (holding ?key))
 :effect (and
    (open ?lockpos)
    (not (locked ?lockpos)))
)
```

Unlock describes the lock position for the lock with a particular key that has a particular shape. Here the MSR or the static graph is defined by the Conn predicate that indicates the relationship between current and lock positions of the lock. Keeping in-mind the definition of ESRs, there is a predicate lock-shape which fulfills both the conditions of being ESR and have node fixed-type i.e. a binary predicate as well as having two variables of different types including the type of objects from MSRs:

Here

- MSR = P_i = (conn ?curpos ?lockpos)
- ESR = P_j = (lock-shape ?lockpos ?shape)
- Static type of MSR = t_i = position
- Node-fixed type = t_j = shape

To find ESR_{L2}, corresponding static graph analysis can further be expanded in the form of predicate (key-shape ?key ?shape) considering previous $t_j = t_i$ = shape and at level-two $t_j = t_k$ = key.

By using this search and analysis method, it not only produces the static relationships between objects in the plan traces but also discovers a further level of networking between the objects depending upon the nature of the domain. For instance, in patience card games, it can provide the relationship between a card, its face value and its suit, as previously shown in the Freecell example.

Shift Operators or Static Modifier (O_{SM})

In our example of Freecell domain, we use following *move-b* operator as an illustrative example of O_{SM}. The operator *move-b* moves a card between columns when that card happens to be the last card in the column.

```
(:action move-b
 :parameters (?card ?newcard - card
              ?cols ?ncols - num)
 :precondition(and
    (bottomcol ?card)
    (clear ?newcard)
    (canstack ?card ?newcard)
    (colspace ?cols)
    (successor ?ncols ?cols))
 :effect (and
    (on ?card ?newcard)
    (colspace ?ncols)
    (not (bottomcol ?card))
    (not (clear ?newcard))
    (not (colspace ?cols)))
)
```

There are two MSRs involved in this operator:

- (successor ?ncols ?cols)
- (canstack ?card ?newcard)

Here, (colspace ?cols) is the modifier predicate of (successor ?ncols ?cols) which represents the transition of the number of empty columns before and after the action execution in the form of the action precondition, effect+ and effect-, respectively. No predicate fulfills the conditions to be the shift predicate for (canstack ?card ?newcard).

From the same domain Freecell, the operators *homefromfreecell*, *colfromfreecell*, *sendtofree*, *newcolfromfreecell* and *sendtofree-b* also modifies *(cellspace ?cells)* to *(cellspace ?ncells)* using MSR *(successor ?cells ?ncells)* . Operators *sendtohome-b*, *sendtonewcol*, *newcolfromfreecell* and *sendtofree-b* has a modifier predicate *(colspace ?cols)* along *(successor ?ncols ?cols)* and modifies to *(colspace ?ncols)*.

Grid Domain Example: From Grid domain, we use the *move* operator as an illustrative example of O_{SM}.

```
(:action move
 :parameters (?curpos ?nextpos)
```

```
 :precondition(and
    (place ?curpos)
    (place ?nextpos)
    (at-robot ?curpos)
    (conn ?curpos ?nextpos)
    (open ?nextpos))
 :effect (and
    (at-robot ?nextpos)
    (not (at-robot ?curpos)))
)
```

(conn ?curpos ?nextpos) is the MSR involved in this operator. Here, (at-robot ?curpos) is the shift predicate for (conn ?curpos ?nextpos) which represents the transition of the position of the robot from its current position to its next position in the form of the action precondition, effect+ and effect-, respectively.

Discussion and Evaluation

The correctness of the plans depends on the correctness of the domain model in model-based planning systems. The aim of the analysis presented is the better understanding of the domain model in a more intuitive way by automatic identification of static domain structure at early design as well as at validation stage instead of doing a manual analysis Among most common ways of checking domain correctness followed in literature are by explicit testing or by using a more formal method of model checking. Model checking is expensive in terms of computational power since it looks for all the reachable states of the domain model and the size of the problem increase exponentially with the domain complexity.

To evaluated the approach and the visual output, we manually tested the present benchmark domains from IPC. To obtain the input plan traces for each domain, available problem generators are used to create the training problems, which are subsequently solved by planners. Potential plan traces can be gathered from multiple sources and applications, for example, the sequence of work-flow in some process execution, logs of commands for installing a piece of software or the moves or steps captured from game playing etc.

The first step is the generation of static constraints for all the domains by considering the plan traces. The acquisition of static constraints and corresponding graphs from plan traces is evaluated by manually comparing the results generated with the known benchmark domains and by performing reachability test for a variety of values of static facts.

The totally ordered graph in the outcome represents a strong static relationship between the arguments. In case of a Directed Cyclic Graph, ASCoL tests if G is fully connected or not. By exploiting the feature of shift operator for the purpose of domain analysis, it is interesting to check if the graph is connected or disconnected. If it is disconnected then which nodes are reachable from the available node. All the nodes reachable from the initial states (which occur as shift property of any shift operator), would also hold this shifting property. This eases the evaluation of reachability condition and guides solution search.

With MSRs as the edges of the graph, the second step again uses graph behaviour to identify and validate the functional properties of the static graphs by identifying the extended nodes and corresponding node-fixed types, i.e. to identify the objects that cannot move between nodes in the static graph and have some fixed relationship with nodes in the static graph. It applies the definition of $ESRs$, ESR_{L2} and shift operators and finds preconditions and effects that fulfils all the conditions. The algorithm takes two parameters, an operator and an MSR. It returns true if and only if the given operator is a shift operator wrt. the argument type of the given predicate.

We used a batch of eleven domains that encode meaningful knowledge. From considered domains, fourteen more examples have been found in addition to Freecell and Grid domain. Table 1 shows the names of domains, the names of the shift/modifier operators (O_{SM}) in each domain, the MSR of the O_{SM}, the discovered shift/modifier predicate pre_{SM} and the arity of pre_{SM}. All the domains used for the evaluation were the simple PDDL versions of the domains. From all fourteen examples, ASCoL learns some additional static relations in *Move* operator of Gripper domain, *Fly* operator of Zenotravel domain and *Fly-Airplane* operator of Logistics domain, while their benchmark hand-coded versions do not contain any MSR. Upon visually generating the additional learnt static facts and verifying the test plans for such domains we concluded that such relations do not reduce the solvability of problems and help in the pruning of search space.

Out of all the domains considered for evaluation, only Freecell, Grid and Logistics domain contain Extended Static Relations (ESRs). Freecell and Grid domains are already discussed in the description section of the method. The benchmark Logistics domain does not contain any same-typed MSR while ASCoL discovers an additional MSR in the Drive-Truck operator of the domain. The additional static relation *connect (loc-from, loc-to)* connects the two locations for the movement of a truck across different cities. Based on the additional MSR discovered, *connect (loc-from, loc-to)* visually extend into two ESRs in the same operator i.e.*In-city (loc-from city)* and *In-city (loc-to city)*.

The above definitions, algorithms and evaluation let us analyse and determine which relations represent edges and which types represent nodes in a static graph which will be encoded in the domain model. In terms of utility, a static graph can signify many things and it is more general than specific for some domains like transport domains.

The static knowledge of a domain model is usually fixed and cannot be changed by the action effects. Therefore, the graph formed by this analysis system can be analysed independently from the state information of the rest of the domain states. The graph analysis will also be correct for all the problem instances attached to the analysed domain.

Application and Conclusion

Modelling a domain involves both capturing the dynamics and background knowledge and validating the captured knowledge to the level where it maximum coincides with actual possible worlds in the domain. The accepted wisdom from the field of Formal Methods is to capture the structure

Domains	Operator(s) O_{SM}	Binary MSR	Shift/Modifier Predicate pre_i	Uniary/Binary
Ferry	Sail	Not-equal	(at-ferry ?location)	Unary
Gold-miner	Move	Connected	(at-robot ?location)	Unary
Gripper	Move	Connected	(at-robby ?room)	Unary
Logistics	Drive-Truck	Connected	(at ?truck ?location)	Binary
Miconic	Up, Down	Above	(Lift-at ?floor)	Unary
TPP	Drive	Connected	(at ?truck ?location)	Binary
Trucks	Drive	Connected	(at ?truck ?location)	Binary
Trucks	Drive	Next	(time-now ?time)	Unary
Visitall	Move	Connected	(at-robot ?place)	Unary
Spanner	Walk	Link	(at ?man ?location)	Binary
Storage	Move, Go-in, Go-out	Connected	(at ?hoist ?area)	Binary
Zenotravel	Fly	Next	(Fuel-level ?aircraft ?level)	Binary
Zenotravel	Fly	Route	(at ? aircraft ?city)	Binary
Zenotravel	Refuel	Next	(Fuel-level ?aircraft ?level)	Binary

Table 1: Examples of Static Modifier Operators (O_{SM}).

and in particular the invariants of a domain in a formal language (Biundo et al. 2003). The contribution of this paper is the development and analyses of static knowledge of the domain model visually in the form of graphs and by manual reachability testing.

The combined method described in this paper builds on the static domain analysis in terms of automatic identification of static graph relations (MSRs, as ASCoL names it), ESRs and O_{SM}. Specifically, by using Static Modifier property, it becomes easy to understand the manipulation of world states in guiding the search space in planning.

Graph embedding and analysis theory are particularly suitable to develop this approach because graphs can be used to represent logical semantics (meaning of propositions and of their formal analogues) without using the language of logic but visual notions and this is why Static support for model analysis is mostly visual. In addition, because many efficient graph-processing algorithms exist, thus graphs can be exploited as a good computational mechanism to compute relations between objects. Particularly, graphs that represent static relationships show the permanent relationships between constant objects of a problem (Botea et al. 2005).

This combined analysis method and the definitions that it is based on exploits certain representational choices that are mostly used when formally representing knowledge in PDDL domain models. There may be substitutes that we have not reflected here that may make the analysis unsuccessful even with the presence of static graphs. It supports knowledge engineers and builds on domain analysis based on the features described. Apart from AI planning and Knowledge Engineering, other wider application areas that involve the synthesis of constraints or invariants can benefit from the method. Examples include the detection of anomalies in domain design, extraction of maps of locations from past activity, learning of the static rules of games such as solitaires, draft etc.

It can also be used by other systems/individuals to extract patterns in large real-world case studies which cannot be dynamically discovered using finite state automata, e.g. to extract data from large databases, to analyse social networks or work markets, the diagnosis of certain diseases like spreading of the virus. Hence, the internet which is a network on its own can be a good application area where different HTML documents act as the nodes of the graph and the hyperlinks act as the edges in the graph. An example includes Google, which uses the structure of Internet in its famous PageRank algorithm (Ma, Guan, and Zhao 2008) for websites ranking.

Although the combined method works on domains, problem instances can also be analysed based on the fact that often the initial conditions in a planning problem that comprise of the static relations are reusable and does not change across a range of problems. Apart from static constraints, clearly, there is much more to an operator of a domain that makes a planning domain and problem complex. The long-term goal is to extend this initial idea of integrating different KE systems to include dynamic analysis and present formal validation method which can provide wider coverage of domain features and lead to increased confidence in the correctness of validated domain model. WE also aim to learn how to structure domain models which are more responsive to static analysis.

References

Bensalem, S.; Havelund, K.; and Orlandini, A. 2014. Verification and validation meet planning and scheduling.

Biundo, S.; Aylett, R.; Beetz, M.; Borrajo, D.; Cesta, A.; Grant, T.; McCluskey, T.; Milani, A.; and Verfaille, G. 2003. Technological roadmap on ai planning and scheduling.

Botea, A.; Enzenberger, M.; Müller, M.; and Schaeffer, J. 2005. Macro-ff: Improving ai planning with automatically learned macro-operators. *Journal of Artificial Intelligence Research* 24:581–621.

Giunchiglia, F., and Traverso, P. 2000. Planning as model checking. In Biundo, S., and Fox, M., eds., *Recent Advances in AI Planning*, 1–20. Berlin, Heidelberg: Springer Berlin Heidelberg.

González Ferrer, A. 2012. *Knowledge engineering techniques for the translation of process models into temporal*

hierarchical planning and scheduling domains. Granada: Universidad de Granada.

Gregory, P., and Cresswell, S. 2015. Domain model acquisition in the presence of static relations in the lop system. In *ICAPS*, 97–105.

Howey, R.; Long, D.; and Fox, M. 2004. Val: Automatic plan validation, continuous effects and mixed initiative planning using pddl. In *Tools with Artificial Intelligence, 2004. ICTAI 2004. 16th IEEE International Conference on*, 294–301. IEEE.

Jilani, R.; Crampton, A.; Kitchin, D.; and Vallati, M. 2015. Ascol: A tool for improving automatic planning domain model acquisition. In *Congress of the Italian Association for Artificial Intelligence*, 438–451. Springer.

Khatib, L.; Muscettola, N.; and Havelund, K. 2001. Mapping temporal planning constraints into timed automata. In *Temporal Representation and Reasoning, 2001. TIME 2001. Proceedings. Eighth International Symposium on*, 21–27. IEEE.

Long, D.; Fox, M.; and Howey, R. 2009. Planning domains and plans: validation, verification and analysis. In *Proc. Workshop on V&V of Planning and Scheduling Systems*.

Ma, N.; Guan, J.; and Zhao, Y. 2008. Bringing pagerank to the citation analysis. *Information Processing & Management* 44(2):800–810.

Shoeeb, S. 2012. Investigation into the theoretical properties of, and the relationship between, ai planning domain models.

Simpson, R. M.; Kitchin, D. E.; and McCluskey, T. 2007. Planning domain definition using gipo. *The Knowledge Engineering Review* 22(02):117–134.

Vaquero, T. S.; Romero, V.; Tonidandel, F.; and Silva, J. R. 2007. itSIMPLE 2.0: An Integrated Tool for Designing Planning Domains. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, 336–343.

Vaquero, T. S.; Silva, J. R.; and Beck, J. C. 2011. A brief review of tools and methods for knowledge engineering for planning & scheduling. *KEPS 2011* 7.

Wickler, G. 2013. Using static graphs in planning domains to understand domain dynamics. *Knowledge Engineering for Planning and Scheduling* 69.

Distributed Planning and Model Learning for Urban Traffic Control

Alberto Pozanco **Susana Fernández** **Daniel Borrajo**
Departamento de Informática, Universidad Carlos III de Madrid
Avda. de la Universidad, 30. 28911 Leganés (Madrid). Spain
apozanco@pa.uc3m.es, sfarregu@inf.uc3m.es, dborrajo@ia.uc3m.es

Abstract

Urban Traffic Control is a key problem for most big cities. Current approaches to handle the city traffic rely on controlling traffic lights. The systems in operation range from static control of traffic light phases to adaptive systems based on numeric models and traffic sensors. Recently, some planning-based approaches have also been proposed. We have identified two main difficulties to the wide use of planning techniques in this domain: generating the control models is a difficult task; and some algorithms scale poorly. In this paper we present APTC, a control system based on Automated Planning, that successfully overcomes these two problems. It combines techniques that continuously: learn an accurate planning model; and also divide the city for distributed reasoning in order to scale to large city networks. Experimental results show that APTC outperforms static approaches as well as other planning-based systems. We also show that the combination of both approaches improves over using only one of them.

Introduction

In the last decades the world's population has grown steadily, becoming more urbanised over the years. This growth consequently increases the public transport demand as well as the number of cars and vehicle movements. However, traffic infrastructures do not grow at the same rate, so implementing efficient Urban Traffic Control (UTC) systems is increasingly important. Hence, traffic management can influence the entire city dynamics, causing significant damages to the population, ranging from unnecessary time and fuel consumption up to deteriorating citizen's health. The traffic control task is a difficult one, since it involves stochastic behavior by independent agents (drivers) and plenty of unforeseen events that can affect the transportation network, such as maintenance, accidents, weather or sports events.

Most current systems control the city traffic using macroscopic approaches (Treiber and Kesting 2013) that model traffic at the flow level rather than taking into account isolated cars. They usually set traffic lights programs, that are defined in terms of three parameters: split and cycle, which refer to the amount of green and red time allocated to each traffic light; and offset, that represents the difference between the start of green time at two consecutive intersections. An appropriate offsets' setting at various connected traffic lights generates "green waves" that allow vehicles not to stop in several consecutive junctions. There are many known algorithms to define those programs, ranging from early static off-line approaches that are still in use in many cities, to most recent adaptive approaches that change the programs according to the network's state (Papageorgiou et al. 2007; Hamilton et al. 2013). An example of a successful adaptive approach is the SCOOT system, a commercial product that uses information coming from traffic sensors to feed numerical models (Bretherton, Wood, and Bowen 1998). A weak point of these systems is that they cannot deal well with dynamic incidents (Vallati et al. 2016). Also, their models are usually difficult to maintain.

Recently, some Artificial Intelligence (AI) approaches have emerged, using diverse techniques: neural networks (Box and Waterson 2012), reinforcement learning (Jin and Ma 2017); or scheduling techniques (Xie, Smith, and Barlow 2012). Automated Planning (AP) has also been recently shown to perform well in this kind of tasks (Cenamor et al. 2014; Gulič, Olivares, and Borrajo 2016; Vallati et al. 2016). The main advantage of using AP is that the problem can be modeled using a declarative language in combination to powerful reasoning engines. Thus, traffic engineers can easily include or modify new actions, sensor information or metrics to adapt the model to evolving traffic conditions.

In general, previous techniques and particularly the ones based on AP have two main drawbacks. Firstly, given the stochastic nature of the control task, as well as the amount of different ways to control traffic lights, properly modeling the planning task requires some knowledge engineering effort. Moreover, in most cases, the defined model does not perfectly fit the real scenario, which affects the system's behavior. Also, most of these models assume all junctions share the same behavior within a city and across cities. Secondly, these approaches scale poorly and can not be implemented in large areas or cities with numerous streets and traffic lights.

In this paper we present APTC (Automated Planning for Traffic Control), a system based on AP to perform UTC. The goal of APTC is to overcome the previously mentioned two problems. First, we propose to automatically update the planning domain model through continuous incremental learning. The automatic generation of planning domains in stochastic environments has been previously studied (García-Martínez and Borrajo 2000; Pasula, Zettle-

moyer, and Kaelbling 2007; Jiménez et al. 2012; Jiménez, Fernández, and Borrajo 2013; Martínez et al. 2016; Mourao 2014). However, these domain-independent approaches are not adequate when the learning tasks involve actions that use many parameters as it is the case of UTC actions. Therefore we propose a domain-dependent model updating approach. The learning technique monitors the observed states at each junction and automatically generates actions that match the junction's dynamics. This allows APTC to quickly adapt to the city traffic behavior and its changes, generating better planning domains requiring less model engineering work.

Second, we propose to use distributed planning by dividing the city into a set of areas. Given that it is difficult to define those areas "a priori", APTC identifies the most important junctions and divides the city according to the traffic flows that dynamically arise or disappear over time. This city splitting criteria not also leads APTC to better scalability. By keeping the junctions involved in a flow in the same planning problem, APTC can automatically generate "green waves". The system generates them by setting the traffic lights in such a way that they allow the vehicles to quickly traverse the junctions involved in the traffic flows. A planning problem is generated in each area and they are solved asynchronously. The resulting plans are concatenated and executed to control the traffic lights. APTC can be seen as an instance of a fully autonomic (autonomous) system (Huebscher and McCann 2008), given that it incorporates many self-* properties, as self-monitoring (continuous observation), self-diagnosis (undesired behavior detection), self-optimization (planning), self-healing (execution of traffic control actions) and self-adaptation (learning).

Planning Models for UTC

We propose to use AP to solve traffic control tasks. From all the different kinds of available planning techniques, we will use those that take as input an explicit model described in the standard PDDL language (Planning Domain Description Language) (Fox and Long 2003). These planning techniques take as input a planning task and return a plan that solves it.

Planning Models

A single-agent STRIPS planning task can be formally defined as a tuple $\Pi = \{F, A, I, G\}$, where F is a set of propositions, A is a set of instantiated actions, $I \subseteq F$ is an initial state, and $G \subseteq F$ is a set of goals. Each action $a \in A$ is described by a set of preconditions (pre(a)), that represent propositions that must be true (or false for negative preconditions) in a state to execute the action and a set of effects (eff(a)), propositions that are expected to be true (add(a) effects) or false (del(a) effects) after execution of the action. The application of an action a in a state s is defined by a function γ, such that $\gamma(s, a) = (s \setminus \text{del}(a)) \cup \text{add}(a)$ if pre(a)$\subseteq$ s and s otherwise (it cannot be applied). Planners should generate as output a sequence of actions, called a plan, $\pi = (a_1, \ldots, a_n)$ such that if applied in order from the initial state I would result in a state s_n, where the goals are true, $G \subseteq s_n$. Under this definition, A and F are fully grounded propositions. To alleviate the definition of planning tasks, the AP community has defined PDDL, a high-level language that allows planning users to easily define these tasks. It is based on predicate logic, and requires the definition of two files: domain and problem. The domain model D contains the definition of predicates for representing sets of propositions and the actions that agents can perform. The problem P describes the particular task instance to be solved; i.e., the objects involved, the initial state and the set of goals to achieve.

Modeling UTC with PDDL

There have been different approaches to model UTC from an AP point of view, but all of them rely on acting over the traffic lights. Vallati *et al.* 2016 propose a PDDL+ formulation (Fox and Long 2006). PDDL+ is an extension of PDDL to model mixed discrete-continuous domains. This approach switches the traffic lights for a certain amount of time depending on the queues of vehicles on the streets entering the junction.

In parallel, and using a simpler PDDL domain, Gulić *et al.* modeled UTC taking into account street's density levels instead of flows and queues of cars in their IAS system (Gulìc, Olivares, and Borrajo 2016). Actions are executed over the city network only when a high density level is detected at any street. This is done for a fixed time and then the system monitors if the congestion has been solved, i.e., the density is low in all streets, returning to the default program. This planning model assumes the world is deterministic and the agent has full observability. But UTC does not follow these premises, since the actions have stochastic outcomes and the agents have partial observability. In order to deal with uncertainty, they follow a simple and popular approach; reasoning (planning) with a deterministic world model and when execution of some action fails (the congestions are not solved), the agent replans (Yoon, Fern, and Givan 2007). Besides, as in robotics, this approach usually employs a simplified model to solve high-level deliberative problem solving. And there is a low-level reasoning model that takes care of some of the complexities of dealing with numeric quantities and continuous processes. In their case, the low-level model translates high-level actions into each crossing's traffic lights for specific amounts of time. In this paper, we will take the UTC model used in IAS as a baseline. It presents two advantages over using PDDL+: there are many more planners that can work with PDDL; and discrete models are usually easier to define and learn. The potential disadvantage would be that the IAS PDDL models are not as accurate as the PDDL+ ones, given that they do not deal with continuous numeric models. However, we can alleviate this problem by the low-level (simple) behavior in this case where the relevant parameters to traffic lights control are set appropriately given a high-level action. And, also, by our learning mechanism later described.

The first step when generating a planning domain is to determine the predicates and the actions to use. Each high-level action in IAS controls all the traffic lights of a junction at once. As an example, in a four-way junction there are four incoming streets and another four outgoing streets (each with a finite number of lanes) and each incoming street

```
(:action one-high
 :parameters (?c - junction ?sin1 - street ...
              ?sout1 - street ...)
 :precondition (and (goes-into ?sin1 ?c)
                    (in-front-of ?sin1 ?sin3)
                    ...
                    (goes-out ?sout4 ?c)
                    (not (= ?sin1 ?sin2))
                    ...
                    (densityLevel ?sin1 high)
                    (densityLevel ?sin3 low)
                    ...)
 :effect (and (densityLevel ?sin1 low)
              (not (densityLevel ?sin1 high))
              (densityLevel ?sin3 low)
              (not (densityLevel ?sin3 high))
              (densityLevel ?sout2 low)
              (not (densityLevel ?sout2 high))))
```

Figure 1: Part of an example description of a PDDL action that sets one traffic light of a junction to green.

is controlled by one traffic light. The actions that can be applied at this specific type of junctions are limited: you can set to green one of the four traffic lights separatedly, or set to green two of them if they are in front of each other. The high-level action that sets to green a traffic light is translated into several low-level actions to set to red the other traffic lights in the junction.

Since we want to model how to control traffic lights, predicates allow us to represent the current state of the junction. We use most of the predicates defined in IAS. There are static predicates that reflect the city network composition such as `goes-into` and `goes-out` to indicate whether a street enters or leaves a junction; and dynamic predicates such as `densityLevel` that indicates the congestion level of a given street.[1] Since sensors return numeric values for the density, we applied a discretization step for its values, using a threshold. We use the same two density values defined by Gulić *et al.*; `high` for density values higher than 0.35 and `low` otherwise.

In UTC, actions should decide how to control the traffic lights. In our case, we will use again the same approach introduced by IAS that sets the traffic lights to green when it decides that the default program is performing badly. So, the preconditions of each action check if the densities of some street sections are high (dynamic predicates) as well as some static preconditions (network structure around each junction). If the action is executed, the effects describe the expected changes in the new state. Figure 1 shows the definition of an action in which a traffic light is set to green.

Problems are mainly composed of a set of objects, an initial state and a set of goals. In our case, the objects are the streets and the junctions. The initial state would be composed of: the static part of the city i.e., the connections between the streets and the geometry of the junctions; and the dynamic part made of the initial density levels of the streets. The goal would be to have low density in all the streets. A potential planning problem would be as shown in Figure 2.

[1]This value is provided by the simulator we use, and it would be generated by street sensors in a real scenario.

```
(define (problem traffic1) (:domain traffic)
  (:objects sin1 ... sout7 - street
            j1 j2 - junction)
  (:init (goes-into sin1 j1)
         (in-front-of sin1 sin3)
         (densityLevel sin1 high)
         (densityLevel sin2 low)...)
  (:goal (and (densityLevel sin1 low)
              (densityLevel sin3 low)
              (densityLevel sout7 low) ...)))
```

Figure 2: Part of an example description of a PDDL initial problem. The goal is to achieve low density in all the streets.

APTC UTC Model

IAS model is a good baseline for our goals, since it is based on a simple traffic model that can be modified by learning. However, IAS actions' descriptions are limited since they only allowed each junction to set one green traffic light at each time step. In order to generate a better model, we would have to study the different cases. The next modeling step takes into account all the possible combinations of density levels and traffic lights that can be set to green at the same time. Finally, it is necessary to guess the effects of applying an action. For instance, in a four-ways junction, if only one incoming street has high density and that traffic light is set to green, we need to guess the density levels of the four outgoing streets after applying this action in order to define the action effects. Clearly, it becomes a hard knowledge engineering task, given the amount of different alternatives, and the variety of behaviors in different junctions and traffic/weather/day conditions.

Also, it is impossible to achieve all goals, i.e., low density in all the streets, under some traffic circumstances. IAS did not return any plan in these cases, which degraded the system's performance. To solve this problem, we transform the hard goals to soft ones following the compilation proposed by Keyder and Geffner [2009]. Individual plans' quality does not necessarily relate to overall quality, since the latter can only be measured at the end of the execution (with metrics such as average waiting time, or pollution). Therefore, we will again use an scheme where we assume that executing more actions implies better performance. Our final domain is composed of ten actions. Only the most basic cases are contemplated in these actions. Then, we propose to automatically update that planning domain by learning the city dynamics, generating junction-based actions that could lead the system to better performance.

APTC Architecture

APTC architecture is based on IAS's and PELEA (Guzmán et al. 2012) architecture and comprises five modules: EXECUTION, MONITORING, PLANNING, MODEL LEARNER and CITY SPLITTER. It is shown in Figure 3.

The EXECUTION component receives an initial AP domain D along with other APTC parameters (M and L). The monitoring rate M indicates how often the city is observed and the duration of the applied actions, i.e., phase sequence. The learning rate L refers to how much time will elapse between two learning episodes. EXECUTION is connected with

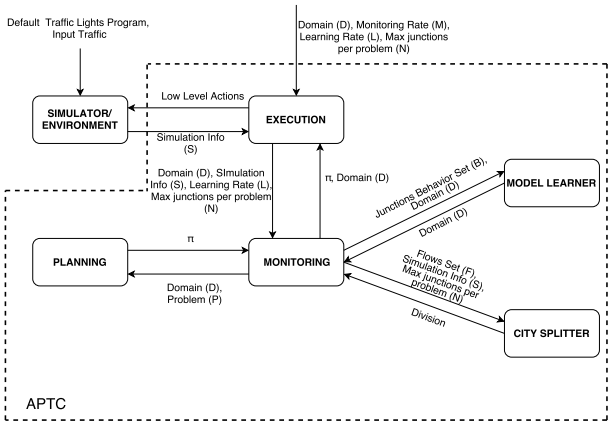


Figure 3: Planning and execution architecture that includes model learning capabilities.

the SIMULATOR that sends the simulation info S every time step and translates the planning actions to low-level actions applied to the traffic lights. S is a directed weighted graph that contains the geometry of the network, $S = (J, E)$. J is the set of vertices (city junctions) and E are the edges that connect those vertices (city street sections). The edges that leave the network to vertices outside it are connected to artificial vertices. An edge $e \in E$ connects two vertices $j_1, j_2 \in J$ if there is a street section that goes from j_1 to j_2. Each $e \in E$ has an associated weight $w(e)$ with values from 0 to 1. At each time step, the weights represent the current density level ratio of that street section. Thus, the structure of the graph has a static component (represented by J and E) and a dynamic component (represented by the weights). At every M steps, EXECUTION asks MONITORING for a plan π and a domain D. If it receives a plan, it is translated from PDDL actions to low-level traffic lights control signals that are sent to the SIMULATOR for their execution. Otherwise, it returns the empty set, so the default traffic light's program in the simulator decides the next actions on the traffic light.

MONITORING is described in Algorithm 1. MONITORING receives a domain, D, the information on the city network, S, the parameter L and the maximum junctions per problem N. It returns a plan π and an updated domain D. It maintains and update two sets based on S: F (Flows) and B (Junction's Behavior), as well as the last executed plan π. F contains information on how the cars traverse the city network and it will be used by the CITY SPLITTER module. B contains information related to the effects of applying an action at a traffic light, and it will be used by the MODEL LEARNER module. The next subsections describe these two modules in detail and how they update D and generate a set of problems $\mathcal{P}$. If a high density street is detected by MONITORING, it automatically generates $\mathcal{P}$ based on the network's state S and the information returned by the CITY SPLITTER. Then, the PLANNER module is called asynchronously with D and every problem $P \in \mathcal{P}$, and the resulting plans are concatenated and returned. We use the notation $\pi_1 \oplus \pi_2$ to represent the concatenation of two plans.

CITY SPLITTER returns a set of disjoint planning tasks in terms of junctions. There is a single traffic light program per junction that controls the incoming streets. The solution of each problem will be composed of actions that affect the junction traffic lights. Since there is only one action per junction and the junctions are only present in one planning problem, we can concatenate the plan of each individual planning task and all these actions are executed in the network in parallel. This strategy will set to green traffic lights that are predicted by the model to have high density levels in the next M (monitoring rate) seconds (starting at the beginning of the planning episodes). The result of this strategy is that green waves are created during these M seconds which can be considered as equivalent to controlling the offsets in classical approaches. The system uses planning to take into account the effects of setting to green a traffic light and how it will affect the surrounding junctions. This is a key difference with respect to reactive systems that usually only consider one traffic light.

Algorithm 1 MONITORING(D, S, L, N)

Inputs: domain D, graph S, learning ratio L, max junctions per problem N
Outputs: π, D

```
B, F, π ←RETRIEVE()
F ←UPDATEF(S)
B ←UPDATEB(S, π)
if simulation-step mod(L) = 0 then
    D ←MODEL LEARNER(B, D)
    A ←CITY SPLITTER(F, S, N)
    F, B ← ∅
if ∃e ∈ E, ISHIGH(w(e))=True then
    P ←GENERATEPROBLEMS(A, S)
    for P ∈ P do
        π_P ←PLANNING(D, P)
    π ← π_1 ⊕ π_2 ⊕ ... ⊕ π_n
else
    π ← ∅
STORE(B, F, π)
return π, D
```

Model Learner

In stochastic environments such as UTC, the world model does not perfectly fit the real world. Also, in the particular case of UTC, most works assume that the model of each action is shared by all network junctions at any time step. But this assumption does not hold in most cases. To overcome these problems we propose to apply learning techniques to continuously adapt and improve the planning model as the actions are executed in the environment. This adaptation starts from the observation of the real effects produced by the execution of each action at each junction in the environment. Every M steps, MONITORING observes the executed action at each junction, the preconditions that hold at that time step, and, at the next MONITORING cycle, the effects after applying that action. Preconditions and effects are related to the density levels of the incoming and outgoing streets of the junction.

Then, the junctions' behavior set B is updated by function UPDATEB. It takes as input the current state of the

network in S and the last executed plan π, and updates the set B. B is composed of tuples b=$\langle j, a, \hat{p}, \hat{o}, f\rangle$ tuples, where $j \in J$ is a network junction; a is an action; $\hat{p}$=$\langle iN, iE, iS, iW, oN, oE, oS, oW\rangle$[2] is the preconditions vector with as many positions as street sections getting in (i) or out (o) of j, with their density values before applying the action; $\hat{o}$ is the effects vector with the same structure as $\hat{p}$, but with the observed density values after applying a; and f is the number of times that the same values for the tuple $\langle j, a, \hat{p}, \hat{o}\rangle$ have been observed. This data is used to compute the most likely effects after applying an action at a particular junction. We will follow the same determinization principles used by Yoon, Fern, and Givan, using only the most frequent effects o for each tuple $\langle j, a, \hat{p}\rangle$ in order to build the new actions. Other determinization schemes could be used in order to apply planning under uncertainty. Table 1 shows an example of a possible set of observations B. Given this data, when `oneHigh` (a) is executed with $\hat{p}$=$\langle h, l, l, l, l, l, l, l\rangle$, $\hat{o}_{a,\hat{p}}$=$\langle l, l, l, l, l, l, l, l\rangle$ are its most frequent effects.

Table 1: Example of B for a specific junction. It shows the frequency with which a set of effects' values $\hat{o}$ are observed after applying an action a with a given vector of preconditions' values $\hat{p}$. h is used for high density and l for low.

j	a	$\hat{p}$	$\hat{o}$	f
$j6$	oneHigh	$\langle h, l, l, l, l, l, l, l\rangle$	$\langle l, l, l, l, l, l, l, l\rangle$	9
$j6$	oneHigh	$\langle h, l, l, l, l, l, l, l\rangle$	$\langle l, l, l, l, l, h, l, l\rangle$	7
$j6$	twoHighIF	$\langle h, l, h, l, l, l, l, l\rangle$	$\langle l, l, l, l, l, l, l, l\rangle$	3
$j6$	twoHighIF	$\langle h, l, h, l, l, l, l, l\rangle$	$\langle l, l, l, l, h, l, l, l\rangle$	7

The MODEL LEARNER module takes as input the previous working domain D and a new set B generating as output a new domain D that includes new actions corresponding to the most frequent observed cases in B. If there is any previous learned action, it is removed from D. So, let us assume that, for each tuple $\langle j, a, \hat{p}\rangle$, $\langle j, a, \hat{p}, \hat{o}\rangle$ is the most frequently observed tuple. a is the action in the original domain D (where st(a) are its static preconditions), $\hat{p}$ are the observed dynamic preconditions, and $\hat{o}$ are the most frequent effects. Then, a new action a' is generated as: a'=$\langle$pre(a'), eff(a')$\rangle$. It does not have parameters, and pre(a')=st(a) $\wedge$ $\hat{p}$ and eff(a')=$\hat{o}$. Since this action is defined for a particular junction, a is maintained in the domain.

The new action is added to the domain model if no conflicts are found. We define a conflict between preconditions $\hat{p}$ of an action a and its most common effects $\hat{o}$ if: a) none of the input streets has high density; or b) when $\hat{p}$=$\hat{o}$. For example, a new action would not be added to the planning domain if $\hat{p}$=$\langle l, l, l, l, h, l, l, h\rangle$, or if $\hat{p}$=$\hat{o}$=$\langle h, l, l, l, l, l, l, l\rangle$.

An example of a learned action from B in Table 1 is shown in Figure 4. The action's preconditions are the conjunction of st(a) and $\hat{p}$. The effects reflect the most frequent density levels after applying a. Note that the action is completely instantiated, given that it refers to a specific junction and a set of specific preconditions. After learning the model, APTC has some new actions that represent the real

[2]This is an example of a junction with four cardinal directions (N, S, E, W).

```
(:action J6-twoHighIF-h-l-h-l-l-l-l-l
 :parameters ()
 :precondition (and (goes-into sin1 j54)
                    (goes-out sout4 j54)
                    (densityLevel sin1 high)
                    (densityLevel sout4 low)
                    ...)
 :effect (and (densityLevel sin1 low)
              (not (densityLevel sin1 high))
              ...))
```

Figure 4: Part of an example description of a learned PDDL action for action twoHighIF in junction J6 from Table 1.

behavior of each junction for a given time period, instead of using a common action that tries to describe the behavior of all junctions in the city at all time steps. Given that the domain model is dynamically learned, it can be adjusted to sudden changes in traffic conditions and automatically return to "normal" conditions when needed. This solves some of the problems that classical approaches face; dynamically adjusting to unexpected situations.

City Splitter

The second contribution of this paper is the use of distributed planning to help scaling up AP when solving UTC problems. The goal is to improve the system's performance and scalability by factoring the whole network into a set of areas $\mathcal{A}$. Each area $s \in \mathcal{A}$ is a subgraph of S such that given two areas s_1=(J_1, E_1), s_2=$(J_2, E_2) \in A$, $J_1 \cap J_2$=$\emptyset$. Therefore, areas have disjoint subsets of junctions. Some edges in the limits of an area can be shared with neighbour areas. These edges are connected to artificial vertices in each area that represent either the vertices outside the network or the junctions in another area. Thus, the same edge can appear in two different areas. Once, a new division in areas $\mathcal{A}$ is computed, APTC generates a set of problems, one per area, that are independent- and asynchronously solved by a planner. The resulting plans can be safely concatenated since they correspond to independent junctions.

The areas could be divided in regular areas following a naïve domain-dependent approach. However, as we show later in the experiments, this leads to a worse performance due to the loss of a property of traffic networks: emergent traffic flows. Since the densities are propagated through the effects of the actions, a bad city partition (split) will not correctly propagate those densities (flows) over different areas. Therefore "green waves" can not be handled.

We overcome this problem by detecting the vehicle flows, taking advantage of the continuous monitoring of the city traffic. APTC joins in a single area those streets and junctions that conform a flow. We say that two street sections with corresponding edges in E, e_1, e_2, are *connected* if there is a pair of junctions $j_1, j_2 \in J$ and: e_1 enters j_1; and e_2 leaves j_1 and enters j_2. If a street e_1 with high density at t_1 is connected with another street e_2 with low density at t_1, and e_2's density becomes high at the next time step t_2, there might exist a flow from junction j_1 to junction j_2. APTC stores these potential transitions in a set of flows F that is updated by function UPDATEF. Each element $fl \in F$ is a tuple fl=$\langle j_1, j_2, f\rangle$, where j_1 is the first junction that vehi-

cles traverse, j_2 the junction cars arrive at and f the number of times that this flow has been observed in the last simulation steps.

F will be used for possibly splitting the city every L steps of simulation. The complete set of tuples in F form a directed graph G=$(J', E') \subseteq S$=(J, E). J'=$\{j|(j, j', f) \in F \vee (j', j, f) \in F\}$; i.e. set of junctions that appear in tuples in F. And E'=$\{(j_1, j_2)|j_1, j_2 \in J, (j_1, j_2, f) \in F\}$ such that $w(e)$=f if e=$(j_1, j_2) \in E'$ and $(j_1, j_2, f) \in F$. All cycles are removed from G to make it acyclic by using the algorithm in (Johnson 1975). In order to maintain the junctions involved in a flow in the same area (and thus planning problem), we look for the maximum-length disjoint paths in G. This can be done in polinomial time, since G is directed and acyclic (Fortune, Hopcroft, and Wyllie 1980). We use the Python package NetworkX[3] for the computation of the maximum length paths as well as the removal of cycles. Algorithm 2 presents the procedure to divide the city.

Algorithm 2 CITY SPLITTER(F, S, N)

Inputs: Set of flows F, graph S, max junctions per problem N
Outputs: $\mathcal{A}$

$U \leftarrow$GETJUNCTIONS(S)
Flows $\leftarrow$MAXLENGTHPATHS(F)
$\mathcal{A} \leftarrow \emptyset$
for each flow **in** Flows **do**
  if `length`(flow)$\geq N$ **then**
    while flow $\neq \emptyset$ **do**
      $\mathcal{A} \leftarrow \mathcal{A} \cup \{\text{flow}[..\mathcal{N}]\}$
      $U \leftarrow U \setminus \{\text{flow}[..N]\}$
      flow $\leftarrow$ flow$[N..]$
  else
    while `length`(flow) $< N$ **do**
      flow $\leftarrow$ flow$\cup$NEIGHBOUR(flow, U)
    $\mathcal{A} \leftarrow \mathcal{A} \cup \{\text{flow}\}, U \leftarrow U \setminus \{\text{flow}\}$
while $U \neq \emptyset$ **do**
  $div \leftarrow \{\text{pop}(U)\}$
  while `length`(div) $< N$ **do**
    $k =$`neighbour`(div, U)
    $div \leftarrow div \cup \{k\}$
    $U \leftarrow U \setminus \{k\}$
  $\mathcal{A} \leftarrow \mathcal{A} \cup div\,, U \leftarrow U \setminus div$
return $\mathcal{A}$

For each flow returned by MAXLENGTHPATHS(F), a partition is generated if it has N elements, the maximum junctions per problem. This parameter affects the planning time and will be discussed in the next section. If the flow includes a higher number of junctions than N, the flow is separated. The first N junctions of the flow are inserted into $\mathcal{A}$ and the remaining junctions ($[N..]$) are the flow that needs to be splitted. If flow has a lower number of junctions than N, the function NEIGHBOUR(flow, U) inserts in flow a junction that is not already part of $\mathcal{A}$, $j \in U$ (the set of unassigned junctions) and it is connected to the last junction in flow. This is done until flow has N junctions. After all junctions in Flows have been assigned to an area and the area added to $\mathcal{A}$, the remaining junctions in U are separated, trying to group neighbour junctions. Finally, the algorithm returns $\mathcal{A}$, that MONITORING will use to generate the different problem files (one per area in $\mathcal{A}$).

[3] `https://networkx.github.io/`

Evaluation

We have used SUMO (Behrisch et al. 2011) for the experiments, an open source traffic simulator. It allows users to define networks, demand and traffic lights control programs. The evaluation is conducted in a grid network similar to the ones present in many cities. The network is composed of 100 junctions and 400 streets. We simulated five hours of a realistic city behavior. This scenario is described in Figure 5. The first 30 minutes of simulation introduces an input flow of cars following a uniform distribution in the city with an average frequency (a vehicle enters the city) of two seconds. With this frequency and distribution, none or just a few congestions are expected to occur. After that, there is a one hour period in which some cars enter the city by two fixed junctions and want to go to the work center. We do this in order to simulate the real behavior of a work day in which most of the people access their job through some main roads. Later, vehicles leave the work and others want to access the stadium in order to attend an sport event. This situation leads to big congestions and the peak of cars in the city is reached at that point. Finally, people leave the stadium and the simulation finishes.

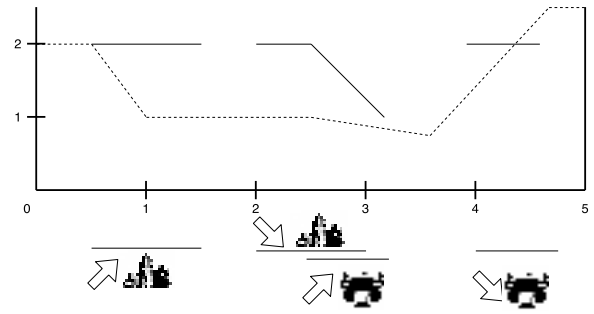


Figure 5: Simulation scenario. The x axis represents the time in hours and the y axis how many cars enter the city every second. The dotted line represents the frequency of cars with random start and destination. The continuous line represents the frequency of cars with fixed start and destination.

The total number of vehicles introduced in the city network is $17,400$ which is way beyond to what other AP-based UTC approaches have reported. For instance, Vallati et al. [2017] used 10 junctions, while our network comprises 100 junctions. Also, Gulič et al. [2016] introduced $5,000$ vehicles in their biggest experiments. Furthermore, most works have used shorter simulation steps than the one reported here.

We compare APTC with three other strategies: STATIC, REACTIVE and IAS (Gulič, Olivares, and Borrajo 2016). STATIC corresponds to the default system used by SUMO and it represents the standard one used in most cities. REACTIVE can be seen as a simplification of SCOOT that does not consider offsets of neighboring traffic lights. IAS does

not have any learning component and only calls the planner when a vehicle has been stopped for a long time. IAS stopped the simulation until a plan was found while APTC does not stop the simulation. We also compare APTC against the starting planning domain without any domain model learning to test whether updating the planning domain represents an improvement or not. We will refer to it as FIXED. Finally, all the AP-based approaches are run twice: one with a MANUAL division and another one with a FLOWS split, the one presented in this paper. We do it in order to test if our factoring criteria works well by itself. We have also tried to compare APTC with the PDDL+ representation employed in (Vallati et al. 2016). However, in our experiments their model is not able to scale up to the networks we use in the time limits we need. Unfortunately, we cannot compare our system against commercial products (e.g. SCOOT).

The values of the parameters used by APTC are the following. Monitoring ratio **M** is the number of steps (seconds) elapsed between two monitoring episodes. This parameter affects the length of the executed actions among others. It has been fixed to 30 seconds, a common cycle in the static traffic lights programs. Learning ratio **L** is the number of steps (seconds) elapsed between two learning episodes. This parameter determines how fast the system is going to react to the changes produced in the environment, updating the planning domain and problems. It has been fixed to 300 seconds (five minutes) in this experiment. We consider that five minutes is a reasonable time to react against changing traffic conditions in the real world. Maximum number of junctions per problem **N** has been fixed to five junctions since it is the maximum number of junctions that a car can traverse in 30 seconds (M) in the experimental city network. This number also allows us to have short planning times.

We use the following metrics: the total amount of CO_2 emitted; the total number of cars that arrive at their destination within the simulation time (DC); the average waiting time of each vehicle (AWT); and the average travel time (ATT). We report the percentage of improvement of each configuration against the base system, STATIC. All the experiments were ran on a Ubuntu machine with Intel Core i7-410U running at 2.00 GHz. All the systems using Automated Planning use Lama 2011 (Richter and Westphal 2010) with a time limit of 20 seconds in order to leave at least 10 seconds to execute the plan. Table 2 shows the results for the simulated scenario. As we can see, APTC Flows, outperforms the rest in all the measured metrics. It is able to reduce the pollution and the waiting and travel times of the vehicles in the city. It also virtually allows all vehicles to reach their destination. It means that the congestion after the end of the sport event has been successfully solved. APTC also outperforms FIXED, the version that does not update the planning domain, showing that learning the city dynamics and how the vehicles traverse the city is in fact a big advantage. Furthermore, the flow-based city split is better than manually dividing the city in equal areas regardless of the planning domain used.

APTC's ability to adapt and react to new traffic conditions is depicted in Figure 6. It shows how the number of new generated domain's actions increases when the traffic dynamics change during the simulation. After learning a traffic behavior, these changes decrease meaning that an effective D has been found for the current scenario.

	$C0_2$	AWT	ATT	DC
STATIC	2972	120	188	16355
REACTIVE	-2%	-15%	-7%	$+1\%$
IASMANUAL	-1%	-13%	-5%	$+1\%$
IASFLOWS	-3%	-16%	-13%	$+2\%$
FIXEDMANUAL	-4%	-21%	-15%	$+3\%$
FIXEDFLOWS	-7%	-25%	-16%	$+4\%$
APTCMANUAL	-11%	-30%	-18%	$+5\%$
APTCFLOWS	-13%	-33%	-20%	$+6\%$

Table 2: Percentage of improvement of each system with respect to the STATIC traffic lights program. AWT and ATT are given in seconds, while $C0_2$ is in kg.

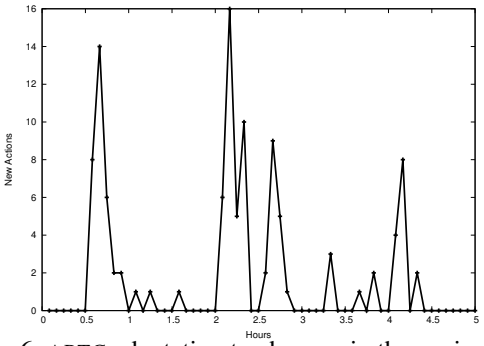


Figure 6: APTC adaptation to changes in the environment.

Conclusions and Future Work

In this paper we have presented APTC, an Automated Planning based system for UTC. As we have shown, there are two main problems when implementing planning systems in this kind of domain. One is related to the generation of accurate planning domains from scratch whose actions reflect the real world dynamics. Our proposal helps designing domain models as well as adapting the models to changes in the environment by updating the model through monitoring and learning. APTC learns the effects of the actions at a junction level and incorporates new actions in the domain. The second problem relates to scalability, where we propose a distributed approach. APTC divides the city according to the detected vehicle flows in order to generate "green waves". Using this approach the system is able to perform well even in large city networks one order of magnitude larger than the ones tested by other AP approaches. The flows-based city splitting criteria and the model learning, the two main contributions of the paper, can improve an AP system even if using them individually. Unifying both techniques we obtain a declarative system for UTC that performs better than other AP-based systems and the static and reactive approaches present in many cities.

In future work, we are interested on analyzing how the different parameters such as the monitoring and learning rates affect the system' performance. Continuous learning

as well as concept drift detection aspects (Gama et al. 2004) could be studied to improve the system' performance and adaptability.

Acknowledgements

This work has been partially supported by MINECO projects TIN2014-55637-C2-1-R and TIN2017-88476-C2-2-R and project PLICOGOR funded by Ministerio de Economía y Competitividad.

References

Behrisch, M.; Bieker, L.; Erdmann, J.; and Krajzewicz, D. 2011. Sumo–simulation of urban mobility. In *The Third International Conference on Advances in System Simulation (SIMUL 2011), Barcelona, Spain*.

Box, S., and Waterson, B. 2012. An automated signalized junction controller that learns strategies from a human expert. *Engineering applications of artificial intelligence* 25(1):107–118.

Bretherton, R.; Wood, K.; and Bowen, G. 1998. SCOOT version 4. In *Proceedings of 9th International Conference on Road Transport Information and Control*.

Cenamor, I.; Chrpa, L.; Jimoh, F.; McCluskey, T. L.; and Vallati, M. 2014. Planning & scheduling applications in urban traffic management.

Fortune, S.; Hopcroft, J.; and Wyllie, J. 1980. The directed subgraph homeomorphism problem. *Theoretical Computer Science* 10(2):111–121.

Fox, M., and Long, D. 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of AI Research* 20:61–124.

Fox, M., and Long, D. 2006. Modelling mixed discrete-continuous domains for planning. *J. Artif. Intell. Res.(JAIR)* 27:235–297.

Gama, J.; Medas, P.; Castillo, G.; and Rodrigues, P. 2004. Learning with drift detection. In *Brazilian Symposium on Artificial Intelligence*, 286–295. Springer.

García-Martínez, R., and Borrajo, D. 2000. An integrated approach of learning, planning, and execution. *Journal of Intelligent and Robotic Systems* 29(1):47–78.

Gulič, M.; Olivares, R.; and Borrajo, D. 2016. Using automated planning for traffic signals control. *PROMET - Traffic&Transportation* 28(4):383–391.

Guzmán, C.; Alcázar, V.; Prior, D.; Onaindía, E.; Borrajo, D.; Fdez-Olivares, J.; and Quintero, E. 2012. PELEA: a domain-independent architecture for planning, execution and learning. In *Proceedings of ICAPS'12 Scheduling and Planning Applications woRKshop (SPARK)*, 38–45. Atibaia (Brazil): AAAI Press.

Hamilton, A.; Waterson, B.; Cherrett, T.; Robinson, A.; and Snell, I. 2013. The evolution of urban traffic control: changing policy and technology. *Transportation planning and technology* 36(1):24–43.

Huebscher, M. C., and McCann, J. A. 2008. A survey of autonomic computing degrees, models, and applications. *ACM Computing Surveys (CSUR)* 40(3):7.

Jiménez, S.; de la Rosa, T.; Fernández, S.; Fernández, F.; and Borrajo, D. 2012. A review of machine learning for automated planning. *The Knowledge Engineering Review* 27(4):433–467.

Jiménez, S.; Fernández, F.; and Borrajo, D. 2013. Integrating planning, execution and learning to improve plan execution. *Computational Intelligence Journal* 29(1):1–36.

Jin, J., and Ma, X. 2017. A group-based traffic signal control with adaptive learning ability. *Engineering Applications of Artificial Intelligence* 65:282–293.

Johnson, D. B. 1975. Finding all the elementary circuits of a directed graph. *SIAM Journal on Computing* 4(1):77–84.

Keyder, E., and Geffner, H. 2009. Soft goals can be compiled away. *Journal of Artificial Intelligence Research* 36:547–556.

Martínez, D.; Alenya, G.; Torrás, C.; Ribeiro, T.; and Inoue, K. 2016. Learning relational dynamics of stochastic domains for planning. In *Proceedings of the 26th International Conference on Automated Planning and Scheduling*.

McCluskey, T., and Vallati, M. 2017. Embedding automated planning within urban traffic management operations. In *Proceedings of the 27th International COnference on Automated Planning and Scheduling (ICAPS-17)*.

Mourao, K. 2014. Learning probabilistic planning operators from noisy observations. In *Proc. of the Workshop of the UK Planning and Scheduling Special Interest Group*.

Papageorgiou, M.; Ben-Akiva, M.; Bottom, J.; Bovy, P. H.; Hoogendoorn, S.; Hounsell, N. B.; Kotsialos, A.; and McDonald, M. 2007. Its and traffic management. *Handbooks in Operations Research and Management Science* 14:715–774.

Pasula, H. M.; Zettlemoyer, L. S.; and Kaelbling, L. P. 2007. Learning symbolic models of stochastic domains. *Journal of Artificial Intelligence Research* 29:309–352.

Richter, S., and Westphal, M. 2010. The LAMA planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research* 39(1):127–177.

Treiber, M., and Kesting, A. 2013. Traffic flow dynamics. *Traffic Flow Dynamics: Data, Models and Simulation, Springer-Verlag Berlin Heidelberg*.

Vallati, M.; Magazzeni, D.; Schutter, B. D.; Chrpa, L.; and McCluskey, T. 2016. Efficient macroscopic urban traffic models for reducing congestion: a pddl+ planning approach. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI-16)*.

Xie, X.-F.; Smith, S. F.; and Barlow, G. J. 2012. Schedule-driven coordination for real-time traffic network control. In *ICAPS*.

Yoon, S.; Fern, A.; and Givan, R. 2007. FF-replan: A baseline for probabilistic planning. In *ICAPS*, 352–360.

Towards a Framework for Understanding and Assessing Quality Aspects of Automated Planning Models

Mauro Vallati and **Thomas L. McCluskey**
PARK research group, University of Huddersfield
Queensgate, HD13DH, Huddersfield
United Kingdom
n.surname@hud.ac.uk

Abstract

A crucial aspect of automated planning is the knowledge model. It is used by the automated planning engines in order to generate solution plans. Despite the fact that the quality of the model has a strong influence on the resulting planning application, the notion of quality for planning models is not well understood, and the engineering process in building such models is still mainly an ad-hoc process.

In order to replace ad-hoc processes and to support a more comprehensive notion of quality, this paper introduces a quality framework specifically focused on automated planning models.

Introduction

Planning knowledge models are conceptual models, in that they are explicit (and formal) representations of some proportions of reality as perceived by some actor (Wegner and Goldin 1999). These models may contain representations of objects, relations, properties, functions, resources, actions, events and processes, in the application domain. There are significant differences between generic conceptual models and planning knowledge models, however, in that the planning model is aimed more for its operational value than for its use in interactions and communications with domain experts and other stakeholders.

Up to now there has been no overall framework for considering the quality of the various components involved in the life cycle of the planning knowledge model. There has been research into the quality of planning applications in terms of verification and validation (Frank 2013), and in terms of accuracy and completeness of the knowledge model (McCluskey, Vaquero, and Vallati 2017), but no overall conceptual model covering the many aspects of such models.

In this paper, building on existing frameworks proposed for general conceptual models (see, e.g., (Lindland, Sindre, and Sølvberg 1994; Krogstie 2012; Krogstie, Sindre, and Jørgensen 2006)), we introduce a quality framework specifically focused on automated planning models. The main benefit of a framework is to replace ad-hoc notions of quality, and ad-hoc knowledge engineering processes, with a connected, composite and over-arching notion, that can be used within all work relating to this endeavour. The proposed framework is exploited for introducing and describing specialised aspects of quality of planning models.

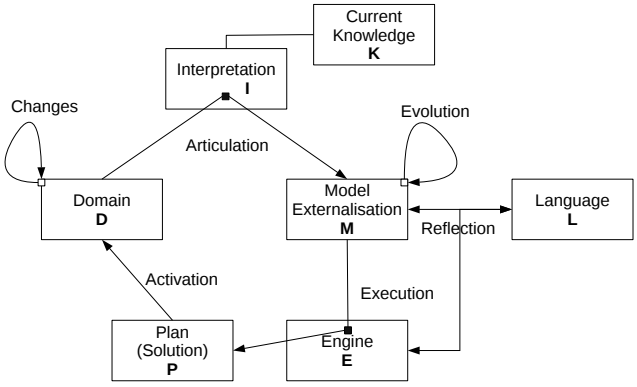


Figure 1: The proposed quality framework for planning models.

Quality Framework for Planning Models

The framework aims at representing all the aspects that affect the quality of knowledge planning models. Figure 1 presents the basic ideas of the quality framework. The framework considers different sets, and processes. The following sets are introduced:

- L represents the language that is used to encode the model. This can be, for instance, a version of PDDL. In this context, the language is expected to have a well-defined syntax, vocabulary, and operational semantics.
- D is the domain specification, a set of requirements for the application domain at hand.
- M represents the model externalisation in a language L, that is, a formal specification of the application domain part of the requirements specification which represents entities invariant over every problem instance, such as object classes, functions, properties, relations, and operators (McCluskey, Vaquero, and Vallati 2017), as well as the specification of problem instances that has to be reasoned upon by the planning engine.
- I is the interpretation, that is, the internal specification of either a human (expert) or an automated technique of the domain requirements D, that allows to generate the model M, in the selected language L.

- K represents the relevant explicit current knowledge, represented in formal or semi-formal format, that is available with regards to the modelling of the domain.
- E is the algorithm exploited by a planning engine in order to generate, given the model externalisation of the domain and of a problem, a solution plan P.
- Finally, P stands for the solution plans that can be obtained, using the engine E on the model externalisation M.

Processes in Figure 1 represent interactions between sets, that lead to changes in one (or more) of the involved sets. In our framework, processes have been named following, to some extent, the existing nomenclature proposed in (Krogstie, Sindre, and Jørgensen 2006), appropriately extended and modified for the sake of dealing with planning domain and problem models.

Articulation ($D \rightarrow M$) is the process where the domain D is encoded as a model M by means of a specific language L. The articulation is performed by an interpreter, on the basis of her interpretation I of the domain, and of the available knowledge K.

The *Reflection* process stands for the impact that a language L has on the model externalisation, as well as on the planning engine E. The impact on the model is extremely intuitive: different languages provide different expressive power, and different ways for formalising the relevant specification requirements of the domain. The language has a strong impact on the planning engine, as different engines support different languages, or different subsets of the language's features. Furthermore, similar dynamics can be differently encoded in different languages, with a potentially different impact on the operationality of planning engines. In this context, operationality refers to the ability of an engine to deliver a solution plan given some resource constraints, that are specified in the domain specification.

Execution ($M \rightarrow E \rightarrow P$) is the process of generating solution plans, by proving as input of the planning engine E the model externalisation M.

Activation ($P \rightarrow D$) captures the changes that the use of the model may trigger in the domain specification D. This focuses on refinement in the specifications that the use of planning highlighted.

The *changes ($D \rightarrow D$)* process incorporates changes to the specification that are due to environmental variables. This can be the case of a logistic company that decides to extended its transport fleet by including different kind of vehicles. As it is apparent, such a decision would change the domain specification, but not because of any automated planning-related aspects.

The *evolution ($M \rightarrow M$)* process focuses on the evolution of M. This can be the result of improvements in the current knowledge K or, for instance, because of the modification of the interpretation I due to a better understanding of the specifications. Evolution can also be due to the use of reformulation techniques, that change the model externalisation in order to improve the performance of the engine E.

Quality Concepts

A single general quality notion has been suggested by a large strand of previous work in the AI planning area (McCluskey, Vaquero, and Vallati 2017; McCluskey 2002). However, as pointed out by the SEQUAL framework (Lindland, Sindre, and Sølvberg 1994), a general quality notion cannot be directly evaluated nor measured. It is therefore pivotal to introduce a number of quality dimensions, that include aspects and elements that is possible to measure and analyse. Quality levels for conceptual models are usually defined following the semiotic ladder (Stamper 1996). The semiotic ladder introduced six levels, corresponding to different dimensions (either related to the IT platform or to the human society).

In the following we specialise the main quality types in order to fit the needs of planning models, and expected users and knowledge engineers. Noteworthy, due to the inner aims of planning models –that are not mainly focused on communicating knowledge, but on allowing the generation of solution plans–, we have to introduce quality aspects that are not covered in the semiotic ladder, and to drop some aspects that are not particularly relevant for the planning models' purposes. Quality aspects separate the goals, which represent what this aspect is trying to assess and maximise, from the means for achieving such goals.

Physical Quality: following the existing literature, this quality has two main goals, the externalisation and the internalisability. The former refers to the fact that the model M is an artefact, resulting from the externalisation (in other words, of making explicit) of the interpretation knowledge I of an interpreter, and is based also on the available current knowledge K. Externalisation also covers the fact that the considered application domain can be represented under the form of some symbolic model, specifically using available planning-oriented languages. The internalisability stands for the fact that the model M is persistent and available to interpreters that can understand it and interpret it, and can be used by an appropriate engine E to generate solutions. In other words, the internalisability focuses on the fact that the model is available for the planning engine, as well as to experts that may need to check or revise it. At a first glance this may seem trivial in the typical planning settings, particularly for a domain model. However, in terms of problem models, it may be the case that such problems are automatically generated by combining information gathered from different sources (sensors, data bases, etc.), and the process may be hard to reproduce.

Semantic Quality aims at the goals of accuracy and completeness. In this context, we rely on the definitions provided by McCluskey, Vaquero, and Vallati (2017). Accuracy is focused on relating the model M and the domain specification D, by ensuring that M is a valid representation of the specification, i.e. it encodes all the aspects that are correct and relevant for the domain. Conversely, completeness involves the solution plans P in that it means that M allows to generate all (and only) solution plans that are correct with regards to the domain specification D.

Pragmatic Quality: this covers how the model externalisation is activated, i.e. the way in which the exploitation of the model, maybe within a larger framework, can affect

the domain specification and, in a broader sense, the application domain itself. In principle, activation and articulation can be seen as a co-design cycle, where feedbacks allow in turn to improve the overall domain understanding and specification, and to evolve and refine the model externalisation, with potential impacts on the efficiency of engines and on the interpretation and current knowledge.

Syntactic Quality is probably the easiest quality aspect that can be measured and assessed, as it aims at the syntactic correctness of the model M with regards to the selected modelling language L. It is important to remark that planning engines E may add additional constraints on the syntax of the language, due to partial support of some language features, for instance. The planning engine can not be selected in isolation: the language and the planning engine are affecting each other and, of course, decisions taken with regards to E and L have repercussions on the rest of the modelling process. For this reason, it is crucial to include also the engine in the analysis of the syntactic quality.

Operational Quality: covers the ability of the selected planning engine E to reason upon the model externalisation M to generate P. This quality aspect incorporates two perspectives. (i) The shape of solution plans that E allows to generate. On this matter there may be preferences in terms of number of actions involved, or makespan, or cost of the actions that are considered. It may also be the case that, for the specific application domain, only *optimal* solution plans are acceptable. (ii) The resource bounds that can be used by E to solve a problem instance. In this context, acceptable resource bounds can be defined –instance– in terms of runtime, memory usage, number of CPUs, etc. Resource bounds can be specified in the domain specification D, or may be derived by the interpretation I, or by the current shared knowledge.

Conclusions and Future Work

Knowledge engineering for automated planning is of great importance to foster the exploitation of planning techniques in real-world applications. A deeper understanding of how quality is involved in the development process of knowledge models, as well as a notion of quality for models, needs to be derived. In this paper we have briefly introduced a quality framework for use in viewing the development of the knowledge model in automated planning. For future work, we encourage studies which demonstrate in more concrete terms the benefits of using the framework.

References

Frank, J. 2013. The challenges of verification and validation of automated planning systems (keynote). *28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

Krogstie, J.; Sindre, G.; and Jørgensen, H. D. 2006. Process models representing knowledge for action: a revised quality framework. *EJIS* 15(1):91–102.

Krogstie, J. 2012. Quality of business process models. In *The Practice of Enterprise Modeling - 5th IFIP WG 8.1 Working Conference, PoEM*, 76–90.

Lindland, O. I.; Sindre, G.; and Sølvberg, A. 1994. Understanding quality in conceptual modeling. *IEEE Software* 11(2):42–49.

McCluskey, T. L.; Vaquero, T. S.; and Vallati, M. 2017. Engineering knowledge for automated planning: Towards a notion of quality. In *Proceedings of K-CAP*, 14:1–14:8.

McCluskey, T. L. 2002. Knowledge Engineering: Issues for the AI Planning Community (Keynote Talk). In *Proceedings of the KEPS Workshop*.

Stamper, R. 1996. Signs, information, norms and systems. *Signs of work* 349–399.

Wegner, P., and Goldin, D. 1999. *Interaction as a Framework for Modeling*. 243–257.

LOUGA: Learning Planning Operators using Genetic Algorithms

Jiří Kučera and **Roman Barták**
Charles University, Faculty of Mathematics and Physics
Malostranské nám. 25, Praha 1, Czech Republic

Abstract

We propose a novel method for learning planning operators (action schemata) from example plans. This method, called LOUGA (Learning Operators Using Genetic Algorithms), uses a genetic algorithm to learn action effects and an ad-hoc algorithm to learn action preconditions. We show experimentally that LOUGA is more accurate and faster than the ARMS system, currently the only technique for solving the same type of problem.

Introduction

Automated planning deals with the problem of finding a sequence of actions that transfer the world from the current state to a desired state. It is a model-based method, where the model formally describes how the actions are changing states of the world. Hence an important aspect of automated planning is obtaining a proper model of actions. In this paper we deal with classical (STRIPS) planning where actions are defined via preconditions and postconditions (effects), each being a set of predicates. The problem that we are addressing in the paper is how to learn these sets of preconditions and postconditions automatically from examples of plans.

There exist various approaches to acquisition of planning domain models. The early works such as EXPO (Gil 1994) or later works such as STRIPS-TraceLearn (Shahaf, Chang, and Amir 2006) improve action models incrementally after observing some problem during plan execution. Another approach learns from expert traces and subsequent simulations (Wang 1995). Frequently, the acquisition problem consists of finding the domain model from examples of plans, which is also the topic of this paper. In other words, the problem is to learn a correct state transition function according to observed sequences of actions and states. The system ARMS (Yang, Wu, and Jiang 2007) uses partially specified plans as its input, namely each plan consists of the initial state, a sequence actions, and goal predicates. Intermediate states might also be partially specified. Using MAX-SAT, ARMS learns the preconditions and effects of actions. The follower of ARMS called AMAN (Zhuo and Kambhampati 2013) allows some actions in plans to be wrongly recognized. LOCM (Cresswell, McCluskey, and West 2009) and LOCM2 (Cresswell and Gregory 2011) do not use a predicate model of world states but they rather learn finite-state automata for objects in the world. These automata describe how properties of objects are being changed by actions. Similarly, Opmaker2 (McCluskey et al. 2009) learns actions as methods to change properties (states) of involved objects and it requires some invariant formulas describing propositions that must be true in any state. ASCoL (Jilani et al. 2015) and LC_M (Gregory, Lindsay, and Porteous 2017) both only extend the LOCM system. ASCoL learns static preconditions for LOCM and LC_M extends it to work with missing and noisy data. Finally LAMP (Zhuo et al. 2010) learns more complex action models with quantifiers and logical implications.

We solve the same problem as ARMS, but we relax the condition of knowing the goal predicates. The proposed system LOUGA (Learning Operators Using Genetic Algorithms) learns from valid sequences of actions (not necessarily from plans reaching certain goals as ARMS). We assume that the initial state and a valid sequence of actions is given as input. LOUGA can also exploit partially specified intermediate states and a final state to find more accurate models. We use a classical genetic algorithm to learn the effects of actions and an ad-hoc algorithm to learn the preconditions of actions. We show experimentally that LOUGA produces more accurate domain models and it is faster than ARMS.

Background and Problem Specification

We work with classical STRIPS planning that deals with sequences of actions transferring the world from a given initial state to a state satisfying certain goal condition. World states are modeled as sets of predicates that are true in those states and actions are changing validity of certain predicates.

Formally, let P be a set of all predicates modeling properties of world states. Then a state $S \subseteq P$ is a set of predicates that are true in that state (every other predicate is false). Each action a is described by four sets of predicates $(B_a^+, B_a^-, A_a^+, A_a^-)$, where $B_a^+, B_a^-, A_a^+, A_a^- \subseteq P, B_a^+ \cap B_a^- = \emptyset, A_a^+ \cap A_a^- = \emptyset$. Sets B_a^+ and B_a^- describe positive and negative preconditions of action a, that is, predicates that must be true and false right before the action a. Action a is applicable to state S iff $B_a^+ \subseteq S \wedge B_a^- \cap S = \emptyset$. Sets $A_a^+$and A_a^- describe positive (add list) and negative (del list) effects of action a, that is, predicates that will become true and false in the state right after executing the action a. If an action a is applicable to state S then the state right after the action a will be $\gamma(S, a) = (S \setminus A_a^-) \cup A_a^+$. If an action

a is not applicable to state S then $\gamma(S, a)$ is undefined. In this work we use additional assumptions about the applicability of actions, namely $A_a^- \subseteq S$ and $A_a^+ \cap S = \emptyset$. The first assumption says that if an action deletes some predicate from the state then this predicate should be present in the state. Similarly, if an action adds some predicate to the state then the predicate should not be in the state before. These assumptions can be easily included in the action model as $A_a^- \subseteq B_a^+$ and $A_a^+ \subseteq B_a^-$.

In practice, operators are used in the domain model rather than actions. Operator can be seen as a parameterized action. Each operator has a set of attributes and specifies preconditions and effects as predicates over these attributes:

```
(:action move
   :parameters (?o - object ?m - place
                ?l - place)
   :precondition (at ?o ?m)
   :effect (and (at ?o ?l)
                (not (at ?o ?m))))
```

Actions are obtained by substituting constants for the attributes. The planning domain model is then specified by the set of predicates and the set of operators. PDDL modeling language (McDermott et al. 1998) is the most widely used language for modeling planning domains; we will use syntax of that language in our examples.

In our approach, we assume two types of input information. First, there is a partial planning domain model consisting of a set of predicates and a set of operators with attributes but without the description of preconditions and effects. The second type of input is a set of plans, where each plan consists of the initial state and a valid sequence of actions. Partially specified intermediate states or a goal state might also be provided. The information about states can be in three forms: a predicate was observed in the state, a predicate was observed not to be in the state, or the state was fully observed. We do not make any other assumptions about the input data unlike ARMS that presumes that some effect of every action is used by some later action or in the goal state. The task is to complete the domain model by learning preconditions and effects of operators such that the provided input plans are valid plans according to this domain model.

LOUGA

The proposed learning approach works in two main stages. First, we will learn action effects using a standard genetic algorithm (Mitchell 1998) with some extensions. Second, we will complete the learned action model by learning action preconditions using a polynomial ad-hoc algorithm.

For the genetic algorithm, we need to encode action effects to a genome. We will show, how to reduce the number of possible genomes by eliminating those violating conditions imposed on action effects. We will also define the fitness function that guides the genetic algorithm and we will show some methods to help the genetic algorithm when being stuck in a local optima. Next, we will show that it is possible to learn the effects predicate by predicate rather than all together. Finally, we will present the method for learning action preconditions.

Genome model

Genetic algorithms work with individuals, each individual encoding a solution candidate. In our case, an individual describes effects of operators. First, we generate a list of all operator-predicate pairs such that the operator can use the predicate in its add or delete lists. This property can be easily verified by checking that all attributes of the predicate are among the attributes of the operator. We assume that attributes are typed though this assumption can be relaxed as we will show in the section on experiments. Each operator-predicate pair will be associated with one of three values:

- 0: predicate is not in operator's add and del lists,
- 1: predicate is in operator's add list (positive effect),
- 2: predicate is in operator's delete list (negative effect).

The individual will be the sequence of numbers that corresponds one-to-one to the description of effects of operators.

For example, let us have a model with the following predicates and operators:

```
(:predicates
  (at ?o - (either object briefcase)
      ?l - place)
  (empty ?b - briefcase)
  (free ?o - object)
  (in ?o - object ?b - briefcase)
)
(:action move
  :parameters (?b - briefcase ?m - place
               ?l - place)
)
(:action put-in
  :parameters (?o - object ?p - place
               ?b - briefcase)
)
```

For this model, LOUGA generates the following pairs:

```
1. ((at ?b ?m), (move ?b ?m ?l))
2. ((at ?b ?l), (move ?b ?m ?l))
3. ((empty ?b), (move ?b ?m ?l))
4. ((at ?o ?p), (put-in ?o ?p ?b))
5. ((at ?b ?p), (put-in ?o ?p ?b))
6. ((empty ?b), (put-in ?o ?p ?b))
7. ((free ?o),  (put-in ?o ?p ?b))
8. ((in ?o ?b), (put-in ?o ?p ?b))
```

Hence, each individual will be described by a list of length eight. For example, the individual with genome '210 11000' corresponds to the model in which operator $(move\ ?b\ ?m\ ?l)$ has predicate $(at\ ?b\ ?m)$ in its del list and predicate $(at\ ?b\ ?l)$ in its add list and operator $(put\text{-}in\ ?o\ ?p\ ?b)$ has predicates $(at\ ?o\ ?p)$ and $(at\ ?b\ ?p)$ in its add list.

Note that it is possible to encode operator's preconditions in the same way, but we will present a more efficient method to learn operator's preconditions later.

Pre-processing

The genome model specifies the search space that the genetic algorithm will explore. We can reduce this space further by eliminating individuals violating constraints of the

model. This is done by exploring the example plans and identifying predicates that cannot be present in the add or del lists of specific operators. LOUGA simulates execution of the plan and for each state it finds two sets of predicates: the first set contains predicates that are definitely in the current state and the second set contains predicates that can possibly be in the current state, but it is not certain. Algorithm 1 describes how these sets are constructed and used.

Algorithm 1 Removing possible values of some genes.

Input: plan P; array M representing possible values of genes
Output: modified array M

1: $Q \leftarrow$ predicates from initial state
2: R - empty set of predicates
3: **for all** actions a from P **do**
4: generate a set of predicates X, which a can use
5: **for all** $p \in X$ **do**
6: **if** $p \in Q$ **then** ▷ p is in the current state
7: M[(a, p),add] = false
8: $R = R \cup \{p\}$
9: $Q = Q \setminus \{p\}$
10: **else if** $p \notin R$ **then** ▷ p is not in the state
11: M[(a, p),del] = false
12: $R = R \cup \{p\}$
13: **end if**
14: **if** predicates S were observed after a **then**
15: $Q = Q \cup S$
16: $R = R \setminus S$
17: **end if**
18: **if** preds. S were observed missing after a **then**
19: $Q = Q \setminus S$
20: $R = R \setminus S$
21: **end if**
22: **end for**
23: **end for**

Initially, the first set Q contains all predicates from the initial state (line 1) and the second set R is empty (line 2). LOUGA then goes through the actions in the order specified by the plan. For each action it generates the set of all predicates that the action can use. If some predicate is present in the state before the action, LOUGA marks that the action cannot have that predicate in its add list (line 7). If a predicate is definitely not present in that state, LOUGA marks that the action cannot delete it (line 11). All predicates generated for the action are then added to the second set and removed from the first one if they were present in it. If there are some predicates observed in the state after the action, all of these predicates are added to the first set and removed from the second one. After that, LOUGA continues with the next action until it processes the whole plan. The justification of this process is as follows. If some predicate can be modified by the action then that predicate can possibly be part of the next state. If some predicate is in the state and it is not modified by the action then the predicate stays in the state. Also, information about observed predicates can be exploited there (lines 14-21).

For example let us assume this short plan:

```
(:state
  (empty b1)
  (at b1 home)
  (free pencil)
  (at pencil home)
  (at rubber home)
  (free rubber))
(put-in pencil home b1)
(move b1 home office)
```

We know that there are exactly six predicates in the initial state. Action ($put\text{-}in\ pencil\ home\ b1$) can work with predicates ($at\ pencil\ home$), ($at\ b1\ home$), ($empty\ b1$), ($free\ pencil$) and ($in\ pencil\ b1$). Pairs made of these predicates and operator $put\text{-}in$ correspond to genes 4-8. Predicates ($at\ pencil\ home$), ($at\ b1\ home$), ($empty\ b1$) and ($free\ pencil$) are definitely present in the state before action $put\text{-}in$, which means that the action cannot add them. As a result, genes 4-7 will have disabled value 1 during evolution. Predicate ($in\ pencil\ b1$) is not in the initial state, which means that the action cannot delete it and therefore gene 8 will have disabled value 2 during evolution.

After processing the action we move all these predicates to the second set (line 8) and delete them from the first set (line 9), if they are present there. Now the sets contain these predicates:

```
first set Q (definitely in the state)
  (at rubber home)
  (free rubber)
second set R (possibly in the state)
  (empty b1)
  (at b1 home)
  (free pencil)
  (at pencil home)
  (in pencil b1)
```

The next action is ($move\ b1\ home\ office$). Operator $move$ corresponds to genes 1-3, that means that the action can use predicates ($at\ b1\ home$), ($at\ b1\ office$) and ($empty\ b1$). Predicate ($at\ b1\ office$) is in none of the sets, therefore the action cannot delete it and gene 2 cannot have value 2. Other predicates are already in the second set, so genes 1 and 3 will remain unchanged. We add predicate ($at\ b1\ office$) to the second set and continue with the next action (if any).

Fitness function

The genetic algorithm uses a fitness function which evaluates the error rate of the model represented by the individual. We assume three types of errors:

- **add error**: an action tries to add a predicate that is already present in the world state,
- **del error**: an action tries to delete a predicate that is not currently present in the world state,
- **observation error**: a predicate was observed in a state in the original plan, but it is not present in the corresponding state of the plan executed according to the current model, or there is a predicate in a state that should not be present according to observations about the corresponding state in the original plan.

Formally we can define these errors as follows: let *S* be a state of a plan executed according to the model represented by the individual, *T* be a set of predicates that were observed in the corresponding state in the input plan, *N* be a set of predicates that were observed not to be present in the corresponding state, a be the action performed from state *S* and $p \in A_a^+$, $q \in A_a^-$, $s \in S$ and $t \in T$ be some predicates. Add error occurs when $p \in S$, del error occurs when $q \notin S$, and observation error occurs when $t \notin S$, $s \in N$ or – if *T* was marked as a fully-observed state – when $s \notin T$.

After all plans are processed, the fitness value of the individual is defined using this formula:

$$(1 - (error_{add} + error_{del})/(total_{add} + total_{del}))* \\ (1 - error_{obs}/total_{obs}), \quad (1)$$

where $error_{add}$, $error_{del}$ and $error_{obs}$ are the numbers of corresponding errors, $total_{add}$ and $total_{del}$ are the numbers of add and delete operations performed in simulation, $total_{obs}$ is the total number of observations about intermediate and goal states plus the number of surplus predicates in fully-observed states.

We tried a version of the fitness function that treated all three types of errors identically, but it turned out not to be ideal. When there were too many or too few observations in input data, evolution could get stuck in a local optima that favors good add and delete error rates over the observation error rate or vice versa. Treating observation errors separately solves this problem. We also tried to split the add and delete errors, but that had a marginal effect on efficiency. Obviously, the fitness value 1 means a perfect individual.

The genetic algorithm

LOUGA uses a classical genetic algorithm with one-point crossover and mutation (Mitchell 1998). We tried more sophisticated versions of those operators but we did not find any that would perform significantly better than the standard versions. We extended the standard algorithm by two additional operators applied when the population stagnates for some time (i.e. when the best individual is of certain age).

The first operator is basically 1-step local search starting from the best individual's genome to find all options how to change one gene to get a better individual. Genes are picked one by one and for every gene, every possible value is tried and resulting individuals are evaluated. As every gene has at most three possible values, there are at most $2 * N$ candidate genomes, where N is the length of genome. All individuals that performed better than the current best individual are added to the population.

The second operator is applied when even the local search cannot find a better individual. It stores the best individual and restarts the population. Next time before restarting it tries to use the information from previous runs by crossing the current best individual with the stored genomes from previous runs. If it breaks the stagnation, evolution goes on as before until it starts stagnating again or a perfect individual is found. If the algorithm cannot find a better local optimum after multiple restarts, the operator deletes the local optima list and the genetic algorithm starts from scratch.

Learning effects predicate by predicate

In complex[1] domain models, individuals' genomes can be too long for the genetic algorithm to work effectively. LOUGA solves this problem by learning operators' lists separately for each predicate type. It means that an instance of the genetic algorithm is run for each predicate type separately. In each instance, genomes are built only from those operator-predicate pairs that use the correct predicate type and the fitness function ignores observations of predicates of types other than the current predicate type.

This method generates the same genome as if all predicates were learned at once, the learning process is only split into multiple parts. These parts are independent to each other because occurrence of a predicate of one type cannot affect whether occurrence of a predicate of another type is incorrect or not. Therefore this method is sound and yields the same outcome as the standard approach.

Algorithm 2 Generation of precondition lists

Input: genome G; set of input plans Q
Output: model M

```
create model M represented by G
Y, N - integer fields indexed similarly as G; all fields initially 0
for all P ∈ Q do
    s ← initial state of P
    for all a ∈ P do
        for all pred. p, which can be generated by a do
            g ← index of gene corresponding to (p,a)
            if p ∈ s then
                Y[g]++
            else
                N[g]++
            end if
        end for
        s ← s after performing a according to M
    end for
end for
for all gene g ∈ G; g corresponds to predicate p and operator o do
    if G[g] = 0 then
        if N[g] = 0 && Y[g] > 0 then
            add p to pre_o
        else if N[g] > 0 && Y[g] = 0 then
            add (not p) to pre_o
        end if
    else if G[g] = 1 then
        add (not p) to pre_o
    else if G[g] = 2 then
        add p to pre_o
    end if
end for
```

[1] As 'complex' models we consider models that have a large number of predicate types and operators. Such models usually have long genomes so the genetic algorithm has to search through a large hypothesis space.

Learning preconditions

After the add and del sets are learned, the sets of preconditions (including preconditions due to static facts) are generated. LOUGA goes through every plan and for every operator and every relevant predicate it counts the number of cases where the predicate was present before the action was performed and the number of cases where it was not present. After every plan is processed, a positive precondition is created for every such pair that the predicate was always present before the action was performed, and a negative precondition is created for every such pair that the predicate was never present before the action was performed. If evolution gives a perfect individual, this method yields proper precondition lists. Algorithm 2 describes this process formally.

Results of Experiments

We evaluated experimentally the contribution of components of LOUGA and we compared LOUGA to ARMS, which is the only other technique solving the same problems. All experiments were run on laptop with Intel Core i5-2410 2.3GHz processor and 8GB of RAM. We used five classical domains from planning competitions, namely Blocksworld, Briefcase, Flat-tyre, Rover, and Freecell. Their basic characteristics are given in Table 1.

	Briefcase	Blocksworld	FlatTyre	Rover	Freecell
# object types	3	1	5	7	3
# predicate types	4	5	12	25	11
# operators	3	4	13	9	10
Avg. size of effect lists	3.3	4.5	2	3	5.6
Avg. parameters of operators	2.66	1.5	2.33	4	4.9

Table 1: Comparison of domain models used in experiments.

For each experiment, we randomly generated 200 valid sequences of actions (plans) and performed five-fold cross-validation test by splitting them in five equal parts and running algorithms five times. During each run we used four groups as learning data and the fifth group as a test set. Plans generated for the first three domains had usually 5-8 actions. For domains Rover and Freecell, we generated random walks (without a preset goal) that had about 15-20 actions.

Most tables show runtimes and error rates of generated models (smaller numbers are better). We define errors in similar way as described in the section about fitness function of LOUGA. Add and delete error rates are calculated by dividing the number of errors by the number of performed add or delete actions. Since ARMS does not work with negative observations, we only evaluate fulfillment of those observations that state which predicates were definitely present in state. Observation error rate is therefore calculated by dividing the number of unfulfilled observations by the total number of predicates observed in intermediate and final states.

The size of population was set to 10, the threshold for local search was set to 7, the threshold for crossover with individuals from previous runs to 10, the threshold for population restart was 15 and mutation probability was 5% with 10% chance for a gene to be switched. From our internal tests we saw that benefits of having bigger population do not outweigh the longer computational time, so we kept the population sizes low. Keeping thresholds high did not provide much benefit neither, because the population did not usually break stagnation in reasonable time anyway.

	Pred. by pred.	Basic version
Briefcase	0.22 ± 0.15	0.86 ± 0.64
Blocksworld	0.81 ± 0.66	22.78 ± 40.38
FlatTyre	2.26 ± 0.74	111.76 ± 116.06
Rover	4.11 ± 0.33	$\gg 600$
Freecell	5.73 ± 1.1	$\gg 600$

Table 2: Performance of LOUGA learning a model predicate by predicate compared to basic version (runtime in seconds with standard deviation).

Efficiency of predicate by predicate approach

In the first experiment, we will show the effect of splitting the learning problem to multiple smaller problems, where each predicate is learned separately. Both versions of LOUGA reliably find flawless solutions, so we present the runtimes only (Table 2).

As expected, the predicate-by-predicate mode performs significantly faster than the basic version. Moreover as the standard deviation indicates, the predicate-by-predicate mode is also more stable. Runtimes of the basic version varied greatly, some runs were even 10 times longer than others. Genetic algorithms usually suffer from such behavior because of the randomness of the method. The predicate-by-predicate mode works more consistently thanks to evolution having a clearer direction. We can say that it generates only one property at a time even though it is composed of many genes. The basic version works with all properties together and improvement in one direction can go hand in hand with step back in other.

Comparison of GA and hill climbing

In the second experiment, we compared the genetic algorithm with the hill climbing approach. In particular, we compared three setups:

- **LOUGA** - the standard version with 10 individuals
- **HC** - hill climbing with 1 individual and local search in every generation; restart when stuck in local optimum
- **GA** - genetic algorithm without local search operator

We performed tests on four different inputs from the Blocksworld, Flat tyre and Freecell domains. For the Flat tyre domain, we used inputs with only goal predicates in ending states, so the problem had many solutions. In the other domains, we used plans with complete initial and ending states.

The results (Table 3) show that the genetic algorithm without the local search operator performs much worse than the

	LOUGA	HC	GA
Briefcase	0.22	0.88	0.36
Blocksworld	0.81	2.22	1.78
Flat tyre (ambiguious)	2.26	1.82	4.2
Rover	4.11	6.92	34.54
Freecell	5.73	11.11	51.52

Table 3: Runtimes [s] of LOUGA, hill-climbing and a genetic algorithm.

	Genome length		Runtime [s]	
	Types	NoTypes	Types	NoTypes
Briefcase	26	32	0.22 ± 0.15	0.97 ± 0.47
Blocksworld	16	108	0.81 ± 0.66	1.91 ± 0.76
FlatTyre	67	405	2.26 ± 0.74	11.47 ± 3.65
Rover	201	2796	4.11 ± 0.33	97.86 ± 7.67
Freecell	291	1481	5.73 ± 1.1	30.7 ± 3.96

Table 4: Performance of LOUGA on models with and without typing.

other two setups. Pure hill climbing performs better on inputs where there are many possible solutions. However if we use complex domains, there is an advantage in incorporating GA, because local search takes a lot of time on big genomes.

Efficiency of using types

We assume that objects (constants) are typed, which reduces the number of candidate predicates for preconditions and effects. LOUGA also works with models without types so our next experiment shows the effect of typing on efficiency.

The results (Table 4) show that LOUGA can handle domain models without types, thought efficiency decreases significantly. The table also shows the increased size of the genome when types are not used. The added genes (predicates) can be split in two groups. The first group consists of genes that use the unary predicates describing types. These genes do not add any difficulty to the problem, because they are not used in any add or delete lists and thus LOUGA immediately finds a trivial solution for those predicates. The second group consists of genes that use the original predicates. These genes do make the problem noticeably harder. In the Rover domain significantly more of these genes were created, because this domain has more operators and predicate types, and therefore more operator's parameter-predicate's parameter pairs were created by removing typing and more genes needed to be added.

Comparison to ARMS

Finally, we compared performance of LOUGA and ARMS (Yang, Wu, and Jiang 2007), which is still the most efficient system for this kind of problem. We used two settings there. First, we used example plans with complete initial states, goal predicates, and a small number of predicates observed in intermediate states (every predicate will have 5% chance to be observed). This is the kind of input ARMS was created for. Second, we used plans with complete initial and ending states but no information about intermediate states, which is input that suits LOUGA well.

ARMS	Add ER	Del ER	Pre ER	Obs. ER	Runtime [s]
Briefcase	0.263	–	0.029	0	6.19
Blocksworld	0.409	0.095	0.039	0.001	28.82
Flat tyre	0.319	0.479	0.342	0.003	504.19
LOUGA	Add ER	Del ER	Pre ER	Obs. ER	Runtime [s]
Briefcase	0	0	0	0	0.29
Blocksworld	0	0	0	0	0.64
Flat tyre	0	0	0	0	1.04

Table 5: Comparison of ARMS and LOUGA systems. Inputs with goal predicates and small number of predicates in intermediate states were used.

ARMS	Add ER	Del ER	Pre ER	Obs. ER	Runtime [s]
Briefcase	0.318	–	0.032	0	6.72
Blocksworld	0.331	0.061	0.036	0.014	29.64
Flat tyre	0.336	0.507	0.311	0.005	548.09
LOUGA	Add ER	Del ER	Pre ER	Obs. ER	Runtime [s]
Briefcase	0	0	0	0	0.22
Blocksworld	0	0	0	0	0.81
Flat tyre	0	0	0	0	2.26

Table 6: Comparison of ARMS and LOUGA systems. Inputs with complete goal states were used.

Tables 5 and 6 clearly indicate that LOUGA outperforms ARMS both in terms of runtime and quality of obtained models. From the data we can also see that ARMS has some problems generating delete lists. In many cases, there were zero predicates in delete lists in total. We assume that it is due to ARMS not having enough information about which predicates need to be deleted. LOUGA has less trouble generating those lists thanks to the assumption that a predicate has to be deleted before it can be added again to the world. But in some cases in the first experiment the learned delete lists were not the same as the delete lists of the original model, because the information about what has to be deleted was not sufficiently present in the plans. In the second experiment LOUGA knew that every predicate in the ending states was observed, so it typically found the original models.

Conclusions

The paper presents a novel approach to learn planning operators from example plans using a genetic algorithm. We presented several techniques to improve performance of the classical genetic algorithm. First, we suggested the preprocessing technique to restrict the set of allowed genomes. Second, we used the genetic algorithm to learn action effects only while the preconditions are learnt separately using an ad-hoc algorithm. Third, we showed that action effects can be learnt predicate by predicate rather than learning the effect completely using a single run of the genetic algorithm.

The presented approach LOUGA achieves much better accuracy and it is faster than the state-of-the-art system ARMS solving the same problem.

Acknowledgments Research is supported by the Czech Science Foundation under the projects P103-15-19877S and P103-18-07252S.

References

Cresswell, S., and Gregory, P. 2011. Generalised domain model acquisition from action traces. In *Proceedings of the 21st International Conference on Automated Planning and Scheduling, ICAPS 2011, Freiburg, Germany June 11-16, 2011*.

Cresswell, S.; McCluskey, T. L.; and West, M. M. 2009. Acquisition of object-centred domain models from planning examples. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling, ICAPS 2009, Thessaloniki, Greece, September 19-23, 2009*.

Gil, Y. 1994. Learning by experimentation: Incremental refinement of incomplete planning domains. In *Proceedings of the Eleventh International Conference on Machine Learning (ICML-94)*, 87–95.

Gregory, P.; Lindsay, A.; and Porteous, J. 2017. Domain model acquisition with missing information and noisy data. In *Proceedings of the Workshop on Knowledge Engineering for Planning and Scheduling, ICAPS 2017*, 69–77.

Jilani, R.; Crampton, A.; Kitchin, D. E.; and Vallati, M. 2015. Ascol: A tool for improving automatic planning domain model acquisition. In *AI*IA 2015, Advances in Artificial Intelligence - XIVth International Conference of the Italian Association for Artificial Intelligence, Ferrara, Italy, September 23-25, 2015, Proceedings*, 438–451.

McCluskey, T. L.; Cresswell, S.; Richardson, N. E.; and West, M. M. 2009. Action knowledge acquisition with opmaker2. In *Agents and Artificial Intelligence - International Conference, ICAART 2009, Porto, Portugal, January 19-21, 2009. Revised Selected Papers*, 137–150.

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. Pddl - the planning domain definition language. Cvc tr-98-003/dcs tr-1165, Yale Center for Computational Vision and Control.

Mitchell, M. 1998. *An Introduction to Genetic Algorithms*. Cambridge, MA, USA: MIT Press.

Shahaf, D.; Chang, A.; and Amir, E. 2006. Learning partially observable action models: Efficient algorithms. In *Proceedings of Twenty-First AAAI Conference on Artificial Intelligence*, 920–926.

Wang, X. 1995. Learning by observation and practice: An incremental approach for planning operator acquisition. In *Proceedings of the Twelfth International Conference on Machine Learning (ICML-95)*, 549–557.

Yang, Q.; Wu, K.; and Jiang, Y. 2007. Learning action models from plan examples using weighted MAX-SAT. *Artif. Intell.* 171(2-3):107–143.

Zhuo, H. H., and Kambhampati, S. 2013. Action-model acquisition from noisy plan traces. In *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, 2444–2450.

Zhuo, H. H.; Yang, Q.; Hu, D. H.; and Li, L. 2010. Learning complex action models with quantifiers and logical implications. *Artif. Intell.* 174(18):1540–1569.

Compiling Away Soft Trajectory Constraints in Planning

Benedict Wright and **Robert Mattmüller** and **Bernhard Nebel**
University of Freiburg
{bwright, mattmuel,nebel}@informatik.uni-freiburg.de

Abstract

Soft goals in planning describe optional goals that should be achieved in the goal state. However, failing to achieve soft goals does not result in the plan becoming invalid. State trajectory constraints describe requirements towards the way the target goal is achieved, thus describing requirements towards the state trajectory of the final plan. Soft trajectory constraints express preferences on how the hard goals are reached, thus stating optional requirements towards the state trajectory of the plan. Such a soft trajectory constraint may require that some fact should be always true, or should be true at some point during the plan. The quality of a plan is then measured by a metric which adds the sum of all action costs and a penalty for each failed soft trajectory constraint. Keyder and Geffner showed that soft goals can be compiled away. We generalize this approach and illustrate a method of compiling soft trajectory constraints into conditional effects and state dependent action costs using LTL_f and Büchi automata. With this we are able to handle such soft trajectory constraints without the need of altering the search algorithm or heuristics, using classical planners.

Introduction

Soft goals in planning are additional requirements towards the resulting plan. These requirements differ from classical (hard) goals in that violating them does not render a plan invalid. PDDL 3.0 (Gerevini and Long 2005) introduced state trajectory constraints, which add constraints towards *how* goals are achieved. These come in two flavors, as hard constraints and as soft constraints. For the rest of the paper, we will refer to optional state trajectory constraints as soft trajectory constraints. We use the term "soft goals" to mean reachability soft goals and soft trajectory constraints alike. This is justified since reachability soft goals φ can be seen as a special case of soft trajectory constraints of the form (at end φ).

For checking satisfaction of reachability soft goals, it is sufficient to test if they hold in the final state. However, for soft trajectory constraints, a more sophisticated method of checking their satisfaction is required. For example, if a soft trajectory constraint requires a fact to be always true, it is not sufficient to check if the fact is true in the final state, but it needs to be tracked to check if the fact holds at any given step of the plan.

The introduction of soft goals changes the overall quality of a plan such that a cheapest plan achieving the hard goals is not necessarily an optimal plan, as it does not take into account the achieving or failing of soft goals. For this, a metric consisting of plan cost and a penalty for violated soft goals is introduced. Thus, an optimal plan would incorporate all soft goals while minimizing the total cost of the plan. This corresponds to a constraint optimization problem, where the constraints are the hard goals and the optimization tries to fulfill the soft goals.

One issue that arises when dealing with soft goals is the trade-off between minimizing cumulative action costs along the way to a state satisfying the hard goals, and maximizing rewards for achieved soft goals. An additional challenge is how to inform the search about which paths appear promising towards optimizing this trade-off. In this paper, we show how soft trajectory constraints can be compiled away using LTL_f, Büchi automata, conditional effects, and state dependent action costs, generalizing the soft goal compilation introduced by Keyder and Geffner (2009). This allows us to use off-the-shelf classical planning heuristics to provide the required guidance.

Related work

Baier and McIlraith (2008) give an overview over planning with preferences, where they use the term preference to state a preference of one plan over another, introducing different preference formalisms based on quantitative, and qualitative languages. Using quantitative languages, the quality of a given plan can be determined by a numeric value, such as the overall reward in Markov Decision Processes (MDP). In these MDPs the reward of an action can be used to specify preferences over actions. Alternatively, the quality of a plan can be determined over a set of properties, such as satisfied preferences. Such a system was implemented in PDDL3 (Gerevini and Long 2005) where preferences can be specified as temporal, or temporally extended predicates, using a subset of LTL.

Baier *et al.* (2009) describe a method of compiling problems with temporally extended preferences into simpler versions consisting only of preferences that hold in the final state, and can be evaluated using an objective function. The authors achieve this by translating the LTL expressed preferences into parameterized non-deterministic finite state au-

tomata (PNFA). They then track the state of each object within the automaton using a predicate for each automaton, which tracks the state of each object within the automaton. Here objects can reside in more than one state of the automaton at each time step. Additionally they introduce a predicate that holds if the automaton is in an accepting state for any given object. Instead of tracking the state of the automaton by extending the existing operators, they modify their search algorithm to automatically apply the automata's state transitions for each object. The quality of their approach can then be measured using an updated objective function.

Keyder and Geffner (2009) show that soft goals can be compiled away by introducing a new hard goal p, which can be achieved in two ways: A $collect(p)$ action which has zero cost but requires the soft goal to be achieved, and a $forgo(p)$ action that has costs equal to the utility of p but can be executed when the soft goal was not achieved. These actions can only be executed after the original plan goal was reached. However their approach does not take trajectory constraints into account, focusing on reachability soft goals only. We build upon this work to generalize their approach towards soft trajectory constraints.

Preliminaries

Linear-time temporal logic on finite traces

Linear-Time Temporal Logic (LTL) is a modal logic capable of expressing logic expressions referring to time. As we will see later in this section, LTL can be used to express trajectory constraints. Let $\mathcal{V}$ be a set of finite-domain state variables with associated finite domains $\mathcal{D}_v$. We call pairs (v, d) with $v \in \mathcal{V}$ and $d \in \mathcal{D}_v$ *facts*, and we denote the set of all facts by F. Then an LTL formula φ over $\mathcal{V}$ is either an atomic fact (v, d) over $\mathcal{V}$, or of the form $\neg\varphi$, $\varphi \vee \psi$, $\bigcirc\varphi$ ("next φ"), or $\varphi\mathcal{U}\psi$ ("φ until ψ"), where φ, ψ are LTL formulas. Other propositional connectives can be defined as abbreviations in the usual way, such as conjunction ($\wedge$), implication ($\rightarrow$), bi-implication ($\leftrightarrow$), truth ($\top$), and falsity ($\bot$). Similarly, $\Diamond\varphi$ ("finally φ") can be defined as an abbreviation for $\top\mathcal{U}\varphi$, and $\Box\varphi$ ("globally φ") as an abbreviation for $\neg\Diamond\neg\varphi$. We also introduce weak until $\varphi\mathcal{W}\psi$ as an abbreviation for $\varphi\mathcal{U}\psi \vee \Box\varphi$. Then the semantics of $\mathsf{LTL_f}$ (LTL on finite traces) is defined as the interpretation over finite traces denoting a sequence of instants of time. Let $\mu = (\mu(0), \mu(1), \ldots, \mu(n))$ be such a trace with $\mu(i) \subseteq F$ for all $i = 0, \ldots, n$. Then the truth of a formula φ along trace μ is defined as follows (De Giacomo and Vardi 2013):

$$
\begin{array}{lll}
\mu, i \models a & \textit{iff} & a \in \mu(i) \text{ for } a \in \mathcal{V} \\
\mu, i \models \neg\varphi & \textit{iff} & \mu, i \not\models \varphi \\
\mu, i \models \varphi_1 \wedge \varphi_2 & \textit{iff} & \mu, i \models \varphi_1 \textit{ and } \mu, i \models \varphi_2 \\
\mu, i \models \bigcirc\varphi & \textit{iff} & i < n \textit{ and } \mu, i+1 \models \varphi \\
\mu, i \models \varphi_1 \mathcal{U} \varphi_2 & \textit{iff} & \exists j, i \leq j \leq n : \mu, j \models \varphi_2 \textit{ and} \\
 & & \quad \forall k, i \leq k \leq j : \ \mu, k \models \varphi_1 \\
\mu \models \varphi & \textit{iff} & \mu, 0 \models \varphi
\end{array}
$$

Trajectory constraints as LTL

PDDL 3.0 (Gerevini and Long 2005) introduced state-trajectory constraints, which are modal logic expressions that ought to be true for the state trajectory produced during the execution of the plan. As shown by De Giacomo *et al.* (2014), these can be expressed using LTL:

$$
\begin{aligned}
\text{(at end } \varphi) &:= \Diamond(\mathit{last} \wedge \varphi) \\
\text{(always } \varphi) &:= \Box\varphi \\
\text{(sometime } \varphi) &:= \Diamond\varphi \\
\text{(within } n\ \varphi) &:= \bigvee_{0 \leq i \leq n} \underbrace{\bigcirc \ldots \bigcirc}_{i} \varphi \\
\text{(hold-after } n\ \varphi) &:= \underbrace{\bigcirc \ldots \bigcirc}_{n} \Diamond\varphi \\
\text{(hold-during } n_1\ n_2\ \varphi) &:= \underbrace{\bigcirc \ldots \bigcirc}_{n_1} \Big(\bigwedge_{0 \leq i \leq n_2} \underbrace{\bigcirc \ldots \bigcirc}_{i} \varphi \Big) \\
\text{(at-most-once } \varphi) &:= \Box(\varphi \rightarrow \varphi\mathcal{W}\neg\varphi) \\
\text{(sometime-after } \varphi\ \psi) &:= \Box(\varphi \rightarrow \Diamond\psi) \\
\text{(sometime-before } \varphi\ \psi) &:= (\neg\varphi \wedge \neg\psi)\mathcal{W}(\neg\varphi \wedge \psi) \\
\text{(sometime-within } n\ \varphi\ \psi) &:= \Box\Big(\varphi \rightarrow \bigvee_{0 \leq i \leq n} \underbrace{\bigcirc \ldots \bigcirc}_{i} \psi\Big)
\end{aligned}
$$

Here, φ and ψ are propositional formulas on fluents, and n, n_1, n_2 natural numbers. The predicate last is introduced during the translation from $\mathsf{LTL_f}$ to LTL, and is true if an only if $\neg \bigcirc \top$ which is the case in the last state of the state trajectory. As the plan resulting from our planning task is always finite, we need this restriction on LTL.

Planning tasks

Since we want to compile away soft trajectory constraints using conditional effects and state-dependent action costs, we base our exposition on a formalization of planning tasks that admits all of those features. This leads us to the following definition:

A *planning task* is a tuple $\Pi = \langle \mathcal{V}, A, s_0, s_\star, \Phi \rangle$ consisting of the following components: $\mathcal{V} = \{v_1, \ldots, v_n\}$ is a finite set of *state variables*, each with an associated finite domain $\mathcal{D}_v$. A *fact* is a pair (v, d), where $v \in \mathcal{V}$ and $d \in \mathcal{D}_v$, and a *partial variable assignment* s over $\mathcal{V}$ is a consistent set of facts. If s assigns a value to each $v \in \mathcal{V}$, s is called a *state*. Let S denote the set of states of Π. A is a set of *actions*, and each action is a pair $a = \langle \mathit{pre}, \mathit{eff} \rangle$, where *pre* is a partial variable assignment called the *precondition*, and where *eff* is an *effect* of the form $\mathit{eff} = \bigwedge_{i=1,\ldots,n} (\mathit{pre}_i \rhd \mathit{eff}_i)$ for some number $n \in \mathbb{N}$ of *conditional effects*, each consisting of an effect condition pre_i, again a partial variable assignment, and an effect eff_i, also a partial variable assignment. The state $s_0 \in S$ is called the *initial state*, and the partial state $s_\star$ specifies the *goal* condition. Each action $a \in A$ has an associated cost function $c_a : S \rightarrow \mathbb{N}$ that assigns the cost of a to each state where a is applicable. Finally, Φ is a finite set of $\mathsf{LTL_f}$ formulas over $\mathcal{V}$, the *soft trajectory constraints*. Each soft trajectory constraint $\varphi \in \Phi$ has an associated *weight* $w_\varphi \in \mathbb{N}$ specifying which importance we

assign to satisfying φ. For states s, we use function notation $s(v) = d$ and set notation $(v, d) \in s$ interchangeably. For facts we also use sets and conjunctive logical expressions interchangeably, where a set of facts is treated equivalently to a conjunction of these facts.

The change set $[\mathit{eff}]_s$ of effect $\mathit{eff} = \bigwedge_{i=1,\ldots,n}(\mathit{pre}_i \rhd \mathit{eff}_i)$ in state s is the set of facts that *eff* makes true if applied in s, i. e., the set $\bigcup_{i=1,\ldots,n}[\mathit{pre}_i \rhd \mathit{eff}_i]_s$, where $[\mathit{pre}_i \rhd \mathit{eff}_i]_s$ is either $\emptyset$, if $s \not\models \mathit{pre}_i$, or eff_i, if $s \models \mathit{pre}_i$. Then an action $a = \langle \mathit{pre}, \mathit{eff} \rangle$ is applicable in state s iff $\mathit{pre} \subseteq s$ and the change set $[\mathit{eff}]_s$ is consistent. Applying action a to s yields the state s' with $s'(v) = [\mathit{eff}]_s(v)$ where $[\mathit{eff}]_s(v)$ is defined, and $s'(v) = s(v)$ otherwise. We write $s[a]$ for s'. A state s is a goal state iff $s_\star \subseteq s$. We denote the set of goal states by $S_\star$. Let $\pi = (a_0, \ldots, a_{n-1})$ be a sequence of actions from A. We call π *applicable* in s_0 if there exist states $s_1, \ldots, s_n$ such that a_i is applicable in s_i and $s_{i+1} = s_i[a_i]$ for all $i = 0, \ldots, n-1$. In that case, we call $\mu_\pi = (s_0, s_1, \ldots, s_n)$ the *state trajectory induced by* π *in* s_0. We call π a *plan* for Π if it is applicable in s_0 and if $s_n \in S_\star$. The *action cost* of plan π is the sum of action costs along the induced state sequence, i.e., $\mathit{cost}(\pi) = \sum_{i=0}^{n-1} c_{a_i}(s_i)$. A plan π is penalized with penalty w_φ for each soft trajectory constraint $\varphi \in \Phi$ that is violated on its induced trajectory. Formally, the value $\mathit{penalty}(\pi, \varphi)$ for π with respect to φ is 0, if $\mu_\pi \models \varphi$, and w_φ, if $\mu_\pi \not\models \varphi$. The *overall penalty* for π is $\mathit{penalty}(\pi) = \sum_{\varphi \in \Phi} \mathit{penalty}(\pi, \varphi)$.

The *total cost* of plan π is its action costs plus its overall penalty, i. e., $\mathit{totalcost}(\pi) = \mathit{cost}(\pi) + \mathit{penalty}(\pi)$. A plan is *optimal* for Π if it minimizes $\mathit{totalcost}$ among all plans for Π.

Automata semantics of planning tasks

A deterministic finite automaton (DFA) is a tuple $\mathcal{A} = \langle \Sigma, Q, \Delta, q_0, Q_a \rangle$ consisting of an alphabet Σ, a set of states Q, a transition function $\Delta : Q \times \Sigma \to Q$, an initial state $q_0 \in Q$, and a set of accepting states $Q_a \subseteq Q$. The transition system of any planning task $\Pi = \langle \mathcal{V}, A, s_0, s_\star, \Phi \rangle$ can be understood as a DFA $\mathcal{A}(\Pi)$ as follows: the input alphabet is $\Sigma = A \times 2^F$. The set of states, the initial state, and the set of accepting/goal states of $\mathcal{A}(\Pi)$ are those of Π, i. e., $Q = S$, $q_0 = s_0$, and $Q_a = S_\star$. Finally, Δ consists of all transitions of the form $\langle s, (a, t), t \rangle$ where $a \in A$ is applicable in s and $s[a] = t$. What was lost in the translation from Π to $\mathcal{A}(\Pi)$ are the action costs and the soft trajectory constraints. Costs are trivial to handle by adding weights to the automaton, and we will come back to that later. To give an automata-based semantics to state-trajectory constraints, we need to review the theory of Büchi automata first.

Büchi automata

A deterministic Büchi automaton (Büchi 1962) $\mathcal{B} = (\Sigma, Q, \Delta, q_0, Q_a)$ consists of the same components as a DFA, and differs from a DFA only in the acceptance condition. Whereas a DFA $\mathcal{A}$ accepts a finite input word μ if after reading μ, $\mathcal{A}$ is in an accepting state, a Büchi automaton $\mathcal{B}$ accepts an infinite word μ if, while reading μ, $\mathcal{B}$ visits an accepting state infinitely often. For every LTL formula φ, there is a deterministic Büchi automaton $\mathcal{B}(\varphi)$ that accepts exactly those infinite words μ with $\mu \models \varphi$. In the case of finite traces (finite words) required by $\mathsf{LTL_f}$, the same automaton accepts the word if at the end of the word the automaton is in an accepting state (Giannakopoulou and Havelund 2001). There are multiple algorithms for constructing a Büchi automaton that accepts exactly those words that satisfy a given LTL formula (Gerth *et al.* 1996; Gastin and Oddoux 2001). Constructing an automaton from a given $\mathsf{LTL_f}$ formula φ can be achieved by first translating φ into a LTL formula as described in De Giacomo *et al.* (2014) and then applying a given construction algorithm. Simply put, this translation adds a new predicate last which is only true in the last instance of the interpretation sequence, and therefore ensuring finite traces.

Now, for a planning task Π with a *hard* state-trajectory constraint φ, the standard automaton construction considers the product automaton $\mathcal{C}$ of $\mathcal{A}(\Pi)$ and $\mathcal{B}(\varphi)$. Then, a state trajectory μ is a solution to Π satisfying φ iff μ is accepted by $\mathcal{C}$. For *soft* state-trajectory constraints, we can still perform the same product automaton construction to track which soft constraints are satisfied by a plan. Unlike with hard constraints, however, the product automaton still has to accept trajectories that violate soft constraints, and the violation has to be reflected in the plan costs, rather than in the acceptance condition of the product automaton. The next section describes the product construction, an assignment of action costs that reflects the satisfaction or violation of soft trajectory constraints, and a compact encoding of the product automaton as a new planning task Π'.

Goal action penalty compilation

Let $\Pi = \langle \mathcal{V}, A, s_0, s_\star, \Phi \rangle$ be the original planning task with soft trajectory constraints Φ and with objective function $\mathit{totalcost}$ as defined above. Transition costs aside, the semantics of Π are captured by the product automaton $\mathcal{C} = \mathcal{A}(\Pi) \times \prod_{\varphi \in \Phi} \mathcal{B}(\varphi)$. However, when compiling away soft trajectory constraints, we do not want to generate an *automaton*, but rather another *planning task* Π' such that $\mathcal{A}(\Pi')$ is isomorphic to $\mathcal{C}$. We now describe this construction. For simplicity of exposition, we assume that Φ consists of a single constraint φ only. Generalization to more than one soft trajectory constraint is straightforward.

The idea behind the construction of Π' is to add a new tracking variable τ_φ to Π that keeps track of the current state of $\mathcal{B}(\varphi)$. The actions in Π' are those from Π, augmented with conditional effects that take care of the correct evolution of the value of τ_φ, thus encoding the soft trajectory constraints into the actions. Action costs stay the same. Finally, a terminal action $\mathit{last\_op}$ is added to Π' that marks termination. Only after termination has been marked, we may start evaluating the penalty term for unsatisfied soft goals.

Formally, let $\mathcal{B}(\varphi) = (Q, \Sigma, \Delta, q_0, Q_a)$ be the deterministic Büchi automaton that accepts φ. Then we create planning task $\Pi' = \langle \mathcal{V}', A', s'_0, s'_\star, \emptyset \rangle$ with $\mathcal{V}' = \mathcal{V} \cup \{\mathit{last}\} \cup \{\tau_\varphi\}$, with a propositional domain for last and domain Q for τ_φ. The initial state s'_0 agrees with s_0 on all variables in $\mathcal{V}$, and additionally, $s'_0(\mathit{last}) = \bot$ and $s'_0(\tau_\varphi) = q_0$.

The actions are $A' = \{o' \mid o \in A\} \cup \{last\_op\}$ where $o' = \langle pre', eff' \rangle$ is constructed from $o = \langle pre, eff \rangle$ as follows: $pre' = pre$ and

$$eff' = eff \wedge \bigwedge_{\substack{\langle q,s,q' \rangle \in \Delta \text{ with} \\ last \notin s}} ((\tau_\varphi = q \wedge P) \rhd \tau_\varphi := q'),$$

where $P = q' \setminus eff$. In words, we add conditional effects to track the value of τ_φ for each transition in $\mathcal{B}(\varphi)$. The facts in s are either already true in q as ensured by the effect conditions P or are set to true by the original actions effect *eff*. As an exception, if s contains the keyword $last$, which can only be true in the last step of the plan, we add the new action $last\_op = \langle s_\star, last := \top \rangle$ instead. To ensure that this action has to be executed as the last step of any plan for Π', we replace the original goal condition $s_\star$ by the new goal condition $s'_\star = last$. As action costs, we have $c_{o'} = c_o$ for all $o \in A$, and $c_{last\_op} = 0$. Additional formal machinery needed for the evaluation of the penalty term is deferred until after the following proposition.

Proposition 1. *Up to transitions with action last_op, the transition system $\mathcal{A}(\Pi')$ is isomorphic to the product of the transition system $\mathcal{A}(\Pi)$ and trajectory constraints LTL_f automaton $\mathcal{B}(\varphi)$.*

Proof. For this proof we slightly alter $\mathcal{B}(\varphi)$ such that each transition not only consists of a partial variable assignment, but a tuple of a state and an action. We replace each transition $\langle q, s, q' \rangle$ in $\mathcal{B}(\varphi)$ by a set of new transitions $\langle q, (o, t), q' \rangle$ for all $(t, o) \in S \times A$ such that $s \subseteq t$, where o is an action that after applying to q results in q'. Doing this creates an automaton with the same signature as $\mathcal{A}$. From this altered Büchi automaton we can now easily construct $\mathcal{A}(\Pi) \times \mathcal{B}(\varphi)$, by simply creating the Cartesian product of the two automata (Baier and Katoen 2008). A transition $\langle s'^q, (o', t'^{q'}), t'^{q'} \rangle$ is contained in $\mathcal{A}(\Pi')$ if and only if o is applicable in $s \in \Pi$ and $t = apply(o, s)$ and $(\tau_\varphi, q) \in s'^q$ and $(\tau_\varphi, q') \in t'^{q'}$. Then $\langle s, (o, t), t \rangle \in \mathcal{A}(\Pi)$ and $\langle q, (o, t), q' \rangle \in \mathcal{B}(\varphi)$ if and only if $\langle (s, q), (o, t), (t, q') \rangle \in \mathcal{A}(\Pi) \times \mathcal{B}(\varphi)$, thus $\mathcal{A}(\Pi')$ is isomorphic to $\mathcal{A}(\Pi) \times \mathcal{B}(\varphi)$. □

Now that we can track the state of each soft trajectory constraint within the planning task Π', we need to add penalties for all constraints not achieved in the reached terminal state. For this we add another propositional variable in_goal to Π' that is initially false, and change the goal $s'_\star$ from $last$ to in_goal. This means that every plan for Π' has to include an occurrence of the new action $penalize = \langle last \wedge \neg in\_goal, in\_goal \rangle$ as its last step. The cost function of the action $penalize$ now simply determines the penalty value $penalty(\pi)$ based on which soft trajectory constraints $\varphi \in \Phi$ are violated by testing whether the corresponding tracking variables τ_φ encode accepting or non-accepting Büchi automata states in the current planning state. More formally, $c_{penalize} = \sum_{\varphi \in \Phi} [\tau_\varphi \notin Q_a^\varphi] w_\varphi$ where $[\tau_\varphi \notin Q_a^\varphi] = 1$ if $\tau_\varphi = q$ and $q \notin Q_a^\varphi$ for some $q \in Q^\varphi$, and 0 otherwise.

Notice that the action $penalize$ has *state-dependent costs* that are not universally supported by planning systems. However, those can be compiled away to state-independent costs, if this is desired (Geißer *et al.* 2015). Notice further that determining the value $[\tau_\varphi \notin Q_a^\varphi]$ is also simple. It can either be rewritten as $\sum_{q \in Q^\varphi \setminus Q_a^\varphi} [\tau_\varphi = q]$, where $[\tau_\varphi = q]$ is 1 if $s(\tau_\varphi) = q$, and 0 otherwise; alternatively, another new propositional variable $is\_violated_\varphi$ can be added to the planning task that is true iff the value of τ_φ represents a non-accepting state. Then $c_{penalize} = \sum_{\varphi \in \Phi} [is\_violated_\varphi] w_\varphi$. A natural modeling would treat $is\_violated_\varphi$ as a derived variable, and would have axioms that express $is\_violated_\varphi$ in terms of τ_φ. We mention this latter possibility since it makes the relation between our proposed compilation and that of Keyder and Geffner (2009) obvious (cf. Remark 1 below).

In any case, it is clear that adding this action preserves the original objective function.

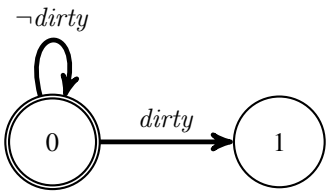


Figure 1: Büchi automaton for $\Box\neg dirty$

Proposition 2. *Let Π' be the compiled task from Π. Then an optimal plan for Π' is also an optimal plan for Π (without the penalize action).*

Proof. From Proposition 1 we get that the compilation is sound and complete. The objective function of the original task is $penalty(\pi) + cost(\pi)$. Up until the $penalize$ action, the objective function sums up all action costs, as the cost functions for each action are not altered by the compilation. The $penalize$ action then adds a penalty for each soft trajectory constraint that is not satisfied, resulting in an objective function identical to the original objective function. □

Example 1. *Let $a_1 = \langle \top, dirty \rangle$ be an action and φ the preference $\Box\neg dirty$. We can then track the state in the automaton in Figure 1 by adding the conditional effect $(\tau_\varphi = 0 \rhd \tau_\varphi := 1)$ to action a_1, as can be derived from Figure 2. Let a_2 be another action that does not have $dirty$ among its effects. Then we need to add the conditional effect $(\tau_\varphi = 0 \wedge dirty \rhd \tau_\varphi := 1)$ to a_2 that transitions from state 0 to state 1 if $dirty$ is true regardless of the effect of a_2.[1] The partial cost function c for this preference is $c = [\tau_\varphi = 1] w_\varphi$ and is added to the cost of the $penalize$ action. This adds w_φ to the total plan cost if $\mathcal{B}(\varphi)$ is in the non-accepting state 1.*

An analysis of the $penalize$ action shows that after applying the EVMDD compilation (Geißer *et al.* 2015), the resulting operations correspond to the operations $collect$, $forgo$,

[1] It is an invariant of this planning task that, whenever $dirty$ is true, τ_φ is 1. Therefore, the effect condition $\tau_\varphi = 0 \wedge dirty$ can never be satisfied. However, detecting this, and then removing conditional effects whose condition is inconsistent with an invariant, and thus simplifying the constructed conditional effects, is beyond the scope of this work.

$$\tau_\varphi := \begin{cases} 1: & \tau_\varphi = 0 \wedge (\mathit{dirty} \in [\mathit{eff}]_s \vee \\ & \qquad (s \models \mathit{dirty} \wedge \neg \mathit{dirty} \notin [\mathit{eff}]_s)) \\ 1: & \tau_\varphi = 1 \\ 0: & \tau_\varphi = 0 \wedge (\neg \mathit{dirty} \in [\mathit{eff}]_s \vee \\ & \qquad (s \models \neg \mathit{dirty} \wedge \mathit{dirty} \notin [\mathit{eff}]_s)) \end{cases}$$

Figure 2: Derivation of conditional effects for τ_φ from Figure 1, where s is the current state of the search

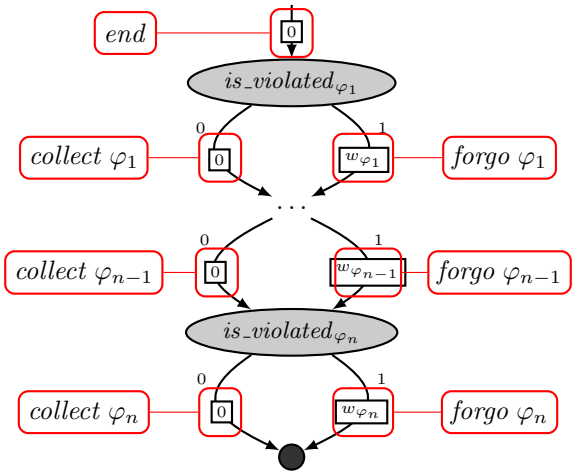


Figure 3: EVMDD compilation of *penalize* action with derived variables $\mathit{is\_violated}_{\varphi_i}$, which are true if τ_{φ_i} is in a non-accepting state. Numbers on edges are partial costs (= costs of compiled actions).

and *end* described in the compilation by Keyder and Geffner (2009). This immediately implies that our approach generalizes the soft trajectory constraint compilation by Keyder and Geffner (2009) to support trajectory constraints.

Remark 1. *We've seen that we can express the cost of the penalize action as* $c_{penalize} = \sum_{\varphi \in \Phi} [\mathit{is\_violated}_\varphi] w_\varphi$. *Expressed as an edge-valued multi-valued decision diagram (EVMDD) (Geißer* et al. *2015),* $c_{penalize}$ *looks as depicted in Figure 3 (without the red annotations). The EVMDD-based action compilation of Geißer* et al. *(2015) now turns each edge of the EVMDD into a new auxiliary action. These new actions are exactly the end, collect, and forgo actions from Keyder and Geffner (2009) (indicated as the red annotations).*

One limitation of this approach is that the achievement of any soft trajectory constraint is only represented by the h-value (up until the *penalize* action). A more desirable compilation would provide the search with a more accurate g-value, thus informing the search when a soft trajectory constraint is achieved. In the following section we will demonstrate a possible solution to this problem.

General action penalty compilation

In this section we will show how the above approach can be extended to provide the search with a more accurate g-value. The main reason for the uninformedness in relation to the g-value is the fact that any penalty is only applied in the very last step of the search in the *penalize* action. However, while tracking the soft trajectory constraint's automaton $\mathcal{B}(\varphi)$, we already have information about the current acceptance status of each soft trajectory constraint. We will now show how this information can be used to add penalties and rewards to the individual actions changing the state of $\mathcal{B}(\varphi)$.

Whenever an action a changes the value of τ_φ, thus transitioning from one state q to another state q' in $\mathcal{B}(\varphi)$, we add a penalty or a reward depending on the type of transition. When q is an accepting state and q' a non-accepting state in $\mathcal{B}(\varphi)$, we add a penalty to the action cost. If, on the other hand, q' is an accepting state and q is a non-accepting state, we can add a reward. The partial cost function for transitions in $\mathcal{B}(\varphi)$ then takes the form $\sum_{q \in Q^\varphi} [\tau_\varphi = q] \omega_\varphi(q, q')$, where $\omega_\varphi(q, q')$ is a penalty or reward term and is added to the cost function c of a. For transitions from accepting to non-accepting states, we set $\omega_\varphi(q, q')$ to a positive penalty term and for transitions from non-accepting to accepting states, we set $\omega_\varphi(q, q')$ to a reward in the form of a negative value. Similarly, the partial cost functions for each type of transition can be formulated, setting $\omega_\varphi(q, q')$ accordingly. For the actual value of $\omega_\varphi(q, q')$, we use the value from the original soft trajectory's weight w_φ. The total cost function of each action is then the sum of the partial cost functions plus the original action cost.

This way, we penalize actions resulting in a transition from accepting to non-accepting states by giving them higher costs, and reward actions that result in an accepting state of $\mathcal{B}(\varphi)$ by applying negative costs. Note, that $\omega_\varphi(q, q')$ only accounts once in the total cost, as we can never add $\omega_\varphi(q, q')$ without subtracting it beforehand.

By construction, minimizing *totalcost* in the compiled task Π' amounts to the same as minimizing the *totalcost* of the original task Π. One minor detail to take in to account is if the initial state of $\mathcal{B}(\varphi)$ is in a non-accepting state, we need to add a penalty to account for this. We do this by adding an additional penalty to the *penalize* action.

The problem now is that we have introduced negative action costs. As we can ensure that we do not have any negative cycles in our search, resulting in a total plan cost ≥ 0, we can use planners that support negative action costs. Note that having such negative-cost cycles would result in arbitrarily low *totalcost*, and the non-termination of the search, as each node in the cycle can be reached by a yet cheaper path. Currently, Fast Downward (Helmert 2006) with blind heuristic supports negative action costs. However, for more sophisticated heuristics, or planners not supporting negative action costs, negative action costs need to be removed.

To remove negative action costs, we introduce a state transition cost (Table 1), where we specify the penalty/reward for each possible transition type. This transition cost table gives us greater control over the implications a state transition in $\mathcal{B}(\varphi)$ has towards fulfilling the soft trajectory con-

Table 1: State Transition Costs

(a) Metric Preserving Costs

From \ To	Accepting	¬ Accepting
Accepting	0	w_φ
¬ Accepting	$-w_\varphi$	0

(b) Positively Shifted Costs

From \ To	Accepting	¬ Accepting
Accepting	w_φ	$2w_\varphi$
¬ Accepting	0	w_φ

(c) Adapted Positively Shifted Costs

From \ To	Accepting	¬ Accepting
Accepting	0	$2w_\varphi$
¬ Accepting	0	w_φ

straint. For instance, by setting the penalty/reward $\omega_\varphi(q, q')$ of a transition from an accepting state to another (or the same) accepting state to $\omega_\varphi(q, q') = 0$ and all other transitions to $\omega_\varphi(q, q') > 0$, we can model the preference of staying in an accepting state over all other possibilities. Additionally, we can set the cost for leaving an accepting and entering a non-accepting state higher as to penalize these actions.

The transition cost table (Table 1a) corresponds to the cost function described above. Table 1b shows the cost function where the costs have been shifted by w_φ to remove negative costs. As one can see this has the negative effect of penalizing state transitions from accepting to accepting states. Therefore, we introduce transition Table 1c, where transitions from accepting to accepting sates are also not penalized. Transitions leaving an accepting state, however, are highly penalized, whereas remaining in a non-accepting state is only penalized by a lower cost.

This cost function is informative regarding h and g values, regardless of the actually used cost table, however the total cost of the compiled task is greater than the original plans total cost $\mathit{totalcost}(\pi') \geq \mathit{totalcost}(\pi)$, where π, π' are plans from Π and Π' respectively. This is due to the fact that penalties from staying in a non-accepting state are added multiple times.

Experiments

We implemented our compilation into a recent version of the Fast Downward planning system supporting state dependent action costs. The evaluation was executed on a a subset of the fifth International Planning Competition (IPC-5) plus the Rovers domain from the IPC-3. We will now first discuss the results for the goal action penalty compilation, followed by the general action penalty compilation, finalizing with a discussion and comparison of the two approaches. In the domain names, SP and QP stand for Simple Preferences and Qualitative Preferences, respectively. The difference in these being that simple preferences use goal state preferences of the form (at end φ) only, and qualitative preferences use more complex state trajectory constraints. As the competition was for satisficing planning only, and many instances were too hard for optimal planning, which we are interested in, we generated additional simpler instances by randomly sampling subsets of the soft trajectory constraints. From each instance, we generated six new instances with 1%, 5%, 10%, 20%, 40%, and 100% of the soft trajectory constraints. We did not alter the hard goals of the original instances, which led to the exclusion of the openstacks domain, as finding optimal solutions for more than the very simple instances proved to be too hard.

Goal action penalty compilation results

For the goal action penalty compilation, we used h^{blind}, $h^{\max}$, and $h^{\text{M\&S}}$ for the optimal track. For the satisficing benchmark, we used h^{add} and h^{FF} with iterative eager greedy search with three iterations. No significant differences where found between the two heuristics in the satisficing benchmark, with a slightly better performance by h^{FF}. In the remaining evaluation, we therefore only consider h^{FF}.

As can be seen in Table 2, the performances varied over the domains. This is a consequence of finding an optimal solution to the hard goals even without considering the soft trajectory constraints. The trucks domain did not execute on the merge and shrink heuristic, as this heuristic does not support axioms, which are introduced by the translate step in the Fast Downward planner.

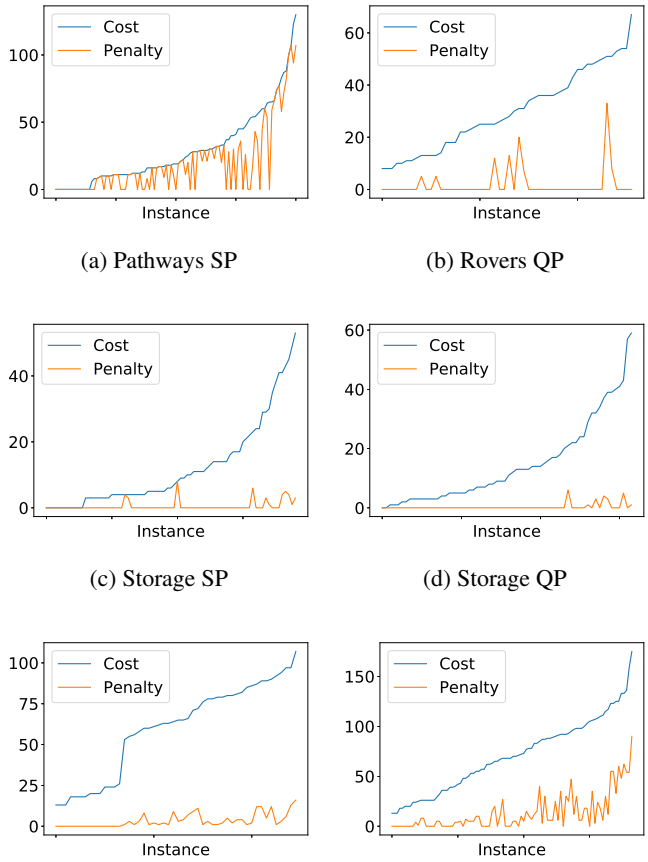
As can be seen in Figure 4, the satisficing benchmark performed rather well on the Rovers, Storage, and Trucks SP domain, as their penalty is always close to zero. The quality of the Trucks QP domain is slightly worse as fulfilling all soft trajectory constraints becomes more difficult, the more complex the instance is. For the pathways domain, we increased the penalty for not achieving soft goals by a factor of 10, as otherwise the optimal plan would be to ignore the soft trajectory constraints. As this domain has no hard goals, but soft trajectory constraints only, this would have resulted in an empty plan. As can be seen in some cases this was not sufficient and the resulting penalty is equal to the total cost, indicating that no soft trajectory constraints where satisfied. The storage domain also has no hard goals, but the penalties where already high in comparison to the action costs, requiring no alteration of the penalties.

General action penalty compilation results

Here we compare the results using the different configurations from Table 1. The experimental setup is identical to the above with the slight exception to configuration from Table 1a where only h^{blind} was used, as it requires negative action costs. As can be seen in Table 3a, the increased informedness of the general action compilation together with the metric preserving cost function did not significantly increase the amount of optimally solved instances. This is a result of the relative uninformedness of the blind heuristic, and the fact that the cost function needs to be evaluated for

Domain	h^{blind}	$h^{\max}$	$h^{\text{M\&S}}$
pathways SP	12.22%	18.33%	12.22%
rovers QP	18.33%	14.17%	16.67%
storage SP	33.33%	39.17%	32.50%
storage QP	25.49%	32.35%	24.51%
trucks SP	23.53%	15.29%	na
trucks QP	19.83%	14.66%	na

Table 2: Coverage of goal action penalty compilation of the IPC-5 benchmark set with additional instances with randomly sampled soft trajectory constraints, A* search for optimal solution.



(a) Pathways SP (b) Rovers QP

(c) Storage SP (d) Storage QP

(e) Trucks SP (f) Trucks QP

Figure 4: Plan quality of the satisficing benchmarks, ordered by *total cost* using goal action penalty compilation and h^{FF} heuristic

each action. As we currently use a relative unoptimized internal representation of the cost function, this significantly increases the search time, leading to timeouts before a solution could be found.

As can be seen in Tables 3b and 3c, the coverage increased significantly on these two cost compilations. This, however, is an artifact of the introduced error, as all actions become more expensive to execute. This results in the penalty for not achieving the soft trajectory constraints to become relatively low compared to the action costs. Thus, the empty plan becomes the optimal plan where no hard goals are specified, and the shortest plan becomes the optimal plan where hard goals are specified. This could be improved by a scaling function, which increases the penalty for not achieving the soft trajectory constraints and/or decreases the action costs.

Domain	h^{blind}	$h^{\max}$	$h^{\text{M\&S}}$
pathways SP	18.00%	na	na
rovers QP	12.00%	na	na
storage SP	12.00%	na	na
storage QP	8.16%	na	na
trucks SP	16.15%	na	na
trucks QP	26.45%	na	na

(a) Metric Preserving Costs

Domain	h^{blind}	$h^{\max}$	$h^{\text{M\&S}}$
pathways SP	36.67%	77.22%	7.22%
rovers QP	19.17%	11.67%	8.33%
storage SP	78.33%	78.33%	8.33%
storage QP	77.45%	76.47%	4.90%
trucks SP	23.53%	12.97%	na
trucks QP	20.54%	8.93%	na

(b) Positively Shifted Costs

Domain	h^{blind}	$h^{\max}$	$h^{\text{M\&S}}$
pathways SP	36.67%	77.22%	7.78%
rovers QP	16.67%	15.00%	10.00%
storage SP	78.33%	78.33%	8.33%
storage QP	77.45%	77.45%	4.90%
trucks SP	23.53%	12.94%	na
trucks QP	20.54%	12.50%	na

(c) Adapted Positively Shifted Costs

Table 3: Coverage of general action penalty compilation with the configurations from Table 1

Comparison to zero penalty compilation

Finally, we executed the same test set without a penalty action cost on goal action penalty compilation with blind heuristics for optimal solutions, and compared it to the above results regarding the average fulfilled soft trajectory con-

Domain	penalty	no penalty
pathways SP	97.19%	46.10%
rovers QP	47.05%	20.20%
storage SP	99.50%	54.20%
storage QP	99.90%	48.40%
trucks SP	98.10%	75.20%
trucks QP	100.00%	100.00%

Table 4: Comparison of average fulfilled soft trajectory constraints with and without penalty cost, only regarding instances for which a solution was found

straints, as shown in Table 4. Here, no penalty corresponds to the accidental fulfillment of the soft trajectory constraint, as the search is not guided towards them. As can be seen, the percentage of fulfilled soft trajectory constraints is significantly higher with cost guidance. The trucks domain does not show significant difference. This is a result of the overall hardness of finding an optimal solution as can be seen in Figure 2, as instances for which a solution was found were also easy to optimize towards their soft goals, whereas harder instances where not solved at all. Harder instances where not solved and thus not accounted for in Table 4.

Conclusion

In this paper, we introduced a method of compiling soft trajectory constraints into actions with conditional effects and state dependent action costs. For this, we created Büchi automata for each grounded soft trajectory constraint and modified the original planning task to track the state of each automaton during the state trajectory of the current partial plan. We then used state-dependent action costs to inform the heuristic guiding the search towards an optimal solution considering the soft trajectory constraints. We then conducted experiments using the IPC-5 benchmark set with additional generated instances. We showed that this approach enables classical planners to search for optimal solutions, taking soft trajectory constraints into account, without altering the search algorithm or implementing special heuristics.

Future work

One issue we found was that some soft trajectory constraints are simply not reachable or contradict hard goals. Therefore, these soft trajectory constraints can be removed from the search completely, and the penalty can be added directly in the $penalize$ action. We expect this to improve the overall performance of our approach, as the effort needed to track the states and calculate the costs is reduced.

Additionally, the cost function and automata tracking can be simplified by applying optimizations on the generation of the Büchi automata.

In the action penalty compilation, we introduced negative action costs. In our setting, using the Fast Downward planner (Helmert 2006), we were only able to use the blind heuristic, as it does not fail on negative action costs. An analysis of alternative heuristics concerning negative action costs could significantly improve the performance of our approach.

Going beyond what is already supported by PDDL 3, conditional preference networks (CP-nets) can express relations between preferences (Baier and McIlraith 2008). We would like to extend this notion to express relations between soft goals in planning such that we can state things like *if* A *then* B, where A is a fact that can become true and B is a soft goal. For example, we could express the soft goal *if in Paris, visit the Eiffel Tower*, where being in Paris may be a hard or a soft goal, or even just a intermediate location, and visiting the Eiffel Tower is not a hard goal but a soft goal. This increases a planner's capability of creating more user centric plans, by incorporating these preferences into a planning instance.

Acknowledgments. This work was partly supported by BrainLinks-BrainTools, Cluster of Excellence funded by the German Research Foundation (DFG, grant number EXC 1086). We thank the the anonymous reviewers for their insightful comments, helping in improving the overall quality of the paper.

References

Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008.

Jorge A. Baier and Sheila A. McIlraith. Planning with preferences. *AI Magazine*, 29(4):25–36, 2008.

Jorge A. Baier, Fahiem Bacchus, and Sheila A. McIlraith. A heuristic search approach to planning with temporally extended preferences. *Artificial Intelligence Journal (AIJ)*, 173(5–6):593–618, 2009.

J. Richard Büchi. On a decision method in restricted second order arithmetic. In *Proceedings of the International Congress on Logic, Methodology, and Philosophy of Science*, pages 1–11, 1962.

Giuseppe De Giacomo and Moshe Y. Vardi. Linear temporal logic and linear dynamic logic on finite traces. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI 2013)*, pages 854–860, 2013.

Giuseppe De Giacomo, Riccardo De Masellis, and Marco Montali. Reasoning on LTL on finite traces: Insensitivity to infiniteness. In *Proceedings of the 28th AAAI Conference on Artificial Intelligence (AAAI 2014)*, pages 1027–1033, 2014.

Paul Gastin and Denis Oddoux. Fast LTL to Büchi automata translation. In *Proceedings of the 13th International Conference on Computer Aided Verification (CAV 2001)*, pages 53–65, 2001.

Florian Geißer, Thomas Keller, and Robert Mattmüller. Delete relaxations for planning with state-dependent action costs. In *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI 2015)*, 2015.

Alfonso Gerevini and Derek Long. Plan constraints and preferences in PDDL3: The language of the 5th international planning competition. Technical report, Department of Electronics for Automation, University of Brescia, Italy, 2005.

Rob Gerth, Doron Peled, Moshe Y. Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proceedings of the 15th International Symposium on Protocol Specification, Testing and Verification*, pages 3–18, 1996.

Dimitra Giannakopoulou and Klaus Havelund. Automata-based verification of temporal properties on running programs. In *Proceedings of the 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, pages 412–416, 2001.

Malte Helmert. The Fast Downward planning system. *Journal of Artificial Intelligence Research (JAIR)*, 26:191–246, 2006.

Emil Keyder and Hector Geffner. Softgoals can be compiled away. *Journal of Artificial Intelligence Research (JAIR)*, pages 547–556, 2009.

Learning Numerical Action Models from Noisy and Partially Observable States by means of Inductive Rule Learning Techniques

José Á. Segura-Muros, Raúl Pérez, Juan Fernández-Olivares,
University of Granada, Granada, Spain
{josesegmur,fgr,faro}@decsai.ugr.es

Abstract

Designing a planning domain is a cumbersome task that requires time and expert knowledge. In order to overcome this problem planning domain learning techniques are used to create planning domains from existing real-world processes. But like any data obtained from a real-world application, data may be incomplete or noisy. This paper presents PlanMiner-O2, an algorithm that uses a classification algorithm, based on inductive rule learning techniques, to learn action models with discrete numerical values (represented as action costs) from incomplete and noisy data. Starting from plan traces with intermediate partially observable states affected by noise, PlanMiner-O2 generates valid PDDL planning domains that can be used correctly to solve planning problems. It was tested using benchmark domains obtained from the International Planning Competition and the results show that is able to learn even with high levels of incompleteness and noise, being competitive in error rate and time efficiency with respect other state-of-art solutions.

1 Introduction

The task of defining an AI automated planning (AP) domain is a lengthy process that requires a lot of time and extensive knowledge of the problems that want to be solved. This issue worsens depending on the complexity of the world in which the planning domain should be developed: in order to make the planning domain able to work in a different array of situations, the domain's designer must take into account each one of them. This lead to the definition of a planning domain being a hard task. To overcome this issue, one approach that is recently receiving a lot of attention is the use of planning domain learning techniques that extract a planning domain from previously executed plan traces.

In this paper, we present PlanMiner-O2: a new planning domain learner that relies on the use of a classification algorithm based on inductive rule learning. PlanMiner-O2 is able to learn action models from incomplete and noisy information. The basis of our approach is the use of a rule learning algorithm to learn the fluents of the previous (pre-state) or subsequent (post-state) states associated to every action of the planning domain and use them to create a planning domain. The main reasons to address this problem by inductive rule learning is that it improves the interpretability of the models used to create the action models as well as allows for the learning of logical and numerical information. Another advantage of inductive rule learning is that the models they produce show explicitly the relationships between the variables involved in the problem. This extra information can be used to guide the learning process.

Domain learning is a very broad subfield of Automated Planning with several solutions proposed (Jiménez *et al.* 2012). In the literature, we can find solutions as different as OBSERVER (Wang 1995) that monitors executions of expert agents, TRAIL (Benson 1996) that relies on an expert human teacher to guide the learning process or EXPO (Gil 1994) that starts from an initial incomplete domain and uses plans executions to complete it. New approaches, like LOCM (Cresswell *et al.* 2013), LC_M, NLOCM (Gregory and Lindsay 2016) or NLOCM$_{BF}$ (Hayton *et al.* 2016), rely on strategies such as using context-free models and fit a series of constraints to create valid models. As examples of domain learners that can deal with incomplete information, we can find ARMS (Yang *et al.* 2007) and LAMP (Zhuo *et al.* 2010). Both generate sets of logic formulas to model domain's actions and select the best ones using a MAX-SAT (ARMS) solver or a Marvok Net (LAMP). Among the domain learners (Rodrigues *et al.* 2010; Pasula *et al.* 2007) that deal with noise in the input data, AMAN (Zhuo and Kambhampati 2013) considers every input plan trace as noisy and fits a collection of models from them. Other solutions (Mourao *et al.* 2012; Halbritter and Geibel 2007; Asai and Fukunaga 2017) use data mining techniques to guide the learning process and can handle noise and incompleteness over observed states. Finally, solutions like (Lanchas *et al.* 2007) apply regression techniques to deal with numeric fluents.

Only a few of these solutions can handle noise and/or incompleteness, the rest of them need complete or noiseless information over observed states in order to learn correctly. Those domain learners that deal with both incompleteness and noise can't learn planning domains with numerical information and they generate black-box models hard to interpret by a human. Finally, some approaches that deal with numeric fluents only deal with this kind of information leaving aside core elements of planning models such as logical fluents. A big drawback of these approaches is that even if they can generate domains with low error rates, they usually don't provide information about

the learned domains' validity. This information is crucial to determine if a domain can be used and can be easily tested with tools like(Howey and Long 2003b). PlanMiner-O2's main objective is to be able to learn valid planning domain's action models using incomplete and noisy observed states using the simplest machine learning models possible. The scope of the numerical information learned is restricted to assignations, increments and decrements of numerical predicates.

PlanMiner-O2 has been tested using data extracted from problems solved using benchmark domains of the International Planning Community. The data was modified randomly with noise and incompleteness in order to prove our hypothesis. The learned domains were compared by measuring the difference between them and the benchmark domains but also by testing its problem-solving capabilities.

Next section will cover in detail every background concept needed to understand how our solution works. In section 3 our domain learner algorithm will be explained. Then, section 4 will contain the information about our experiments and its results. Finally, in section 5 the conclusions drawn from the results will be discussed together with possible improvements of our solution in the near future.

2 The Learning Problem

By considering the domain learning problem as a classification problem we reduce the problem to find a collection of hypothesis that model a collection of states of the world. In AP the world is represented as a conjunction of fluents. A fluent is a statement in the form of $p(arg_1, arg_2, ..., arg_n)$ where p is a logic predicate and arg_x an object of the world. Objects may have a type associated, and those types may have a hierarchical relationship with other types. Each fluent has a value associated: True or False in the case of literal fluents or a numerical value in the case of function fluents.

In the other hand, a planning domain can be seen as a tuple $< Ont, Act >$ where Ont is the ontology of the world, the definition of the predicates and objects of the world, and Act is a collection of PDDL actions. In the same way, a PDDL planning action is a tuple $< Name, Par, Pre, Eff >$, where $Name$ is the action's name, *Par* are the parameters of the action, *Pre* the preconditions that must be true to allow the execution of the action and *Eff* the effects of the action in the world after being executed. An action whose *Par*, *Pre* and *Eff* are not instantiated with world's objects is called Action Model. This paper focuses on the learning of deterministic action models. In these action models, its preconditions and effects are unique among the rest of the domain's preconditions and effects. Finally, a *plan* is an ordered sequence of instantiated actions whose execution modify the world to achieve a given goal.

A *Plan Trace* (PT) is a plan with interleaved states S_x between the plan's actions A_x $< S_0, A_0, S_1, A_1, ..., S_n, A_n, S_{n+1} >$. Where S_0 is the initial state of the problem solved by this plan, S_{n+1} is the goal state of that problem and the rest of states

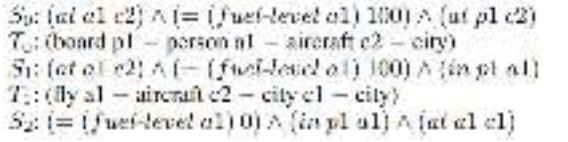


Figure 1: Extract of a PT from a Zeno Travel problem.

are snapshots of the world in a given point during the execution of the plan. Each action has an associated prestate and poststate. The state S_x of an action A_x is the prestate associated with the action and can be seen as the world just before executing the action. In the same way, the state S_{x+1} is the poststate associated with A_x and is the state of the world just after executing the action. Figure 1 shows an example of a PT.

The world's states of a PT are usually observed during the execution of a given plan. This can lead to have partially (incomplete) or wrongly (noisy) states observed. Incompleteness occurs when some fluents of the state (or the whole state) are not observed. Noise, on the other hand, is a problem where the value of a fluent is different of the value of the observed fluent. Following the states shown in Figure 1 an incomplete version of state S_0 would be $(at\ p1\ c2) \wedge (= (fuel\text{-}level\ a1)\ 50) \wedge (at\ a1\ c2)$ an example of noise over the same state.

3 PlanMiner-O2

PlanMiner-O2, the algorithm presented in this paper, first extracts the actions of an input set of plan traces with its associated prestates and poststates grouping them by the action's names. Then, the information of the states associated with a given action is included in a dataset, and after a preprocessing stage, and sent to a classification algorithm in order to obtain a classification model. The process is repeated until every action of the domain has been covered. The classification models contain the information needed to discern between the pre-states and post-states of an action and disassociates them. This disassociation helps to extract the maximum information by obtaining the tuples $< attribute, value >$ that form both the prestates and post-states of the action. In the final stages of PlanMiner-O2 this disassociation can be reversed easily in order to create the PDDL action models of the domain. Algorithm 1 shows an overview of PlanMiner-O2 where PTs is the collection of input plan traces. The main steps of PlanMiner-O2 are:

- **EXTRACT_INFO**. Extracts the information contained in PTs generating a dataset for a given $action$ that contains the information of the prestates and post-states associated with $action$.
- **PREPROCESS**. Increases PlanMiner's tolerance to incompleteness and noise by applying an array of techniques to the datasets (explained later).
- **LEARN_RULES**. This step uses a classification algorithm that takes a $Dataset$ and outputs a set of rules for each class. Each of these rules model a set of examples of the given class.

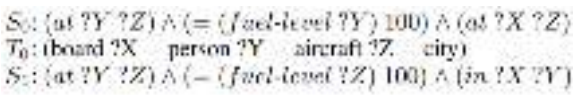


Figure 2: Schema form of an action and its associated states.

- **COMBINE** takes a set of rules and combines those rules whose class is the same in a single rule. Finally, the output rules are post-processed in order to create an action model.

Algorithm 1 Plan Miner algorithm overview.

Input: A collection of Plan Traces.
Output: A set of learned action models.

PlanMiner-O2(PTs)

1. ActM ← {}
2. **Foreach** $action$ **in** PTs, **Do**
   (a) $Dataset$ ← EXTRACT_INFO($action$, PTs)
   (b) $Dataset$ ← PREPROCESS($Dataset$)
   (c) $rules$ ← LEARN_RULES($Dataset$)
   (d) ActM ← ActM + COMBINE(rules)
3. **Return** ActM

EXTRACT_INFO creates a dataset for each different $action$ in PTs. Given an $action$ of the domain to be learned it extracts from the set of plan traces PTs every pair $< prestate, poststate >$ associated with it. An action in a PT consist in the header of a PDDL action with its parameters instantiated. Then, the procedure calculates the schema form of every pair by taking each argument $< arg_1, arg_2, ..., arg_n >$ of the action and replacing the i-th argument in every prestate's and poststate's fluent in which it appears with a $Param_i$ token that represents a variable. Finally, every fluent in the state that has not undergone at least one substitution is erased from the state following a criterion of *relevance* (Yang *et al.* 2007). A fluent is relevant if it shares anyone of its parameters with the associated action's parameters. Figure 2 shows an example of the schema form of the action board presented in Figure1 and its associated states.

In order to use the information contained inside a PT in a classification algorithm, datasets must be created from the extracted states. Those datasets are common in machine learning and are described as size $n * m$ matrices where n is the number of examples of the dataset and m the number of attributes. In the next pages, we are going to use planning terms instead of machine learning term when referring to the learning data. So will lead to using the term "state" when talking about a dataset's example or "fluent" when referring to an attribute.

$Fluent_1$	$Fluent_2$	...	$Fluent_m$	Class
$Value_{11}$	$Value_{12}$	...	$Value_{1m}$	$Class_1$
$Value_{21}$	$Value_{22}$	...	$Value_{2m}$	$Class_2$
...	...	...	...	...
$Value_{n1}$	$Value_{n2}$	...	$Value_{nm}$	$Class_n$

$Fluent_j$ are the elements which make up the state S_i, and $Value_{ij}$ the values of those fluents. $Value_{ij}$ depends on the $Fluent_j$ type. Literal fluents values can be True or False, while function fluents values are a numerical value. Dataset class are $prestate$ or $poststate$ depending on the relation of the state with the given action. When representing states with a different number of fluents, the set of all attributes is calculated as the union of the different sets of fluents of each example. If a fluent doesn't appear in an example its value is set as a Missing Value. Dataset's Missing Values (MV) are treated depending on the world assumption made in the planning domain: When interpreting the incompleteness of a state two interpretations can be used: the Closed World Assumption (CWA) or the Open World Assumption (OWA). CWA interprets the world by considering that unobserved fluents are false. Meanwhile, OWA considers that unobserved fluents are missing, nor true or false, and can't be evaluated. PlanMiner-O2 follows the OWA interpretation. By not considering the lack of information as falsity, PlanMiner-O2 is able to accurately produce planning domains.

F_1	F_2	F_3	F_4	Class
true	100	*true*	MV	*Prestate*
false	0	MV	*false*	*Poststate*

Example dataset using the data of Figure 2. From left to right, F_x relate to: $(at\ ?Y\ ?X)$, $(fuel\text{-}level\ ?Y)$, $(at\ ?X\ ?Z)$ and $(in\ ?X\ ?Y)$.

Before beginning the learning process, **PREPROCESS** function cleans the dataset and adds new attributes to it, helping the classification algorithm to learn correctly. Noise reduction is realized over the state's information by applying various techniques. These noise reduction techniques are applied differently depending on the type of the information contained in the attribute. The cleaning process to attributes with logical information is done by erasing noisy and missing values following the next procedure: On the one hand, noisy values are detected by calculating the appearance frequency for each attribute's values in relation to the dataset's classes. If the frequency of a certain attribute's value is lower than a threshold it is replaced by an MV in the dataset. Wrongly setting the threshold value will lead to the learning algorithm to detect low appearance rate values as noise, affecting the learned domains by discarding needed fluents in the action models. On the other hand, those examples of the dataset whose attributes only contains MVs are erased. This ensures that every example in the dataset had at least one attribute with useful information, minimizing noise problems.

PREPROCESS function add new attributes to each example of the dataset. These new attributes are calculated by selecting the different numerical fluents of the dataset's examples and computing the difference between the fluent's values of each associated prestate

and poststate included in the dataset. These new values are added to the given post-state's dataset row. If there's an MV in one of the states this difference can't be calculated.

LEARN_RULES uses the NSLV (New SLaVe) (González and Pérez 2009) algorithm used to learn the rules that model action's states. NSLV is a classification algorithm based on inductive rule learning. The rules created by NSLV use a weighted Disjunctive Normal Form (DNF) model, following the structure detailed below:

IF C_1 and C_2 and ... and C_m **THEN** $Class$ is B

with **weight w**

where a condition C_i is a sentence X_h is A, with A a label (or a set of labels) of the domain of the variable X_n. X_n is an element of X the set of antecedent variables of the rule, those antecedent correspond with the attributes of the problem's dataset. The domain of the variables that correspond with logical fluents contains only two labels to model "True" or "False", while numerical variable's domains contain a label for each different value in the examples of the corresponding variable. Finally, B is the value that represents a class of a particular problem and *w* a measure of the quality of the rule.

NSLV uses the Sequential Covering(Mitchell 1997) (SC) strategy described in Algorithm 2, where E is a collection of examples. The main steps of the SC strategy used are:

- **PERFORMANCE**. Measures the difference in the degree of completeness that causes the inclusion of a given rule in the ruleset. In other words, it measures the number of new examples of E explained by the addition of the rule in the collection of previously learned rules.
- **LEARN_ONE_RULE**. Uses a genetic algorithm(GA) to select which tuples $< attribute, value >$ define the antecedent of the rule that best fits a set of examples E. The rule learned must cover at least one example of E. The GA used is a steady state genetic algorithm whose population size maintains constant: each time an element is included in the population the worst element of it is erased.

Starting from an empty ruleset, a new rule is extracted and added to it in each iteration. The examples covered by this new rule are penalized (step 3.b) in order to guide the GA to learn rules that explain new examples. Penalization is realized by marking the examples instead of erasing them from the examples set. Marked examples are avoided in the next iterations of the learner, helping the algorithm to find a new rule that explains new examples besides previously covered examples. This process ends when the **PERFORMANCE** of an extracted rule is zero or less.

The criterion used to select the best rule is a key element in NSLV. The criterion defined uses a multicriteria evaluation guided by a Lexicographical Evaluation Function(LEF). The evaluation function's different criteria are ordered by their importance level. This order is essential to assure the rule's accuracy and interpretability level. The measures used are: completeness, consistency and simplicity. Ordered by its importance level.

Algorithm 2 Sequential covering strategy of NSLV

Input: A set of examples and a fitness function.
Output: A learned ruleset.

SEQUENTIAL_COVERING (E, f)

1. Learned_rules ← {}
2. Rule ← LEARN_ONE_RULE (E, f)
3. **While** PERFORMANCE (Rule, E) > 0, **Do**
   (a) Learned_rules ← Learned_rules + Rule
   (b) E ← Penalize (Learned_rules, E)
   (c) Rule ← LEARN_ONE_RULE (E, f)
4. **Return** Learned_rules

NSLV is able to output two different types of DNF rules: descriptive rules and predictive rules. Predictive rules contain the minimum information needed to classify an example, while descriptive rules contain the minimum information needed to model an example. In terms of information, the difference between predictive and descriptive rules is that predictive rules contain only the minimum information needed to select those examples that match the rule's class. On the other hand, descriptive rules contain every relevant attribute of the set of examples covered by the rule. PlanMiner-O2 uses descriptive rules to create the domain's actions models. Descriptive rules filter the dataset's attributes giving only those tuples $< attribute, value >$ important to model a set of examples, ignoring the rest. Another benefit of use NSLV is that we can learn both numerical and logic fluent together. This helps to use the classifier to find relations between the different types of attributes.

COMBINE takes a ruleset and blends it into a single rule. This ruleset is a subset of the ruleset generated using NSLV where every rule has the same class. COMBINE is called until every class of the problem has been covered. In a noise-free dataset, the number of rules learned by NSLV is always 2, one for each different class (pre-states and post-states), so rules combination is not necessary. COMBINE algorithm can be seen in Algorithm 3

Then, COMBINE shorts the fluents by its associated numerical value and tries to insert, in order, each one in a new rule with an empty antecedent. If there's no conflict between the rule and the fluent then the fluent is added to the rule's antecedent. A fluent can create a conflict in the rule if there's already a fluent in the rule's antecedent with the same predicate but different value. If so, COMBINE procedure computes the accuracy of the rule before and after substitute the conflicting fluent with the new one, and then decides about how to solve it.

EXTRACT_FLUENTS extracts each different fluent contained in the rules' antecedent and assigns a numerical value to each fluent. This value is the

Algorithm 3 Rule combination algorithm

Input: A collection of DNF rules and a dataset.
Output: A single DNF rule.

COMBINE($Ruleset$, $Dataset$)

1. $rule = \{\}$
2. $Fluents$ = EXTRACT_FLUENTS($Ruleset$)
3. **For each** $fluent$ **in** $Fluents$:
   (a) **If** CONFLICTIVE($fluent$, $rule$):
      i. $newrule$ = REPLACE($rule$, $fluent$)
      ii. **If** DIFF($rule$, $newrule$, $Dataset$) > 0.05:
         A. DEL_CONFLICT($rule$, $fluent$)
   (b) **Else**:
      i. $rulePre \leftarrow fluent$
4. **Return** rule

number of examples covered by the rule. If the fluent was already extracted from a different rule's antecedent the number of examples covered is added to the fluent's associated numerical value. Finally, EXTRACT_FLUENTS sorts the fluents in descendent order by its numerical value.

CONFLICTIVE function returns if there's a conflict between a $fluent$ and the $rule$. For example if $(at\ Param_1\ Param_2) = false$ is in the antecedent of $rule$ it will create a conflict with the $fluent\ (at\ Param_1\ Param_2) = true$. If the difference between the accuracy changes of the new rule and the old one is relevant enough, Replace function is called, otherwise, DEL_CONFLICT is used. REPLACE functions swaps the conflictive fluents in the $rule$. Finally, DEL_CONFLICT erases the conflictive fluent of the antecedent of the rule.

Once the rules are combined, COMBINE converts the DNF rules to a PDDL action following a straightforward process:

- Action's preconditions are taken directly from the prestate class' rule antecedent. Antecedent's attributes are translated directly into predicates whose value is the attribute value.
- Action's effects are extracted from the difference $\Delta(pre, post)$ between the prestate rule's antecedent and the poststate rule's antecedent. $\Delta(pre, post)$ is defined as the set of changes that must be done over pre in order to make it equal to $post$

Numerical fluents are converted to a numerical PDDL numerical precondition or effect by taking the CRISP value associated with the set assigned during the **LEARN_RULES** procedure. COMBINE differentiates between the original numerical attributes of the dataset and the new ones added during the **EXTRACT_INFO** step. Artificially added numerical attributes are translated as $(increase/decrease\ (fluent)\ value)$ according to its value.

Once the whole learning process has finished the rest of the PDDL planning domain is created by simply adding the list of different types' and parametrized fluents extracted from PT to it. PlanMiner-O2 is able to generate OWA and CWA action models. When calculated the preconditions and effects for a certain action, PlanMiner-O2 can generate them only deciding to explicitly represent the negative fluents or not.

4 Experiments and Results

PlanMiner-O2[1] was tested using a collection domains from the International Planning Competition IPC. The objective of these experiments is to demonstrate that PlanMiner-O2 is able to learn planning domain's action models with high levels of missing states' information and some levels of noise. In order to demonstrate that PlanMiner-O2 can learn numerical information, a number of benchmark domains can use function fluents and action costs. These domains are Driverlog, Satellite and Zenotravel. The details of the domains used can be seen in Table 1. From each domain, 200 problems were set. The 80% of these problems were used as training problems and the 20% left as test problems. Problems were solved using a goal directed planner, not looking for optimal plans. The experimental process used was defined as follows:

1. Training problems were solved using the handcrafted planning domain.
2. For each plan obtained in Step 1, a PT was created.
3. PTs were modified with noise or incompleteness if applicable.
4. A new domain was learned from the collection of PTs.
5. The learned domain and the original one were compared and performance values were calculated.
6. Test problems are solved using the learned domain.
7. New plans generated in Step 6 were validated with the original domain.

In order to ensure the results, a 10 fold cross-validation was used. The final result is the average of the results obtained in Steps 5 and 7 in each validation. Noise and incompleteness affect only to the PT's states and were included in the PT's states randomly, first, by changing the value of a given percentage of fluents selected randomly and then, by erasing a given percentage of fluents selected randomly. Noise was also included by adding new possible fluents that can not be found in the given states. The threshold value used to discern between noise or not during the preprocessing stage is set to 0.05%. The parameters of NSLV's genetic algorithm (population size, crossover and mutation probabilities) are automatically generated by the classifier.

Performance is measured using 2 different criterions:

- Learned domain's error rate.
- Learned domain's validation rate with test problems.

[1] PlanMiner-O2 and experimental data used can be found at https://github.com/Leontes/PlanMiner

Problem	$\|actions\|$	$\|fluents\|$	$\tilde{PL}$	$\tilde{SL}$	$\tilde{CPUt}$
BlocksWorld	4	5	600	500	100
Depots	5	6	236	381	83
DriverLog	6	6	173	169	70
ZenoTravel	5	4	165	95	40
Satellite	5	8	91	178	37
Parking	4	5	57	200	98

Table 1: Benchmark Domains Characteristics(from left to right): domain's number of actions, domain's number of fluents, average number of actions in the plans solved, average number of fluents in the plans' states and average CPU time(in seconds) to learn a domain.

Noise %	0%				5%			
Incompleteness %	0%	10%	50%	90%	0%	10%	50%	90%
BlocksWorld	✓	✓	✓	✓	✓	✓	✓	✓
Depots	✓	✓	✓	✓	✓	✓	✓	✓
DriverLog	✓	✓	✓	✓	✓	✓	✓	✓
Satellite	✓	✓	✓	✓	✓	✓	✓	✓
ZenoTravel	✓	✓	✓	✓	✓	✓	✓	✓
Parking	✓	✓	✓	✓	✓	✓	✓	✓

Noise %	10%				20%			
Incompleteness %	0%	10%	50%	90%	0%	10%	50%	90%
BlocksWorld	✓	✓	✓	✓	X	X	X	X
Depots	X	X	X	X	X	X	X	X
DriverLog	✓	✓	X	X	X	X	X	X
Satellite	X	X	X	X	X	X	X	X
ZenoTravel	X	X	X	X	X	X	X	X
Parking	X	X	X	X	X	X	X	X

Table 2: Domains validity matrix.

The criterion used to measure the quality of the learned domains is the domain's error rate (Zhuo *et al.* 2010). Domain's error rate is measured by comparing the learned domain with the original one. Domain's error is defined as $\frac{\sum_{a \in Actions} error(a)}{|Actions|}$ where $Actions$ is the set of Actions of a given domain. Action's error rate, $error(a)$, is computed by counting the number of missing or extra fluents in the learned action's preconditions and effects and dividing it between the number of possible fluents in those preconditions and effects. When counting fluents, we take into account that two fluents are equally semantically rather than syntactically.

The results showed in Figure 3 demonstrate that our solution can model planning domains close to the original handmade planning domains: error rates fall below 4% even with high levels of incompleteness. In fact, incompleteness affects little to the learning process. In the worst cases, domains' error rate doesn't rise beyond 7% with high levels of incompleteness and some levels of noise. Using complete and noiseless plan traces, PlanMiner-O2 achieves easily zero error. In those cases where even with this kind of plan traces PlanMiner-O2 learns domains with some errors, the error is produced because PlanMiner-O2 doesn't use information of how the fluents relate with others fluents provoking an overfitting of some models of the states. This overfit is produced because without this information, PlanMiner-O2 can't discern which fluents are redundant in a state. For example, Satellite and Parking domains' error rate in the experiments without noise is produced by, in both cases, the existence of a single fluent with a given value in some preconditions. A human expert designing these domains would have omitted these fluents because it can be inferred using other fluents' values.

The second measure of the quality of the learned domains is the domain's validity. Plan validation is calculated using the set of test problems selected for each domain. A domain is valid if every plan obtained with it can be validated using the original domain. Plan validation is realized using VAL(Howey and Long 2003a), an automatic validation tool used in the IPC. Roughly, VAL takes a problem, a plan and a planning domain and executes the plan's actions in order over the initial state defined in the problem using the action's definition contained in the planning domain. A plan is valid if the resultant state of applying every plan's action is equal to the problem's goal state.

PlanMiner-O2 learned domains' validity calculated using VAL can be seen in Table 2. These results demonstrate that PlanMiner-O2 outputs valid domains even with high levels of noise. Validity criterion is the hardest one to meet because a single error in the effects can make a domain not valid. Validity is a binary measure because during our test we encounter that if a single problem couldn't be validated with a given domain any of the other problems couldn't either. Redundant or missing fluents in the preconditions can be tolerated sometimes when measuring validity, but in the effects, those errors lead to incorrect plans and hence invalid domains. The accuracy rate of the rules was calculated too as a secondary quality measure. NSLV maintained an accuracy rate above the 90% of success even with 10% noisy information during the learning process. In the worst cases, accuracy never fell below of the 60% of success.

Comparing our solution with other state of the art solution of the bibliography like (Yang *et al.* 2007) we can see that PlanMiner-O2 can learn planning domains with lower error rates. Comparisons with other approaches in the literature are difficult due to differences in the learning settings. But using the same testing domains, PlanMiner-O2 learned domains show lower rates even with some levels of noise in the input data: with a 90% (the highest reported) of missing fluents and no noise (this solution only deals with incomplete input data) the error rates of the domains learned by (Yang *et al.* 2007) range from above 0.2 (Depots domain) to less than 0.6 (Satellite). PlanMiner-O2 results are notably better even with some levels of noise. Using another newer solution able to deal with both noise and incompleteness to compare we get similar results. The domains learned by (Mourao *et al.* 2012) show an error rate ranging from above 0.04 (Zenotravel) to 0.1 (BlocksWorld, Depots, Driverlog) in the worst cases (90% missing fluents and 5% noise). Our learned domains' error rates maintain below 0.016 in every domain even with higher levels of noise. Furthermore, our learning process is fast. Using a single thread of an Intel Core i7-6700 CPU and a 16 GB RAM the learning process took less than 4 minutes with the hardest problem. With high levels of incompleteness, processing time fell, in some cases, below 10 seconds.

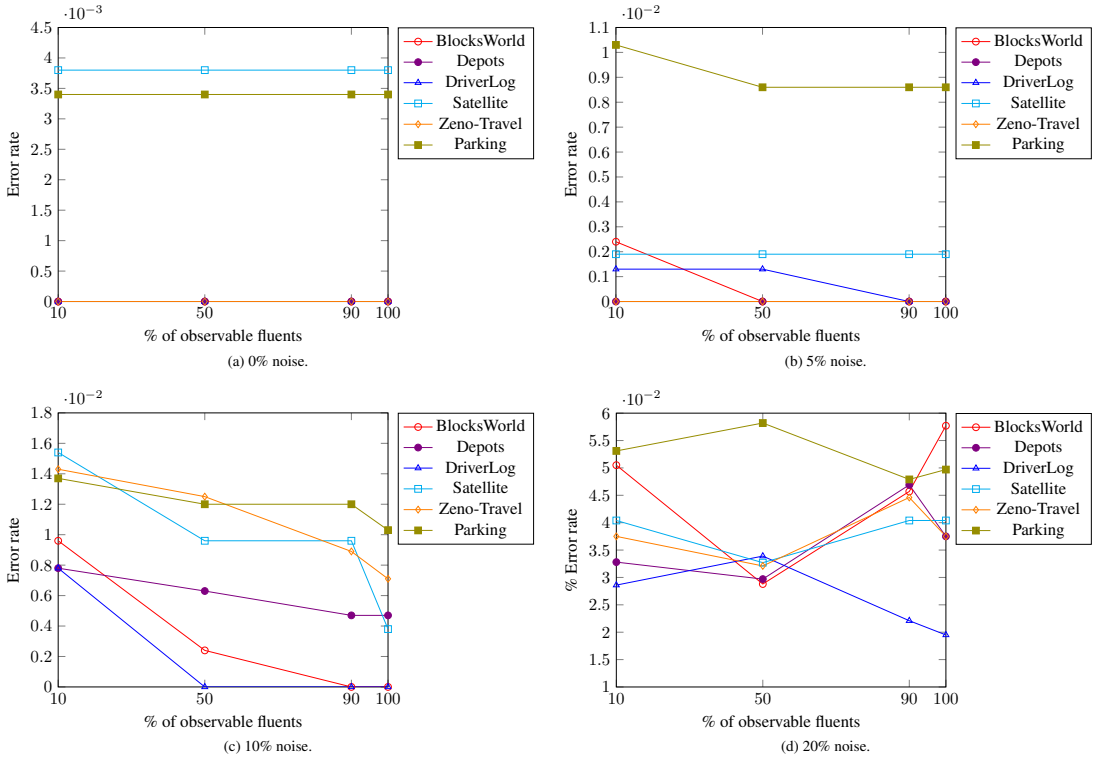


Figure 3: Learned domains error rates.

5 Conclusions and Future Work

In this paper, we have developed a new planning domain learner (PlanMiner-O2) that uses a data mining classification algorithm to learn the domains' action models with logical and numerical information from incomplete and noisy information. The results obtained show that PlanMiner-O2 is able to learn actions models, even with high levels of incompleteness and some levels of noise. In fact, experiments showed that incompleteness affects little to the results of PlanMiner-O2, even with low levels of noise. The experiments carried out focus on the reduction of the domain's error but also in the test of the validity of the domains solving a set of benchmark planning problems.

Our next steps will lead to improving the robustness of PlanMiner-O2 when dealing with noise. As said earlier, a single error in an action model's effects can lead to making the whole domain invalid. We want to focus on the reduction of the effect's error rate in order to improve the validity of our learned domains. As NSLV accuracy rate modelling states is high enough to fit our expectations our focus will be on the development some new pre-process procedure to modify the information before sending it to NSLV and some post-process procedure to manage its output better.

In order to improve the capabilities of PlanMiner-O2 to deal with more complex problems are going to include the capability of dealing with continuous numerical fluents in it. As NSLV is already able to deal with this kind of information by using discrete sets our work will focus on a procedure to extract and manage correctly this kind of information before sending it to the classification algorithm.

PlanMiner-O2 is the first of a new family of domains learners we are going to develop. These new domain learners aim to learn HTN planning domains from real-world data. The PlanMiner-O2 algorithm is the first solution that addresses part of this challenge. Next versions will include new functionalities to deal with real-world data or learn the hierarchical structures needed in HTN Planning.

References

[Asai and Fukunaga 2017] Masataro Asai and Alex Fukunaga. Classical planning in latent space: From unlabeled images to pddl (and back). *The ICAPS-2017 Workshop on Knowledge Engineering for Planning and Scheduling*, 2017.

[Benson 1996] Scott Sherwood Benson. *Learning Action Models for Reactive Autonomous Agents*. PhD thesis, stanford university PhD thesis, 1996.

[Cresswell *et al.* 2013] Stephen N. Cresswell, Thomas L. McCluskey, and Margaret M. West. Acquiring planning domain models using locm. *The Knowledge Engineering Review*, 28(2):195–213, 2013.

[Gil 1994] Yolanda Gil. Learning by experimentation: Incremental refinement of incomplete planning domains. In William W. Cohen and Haym Hirsh, edi-

This research is being developed and partially funded by the Spanish MINECO R&D Project PLAN MINER TIN2015-71618-R

tors, *Machine Learning Proceedings 1994*, pages 87 – 95. Morgan Kaufmann, San Francisco (CA), 1994.

[González and Pérez 2009] A. González and R. Pérez. Improving the genetic algorithm of slave. *Mathware and Soft Computing*, 16:59–70, 2009.

[Gregory and Lindsay 2016] Peter Gregory and Alan Lindsay. Domain model acquisition in domains with action costs. In *ICAPS*, pages 149–157, 2016.

[Halbritter and Geibel 2007] Florian Halbritter and Peter Geibel. Learning models of relational mdps using graph kernels. *MICAI 2007: Advances in Artificial Intelligence*, pages 409–419, 2007.

[Hayton *et al.* 2016] Thomas Hayton, Peter Gregory, Alan Lindsay, and Julie Porteous. Best-fit action-cost domain model acquisition and its application to authorship in interactive narrative. In *AIIDE*, 2016.

[Howey and Long 2003a] R. Howey and D. Long. Val's progress The automatic validation tool for pddl2.1 used in the international planning competition. In *Proceedings of the ICAPS 2003 workshop on The Competition: Impact, Organization, Evaluation, Benchmarks*, pages 28–37, Trento, Italy, June 2003.

[Howey and Long 2003b] Richard Howey and Derek Long. VAL's Progress: The Automatic Validation Tool for PDDL2.1 used in the International Planning Competition. In *Proceedings of ICAPS'03 Workshop on the Competition: Impact, Organization, Evaluation, Benchmarks*, Trento, Italy, June 2003.

[Jiménez *et al.* 2012] S. Jiménez, T. De La Rosa, S. Fernández, F. Fernández, and D. Borrajo. A review of machine learning for automated planning. *The Knowledge Engineering Review*, 27(4):433–467, 2012.

[Lanchas *et al.* 2007] Jesús Lanchas, Sergio Jiménez, Fernando Fernández, and Daniel Borrajo. Learning action durations from executions. In *Proceedings of the ICAPS*. Citeseer, 2007.

[Mitchell 1997] Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997.

[Mourao *et al.* 2012] K. Mourao, L. S. Zettlemoyer, R. P. A. Petrick, and M. Steedman. Learning STRIPS operators from noisy and incomplete observations. *Proceedings of the Twenty-Eighth Conference on Uncertainty in Artificial Intelligence*, 2012.

[Pasula *et al.* 2007] Hanna M Pasula, Luke S Zettlemoyer, and Leslie Pack Kaelbling. Learning symbolic models of stochastic domains. *Journal of Artificial Intelligence Research*, 29:309–352, 2007.

[Rodrigues *et al.* 2010] Christophe Rodrigues, Pierre Gérard, and Céline Rouveirol. Incremental learning of relational action models in noisy environments. In *ILP*, pages 206–213. Springer, 2010.

[Wang 1995] Xuemei Wang. Learning by observation and practice: An incremental approach for planning operator acquisition. In *In Proceedings of the 12th International Conference on Machine Learning*, pages 549–557. Morgan Kaufmann, 1995.

[Yang *et al.* 2007] Q. Yang, K. Wu, and Y. Jiang. Learning action models from plan examples using weighted MAX-SAT. *Artificial Intelligence Journal.*, page 107–143, 2007.

[Zhuo and Kambhampati 2013] H. H. Zhuo and S. Kambhampati. Action-model acquisition from noisy plan traces. *Proceedings of the Twenty-Third international joint conference on Artificial Intelligence.*, pages 2444–2450, 2013.

[Zhuo *et al.* 2010] Hankz Hankui Zhuo, Qiang Yang, Derek Hao Hu, and Lei Li. Learning complex action models with quantifiers and logical implications. *Artificial Intelligence*, 174(18):1540 – 1569, 2010.

On the use of ontologies to extend knowledge in online planning

Mohannad Babli, Eliseo Marzal, Eva Onaindia
Department of Computer Systems and Computation
Universitat Politècnica de València, Valencia, Spain
{mobab, emarzal, onaindia}@dsic.upv.es

Abstract

Despite the progress in online planning, goal driven autonomy, and opportunistic planning, agents still need to be fed by carefully engineered models that are fine tuned for particular applications. Approaches to goal-directed behaviour tackle a change in the environment by generating alternative goals to avoid failures or seize opportunities. However, current approaches only address unanticipated changes related to objects or object types already defined in the planning task that is being solved. Hence, agents lack autonomy because they still rely on the prior knowledge of their designers rather than their own percepts. This article describes a domain-independent approach that advances the state of the art by extending the knowledge of a planning task with relevant objects of new types. The approach draws upon the use of automated planning and ontologies to accommodate new acquired data that trigger the formulation of goal opportunities inducing a better-valued plan, thus bolstering the agent with higher autonomy capabilities.

Introduction

Planning research has been mostly devoted to offline planning with some incursions in online plan-repair to address failures during the plan execution. Whilst online planning has demonstrated its usefulness to handle plan failures, unanticipated events that bring about an opportunity for the task at hand has been rarely studied. Goal-directed behaviour (GDB) is a hallmark of intelligence widely used for high levels of autonomy when the environment is dynamic, partially observable, and open to new data (Vattam et al. 2013). In GDB, the agent monitors the execution of the plan in the environment and it is capable of formulating alternative goals on the fly (Cox 2007; Dannenhauer and Muñoz-Avila 2013; Klenk, Molineaux, and Aha 2013). One limitation of most of the current GDB approaches is that goals are formulated on the basis of objects that already exist in the agent model. An exception to this can be found in (Cashmore et al. 2017), an approach to *opportunistic planning* which allows the agent to generate new goals involving objects that are not present in its current model. Nevertheless, the new object must be of one of the predefined classes (types) in the agent model. Hence, intelligent agents still rely on the prior knowledge of their designers rather than their own percepts, therefore they lack autonomy (Russell and Norvig 2010).

The motivation of this work is to overcome the general lack of research in GDB towards the formulation of goal opportunities that stem from objects or object classes that are unknown to the agent at design time. Specifically, given a planning task and a plan (sequence of actions) that solves the task, the process initiates with the execution of such a plan. While executing the plan, the external events received from the are classified into three categories: events that confirm the correct execution of the plan actions; events that bring about a failure in the plan execution; or events that may induce a new goal opportunity in the context of the planning task and the plan. This paper puts the focus on the latter and proposes an approach to handle context-aware open planning tasks.

Our contribution is a domain-independent approach that extends the knowledge of a planning task with relevant objects extracted from a collection of ontologies that describe features of interest for the specific domain. The approach draws upon the richness and expressivity of standard ontology representations, semantic measures and ontology alignment for accommodating the new acquired objects into the planning task specification. These new objects may subsequently trigger the formulation of a goal that induces a better-valued plan.

Next section presents some basic notions on planning and the following two sections outline the components of our approach and the identification of new goal opportunities, respectively. Then, section *Cases of study* presents an example of application of the ontology-based approach and the last section concludes and outlines some future work.

Background

Consider a scenario of a repair agency in which a robot located in a warehouse has the task of a one-day maintenance of the electronics and furniture items that are received by the agency. The warehouse has three designated areas, a transit area for items that require maintenance, an inspection area for maintenance, and a stor-

age area for items after maintenance. The scenario is formulated as a planning task including a specified set of different categories of electronics and furniture, the operations that the agent is able to perform and their durations (movement between the warehouse areas, maintenance, loading, and unloading). The planner solves this task and returns a plan which includes a total repairing of four items: two items of type *television*, one item of type *refrigerator*, and one item of type *sofa*. During the plan execution, the repair agency receives a new item *bosch_ID3400* into the transit area from a different delivery agent that is operating in the same city. The new item type is found to be *dishwasher*, not formerly considered in the planning task of the repair agency. This may represent an opportunity if the goal of repairing the new object can be aligned within the modelling of the planning task of the agency and triggers a plan that also fits the current goals.

A planning task is defined as $\Phi = \langle \mathcal{D}, \mathcal{I} \rangle$, where $\mathcal{D}$ is the domain of the task (e.g. repair agency) and $\mathcal{I}$ is a particular problem instance (e.g. a one-day maintenance). The elements that define the domain are $\mathcal{D} = \langle \mathcal{T}, \mathcal{V}, \mathcal{A} \rangle$: $\mathcal{T}$ is the set of object types (e.g. types of items); $\mathcal{V}$ contains the set of boolean variables of the form $(p\ o_1 \dots o_n)$, where p is a predicate symbol, and arguments $\{o_i\}_{i=1}^{n}$ are of types included in $\mathcal{T}$ (e.g. `(be ?robot ?area)`); $\mathcal{A}$ is the available action schemas with headers $(a\ o_1 \dots o_m)$, $\{o_i\}_{i=1}^{m}$ are of types included in $\mathcal{T}$ (e.g. `(repair ?robot ?item)`). On the other hand, an instance is described by $\mathcal{I} = \langle \mathcal{O}, \mathcal{S}, \mathcal{G} \rangle$, where $\mathcal{O}$ is the set of objects (e.g. items to be repaired); $\mathcal{S}$ is a full assignment of values to variables in $\mathcal{V}$ that represent the current state of the problem ($|\mathcal{S}| = |\mathcal{V}|$ and initial values of $\mathcal{V}$ denote the initial state of the task); and $\mathcal{G}$ is a partial assignment of values to variables of $\mathcal{V}$ that represent the goals to be accomplished (e.g. set the variable `(repaired sofa_ID2005)` to `true`). The planner receives Φ as input and outputs a plan $\pi = \langle a_1, \dots, a_n \rangle$ composed of a sequence of ground actions (e.g. `(move av area_transit area_inspect) (load av sofa) ...`).

Overview of the approach

We first briefly introduce the plan monitoring and execution simulator our approach relies on (Babli et al. 2016). The simulator takes Φ and π as input and encodes them into a timeline as a collection of chronologically ordered timed events that encapsulate the changes to be expected in the subsequent states. The monitoring process of the simulation system simulates:

- receiving exogenous events and adds them to the timeline; exogenous events convey external information received from other agents operating in the same environment modifying the real world states in a dynamic manner
- the execution of the timed events, checking that conditions of the plan actions are satisfied and the effects happen when they should, thus validating and updating, respectively, the states of the world (timeline).

A sample of a timeline is shown in Figure 1, where timed events appear in chronological order, with the corresponding conditions to be satisfied and effects to be applied at each time point.

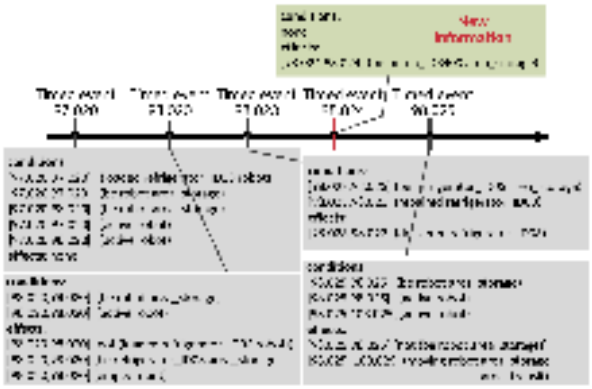


Figure 1: A sample of a timeline

Dynamically simulating plan monitoring consists in observing the state that results from executing the plan actions in the environment and checking whether the *observed state* $\mathcal{S}$ matches the *expected state*. This operation creates a *discrepancy set* as the difference between the two sets, which will comprise instantiated variables that are found in the observed state but not in the expected state and viceversa. After discarding the variables of the discrepancy set that represent a failure, the remaining variables denote a potentially achievable goal opportunity. More specifically, the discrepancy set will contain instantiated variables of the form $(p\ o_1 \dots o_n)$ where $\exists o \in \{o_i\}\ / o \notin \mathcal{O}$. In our approach the system requests the type t of the new object o from the source agent that delivered it; however, other advanced techniques can be used to identify the type of the object such as image recognition using tensorflow in deep learning, or finding the type (class) of the newly received objects (individual) from other ontologies e.g., Tourism ontologies in the context of Tourism. We aimed to distinguish three cases, $t \in \mathcal{T}$, literally one of the existing types; $t \equiv t' \in \mathcal{T}$, the new type is semantically equivalent to one of the existing types although syntactically different; and $t \notin \mathcal{T}$ a brand new type. Examples for the three previous cases of discrepancies in the repair agency domain is {`(be sofa_ID04 area_transit), (be couch_ID1400 area_transit), and (be bosch_ID3400 area_transit)`}, where the first object `sofa_ID04` is a new object that belongs to the existing type *sofa*, the second object `couch_ID1400` is a new object of a type *couch* that is semantically equivalent to the type *sofa* $\in \mathcal{T}$, and the object `bosch_ID3400` is a new object that belongs to the new type *dishwasher* $\notin \mathcal{T}$. The first case is trivial, on the

other hand, to be able to handle the second and third cases, and in order to position the new type into $\mathcal{T}$, generate new goals that includes o that will be handled using $\mathcal{A}$ we designed the approach sketched in Figure 2 which works in Four stages:

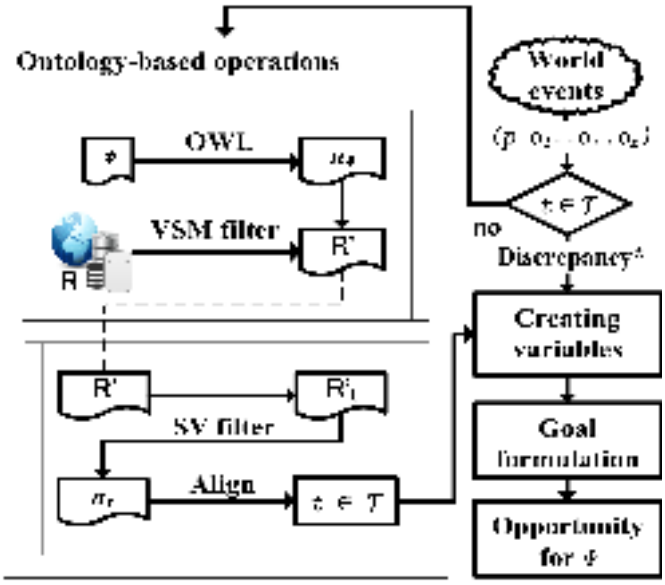


Figure 2: The ontology-based Goal formulation model

Stage 1: Identification of similar ontologies. First, we create an OWL ontological representation of the types $\mathcal{T}$ of Φ, called n_Φ, that will be the base ontological representation for the rest of stages. Second, we retrieve a set R of remote ontologies from on-line repositories. Subsequently, we apply a vector space distance similarity measure (VSM) to R and we obtain the set $\mathsf{R'}$, which contains the most similar ontologies to n_Φ.

Stage 2: Positioning a new object. When the information of a new object o ($o \notin \mathcal{O}$) is received in the form of a variable $(p\ o_1 \dots o \dots o_n)$, the agent identifies the type t of o from the source agent which delivered o. In case that $t \in \mathcal{T}$, (literally one of the existing type) we simply add o to $\mathcal{O}$. Otherwise, the system creates R'_t as the set of ontologies out of $\mathsf{R'}$ that contain t, and the ontology of R'_t that most accurately models the semantic knowledge of the application domain $\mathcal{D}$ is selected using the semantic variance measure (we will refer to this ontology as n_t). The system attempts to position t in n_Φ via a semantic alignment with a neighbourhood constraint between n_Φ and n_t. If the alignment is successful, t is either identified as an existing type in $\mathcal{T}$ (semantically equivalent although syntactically different), and simply o is added to $\mathcal{O}$, or t is found to be a new type that is positioned in the hierarchy of types, then t is added to $\mathcal{T}$ and o is added to $\mathcal{O}$.

Stage 3: Creating the new variables. If the new object o is successfully positioned in Φ, the next step is to instantiate the required planning variables $\mathcal{V}$, besides $(p\ o_1 \dots o \dots o_n)$, that describe o. The system automatically identifies the information required for integrating o in Φ (to be handled with the same action schemas) and adds it to $\mathcal{S}$. There exist multiple sources from which such information could be retrieved autonomously such as Open Data platforms, ontologies, or other agents with similar planning tasks.

Stage 4: Goal formulation. If the type t of object o is a type or a sibling of a type that is involved in a goal $g \in \mathcal{G}$, then we formulate x candidate new goals that involve the newly received object o, where x depends on the possible permutations of objects in the goal predicate; $x = 1$ if the goal has only o as a parameter such as $g = (q\ o)$ (e.g. $g =$`(repaired bosch_ID3400)`).

The following section detail the OWL ontological representation of the types, the identification of similar ontologies, the selection of the ontology with the highest semantic insight, and the alignment with a neighbourhood constraint.

Ontology-based operations

In this section we detail the tasks required to include new objects in the planning task specification.

OWL Ontological representation

OWL has been the World Wide Web Consortium recommendation since 2004 (Patel-Schneider, Hayes, and Horrocks 2014). In this section, we explain the generation of n_Φ, the ontological representation of the types $\mathcal{T}$ of a planning task Φ. We used *OWL API* which is an open source Java API and reference implementation for creating, manipulating, and serialising OWL Ontologies (Horridge and Bechhofer 2011). Throughout this section we use snapshots from the GUI of Protégé to show visual explanations of n_Φ.

The OWL ontological representation consists of a set $\mathcal{C}$ of concepts (OWL classes) that represent $\mathcal{T}$ and a set of OWL annotation properties that describe $\mathcal{C}$. A class $c \in \mathcal{C}$ can have one or many annotation properties.

On the other hand, the Planning Domain Description Language (PDDL) (Edelkamp and Hoffmann 2004) offers the ability to express a type structure for the objects in a domain, typing the parameters that appear in predicates and actions. Furthermore, types can be expressed as forming a particular type hierarchy. For each type in $\mathcal{T}$, an OWL class is created in n_Φ abiding the exact hierarchy [1].

The left part of Figure 3 shows the types $\mathcal{T}$ of the *repair agency* domain specified in PDDL and the right part of the figure shows the corresponding $\mathcal{C}$ in n_Φ. For instance, the type `sofa` is represented as furniture::sofa. We can observe the types of the planning task are arranged in a reasonable hierarchy and that the OWL representation follows truthfully this hierarchy. Although using real names of types that convey a semantic meaning and having the types arranged in a reasonable hierarchy do not affect the ontological representation, these

[1] The symbol :: is used to refer to a subclass, for instance, c_i::c_j means c_j is a subclass of c_i.

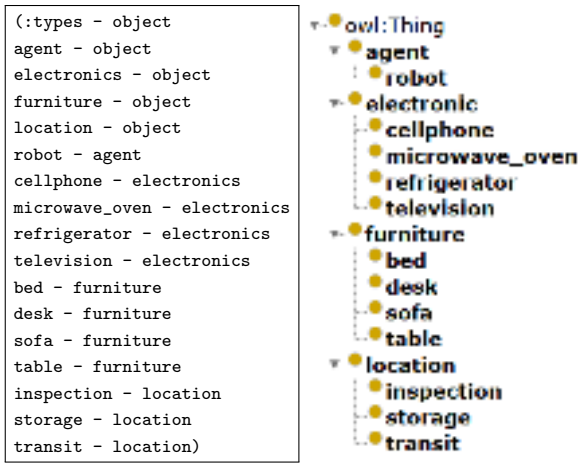


Figure 3: Representation of PDDL types

aspects are important to find similar ontologies to the domain and for positioning a type in $\mathcal{T}$. Nonetheless, the dependency of using real names and a significant type hierarchy in the system does not affect its domain-independent nature.

Identifying similar ontologies

In ontology engineering, it is useful to know quickly if two ontologies are close or remote before deciding to match them (David and Euzenat 2008). In this step, we measure the distance between n_Φ and the ontologies of R to filter out the unrelated ones and obtain R'. Furthermore, we need to tackle the natural complication that different people could model the same application domain using different terms, or even in different languages; e.g., ontologies A and B in Figure 4 model the *repair agency* domain using different terminology, and ontologies C and D model a *product delivery* domain also using different terminology.

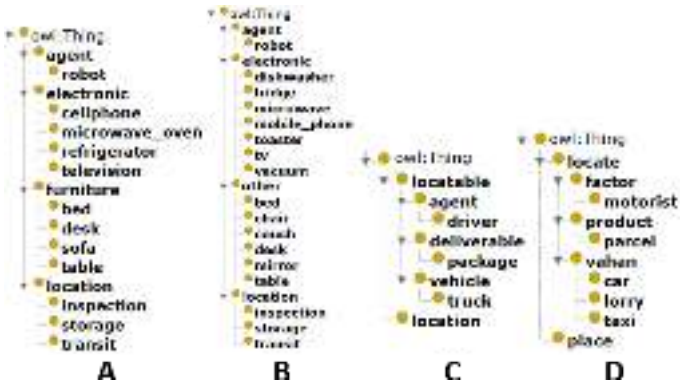


Figure 4: Using different terms when modelling ontologies

For that purpose, we use ConceptNet as a standard mean to describe the classes of the ontologies. ConceptNet (Speer, Chin, and Havasi 2017) is a knowledge graph that connects words and phrases of natural language using labelled edges. Its knowledge is obtained from various sources that combines expert-created resources, crowd-sourcing, and games with a purpose. ConceptNet utilises a closed class of 36 selected relations such as isA, usedFor, hasProperty, etc., with the aim of representing relationships independently of the language or the source of the terms it connects. Therefore, we augment the classes of n_Φ and of the ontologies of R with the relations and classes brought from ConceptNet as OWL annotations. As a result, even if the names of the classes are different, classes that refer to the same concept will have annotations in common and will be found similar when measuring semantic distances or when performing the alignment. Figure 5 shows a small portion of the forty eight annotations attached to the class sofa.

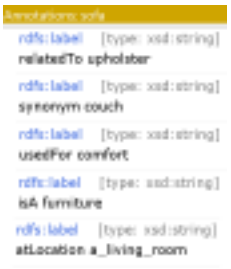


Figure 5: Annotations sample assigned to c_{sofa} class

For measuring the distance between ontologies we decided to look at the ontologies as a bag of terms and apply ontology distance measures based on the Vector Space Model (VSM) using cosine index with weighted term frequency (TF). VSM has proven to obtain good results compared to other distance measures and it is computed largely faster, but it is not much robust to lexical alterations (David and Euzenat 2008). However, lexical alterations do not impact our approach because the lexical information in each term of the ontology comes not only from the local name of the term but also from the OWL annotations imported from ConceptNet relations and classes. We used *OntoSim* to compute the distance, an independent Java API to compute similarities between ontologies that provides a variety of distance measures. At this stage, the ontologies R' with the highest similarity with respect to n_Φ are obtained.

Selecting the ontology with the highest semantic insight

The system calculates R'_t, the set of ontologies from R' that contain the type t. The next step is to select $n_t \in \mathsf{R}'_t$ that best fits the planning task Φ. For this purpose we decided to use semantic variance (SV) as introduced in (Sánchez et al. 2015) (SV filter in Figure 2). SV is an intuitive and inherently semantic measure to evaluate the accuracy of ontologies. Unlike ad hoc methods, SV is a mathematically extension of the standard numerical variance to measure the semantic dispersion of the taxonomic structure of ontologies. The

value of SV for an ontology n, which taxonomically models a set of concepts $\mathcal{C}$, is defined as the average of the squared distance between each concept $c_\mathrm{i} \in \mathcal{C}$ and the root node of n (Sánchez et al. 2015). The formula to compute SV is shown below:

$$SV = \frac{\sum_{c_\mathrm{i} \in \mathcal{C}} d(c_\mathrm{i}, \mathsf{root})}{|\mathcal{C}|}$$

where $|\mathcal{C}|$ is the cardinality of $\mathcal{C}$ excluding the root node and $d(c_\mathrm{i}, c_\mathrm{j})$ is the distance between two concepts c_i and c_j calculated as a function of the number of their non-common ancestors divided by their total number of ancestors:

$$d(c_\mathrm{i}, c_\mathrm{j}) = \log_2(1 + \frac{|A(c_\mathrm{i}) \ \cup \ A(c_\mathrm{j})| - |A(c_\mathrm{i}) \ \cap \ A(c_\mathrm{j})|}{|A(c_\mathrm{i}) \ \cup \ A(c_\mathrm{j})|})$$

The semantic distance d aggregates features in a logarithmic way, which better correlates with the non-linear nature of semantic evidences, and more importantly, variance does not depend on the cardinality of the ontology. We calculate the SV for each ontology in R'_t, and select n_t the one with the highest SV.

Alignment with neighbourhood constraint

The next step is to determine where to position the class c_t that corresponds to the new type t within the hierarchy of concepts $\mathcal{C}$ of n_ϕ. We perform an alignment between $\mathcal{C}$ of n_ϕ and the part of the taxonomic branch of n_t that includes c_t, the parent class $c_{\mathsf{parent(t)}}$, and the siblings $\mathcal{C}_{\mathsf{siblings(t)}}$. The alignment is the process of determining correspondences between concepts in ontologies. For the alignment we used CIDER-CL introduced in (Shvaiko et al. 2013), a schema-based ontology alignment system that compares the classes of two ontologies using also VSM. The result of the alignment is an RDF file that contains the degree of matching between two classes. We distinguish two main cases:

- The class c_t is found to be semantically equivalent (albeit they are syntactically different) to an existing type if c_t matches one of the classes in n_Φ with a matching degree above a specified threshold. In this case, we simply add the new object o to $\mathcal{O}$.
- The matching degree of class c_t with all the classes in n_Φ is below a certain threshold. In this case, we use the neighbourhood constraint as suggested in (Doan et al. 2003) where "*two nodes match if nodes in their neighbourhood also match*":
  - if $c_{\mathsf{parent(t)}}$ matches a class c_x in n_Φ, then we establish c_x::c_t in n_Φ.
  - if no match is achieved with the parent, we apply the neighbourhood constraint procedure that matches $\mathcal{C}$ with $\mathcal{C}_{\mathsf{siblings(t)}}$; if the degree of matching the siblings exceeds a specified threshold, and the matched classes are found to be under a common parent in n_Φ, then we list c_t as a subclass of that superclass in n_Φ. If the alignment is successful, we add t to $\mathcal{T}$ and o to $\mathcal{O}$.

For instance, consider the ontologies A and B in Figure 4, $\mathcal{C}$ in A represents the classes of A, and $\mathcal{C}$ in B represents the classes of B. An example of when the new object type is found to be semantically equivalent to an existing type is when the newly received variable is `(be LG_ID6400 area_transit)`; the new object o is `LG_ID6400`, of type t `tv` $\notin \mathcal{T}$, the alignment finds that c_tv in B matches $c_\mathsf{television}$ in A and simply `LG_ID6400` is added to $\mathcal{O}$. An example of when the new object type is positioned using the neighbourhood constraint depending on the parent class is when the newly received variable is `(be bosch_ID3400 area_transit)`; the new object o is `bosch_ID3400`, of type t `dishwasher` $\notin \mathcal{T}$, the alignment finds that $c_{\mathsf{parent(dishwasher)}}$ is $c_\mathsf{electronic}$ in B and it matches $c_\mathsf{electronic}$ in A; therefore, the system asserts $c_\mathsf{electronic}$::$c_\mathsf{dishwasher}$ in A, creates a new entry `dishwasher - electronic` in $\mathcal{T}$, and adds `bosch_ID3400` is to $\mathcal{O}$. An example of when the new object type is positioned using neighbourhood constraint depending on siblings classes is when the newly received variable is `(be mirror_ID202 area_transit)`; the new object o is `mirror_ID202`, of type t `mirror` $\notin \mathcal{T}$, the alignment finds that $c_{\mathsf{parent(mirror)}}$ is c_other in B and it does not matches any class in A; however, c_bed, c_couch, and c_table (the siblings $\mathcal{C}_{\mathsf{siblings(mirror)}}$ in B) matches c_bed, c_sofa, and c_table, respectively in A, therefore, the system asserts $c_\mathsf{furniture}$::c_mirror in A, creates a new entry `mirror - furniture` in $\mathcal{T}$, and adds `mirror_ID202` is to $\mathcal{O}$.

Opportunity identification

Once the x candidate new goals are formulated as explained in Stage 4 of the overview of our approach, the system generates $\Phi' = \Phi'_1, \ldots, \Phi'_x$ (modified versions of Φ), where the added information includes o, t, the information of o, the discrepancy proposition, $\mathcal{G}' = g'_i \cup \mathcal{G}$, and the new current state $\mathcal{S}$. We use a planner to solve each $\Phi'_i \in \Phi'$ to know which g'_i can be considered an opportunity to Φ in the context of π. g'_i is considered an opportunity goal for Φ, when the planner is able to generate a plan π'_i to solve Φ'_i that includes the new goal plus the original set of goals. For the purpose of simulating the execution of the plan we reutilised the simulation system of the work in (Babli et al. 2016).

Cases of study

The aim of this section is to show the behaviour of our system with a representative example. We have tested the model with several application domains; tourism, underwater installations maintenance, driverlog, transport, and a repair agency.

For this paper we are going to consider the *repair agency* Φ, with a warehouse that has three areas a transit area, an inspection area, and a storage area, a robot `av` has the task of a one-day maintenance of several electronics and furniture items that are received by the agency, the robot is able to load one item at a time, finally the end location of the robot must be at the

```
• (:predicates
; a robot or an item is located at ?loc
(be ?locatable - (either robot electronics furniture)
?loc - location)
; controls when the robot is active
(active ?robot - robot)
; a robot is moving between locations
(moving ?robot - robot ?area1 ?area2 - location)
; an item requires repairing
(require_repair ?item - (either electronics furniture))
; an item has been repaired by a robot
(repaired ?item - (either electronics furniture))
; an item is loaded to a robot
(loaded ?item - (either electronics furniture) ?robot - robot)
; a robot is not carrying an item
(empty ?robot - robot)
; a repaired item is delivered to a storage area
(delivered ?item - (either electronics furniture)))

; move action between two locations
• (:durative-action move
:parameters(?robot - robot ?to - ?from - location)...)
; load item from a location into robot
(:durative-action load
:parameters(?robot - robot
?item - (either electronics furniture)
?location - location)...)
; unload item from robot to location
(:durative-action unload
:parameters(?robot - robot
?item - (either electronics furniture)
?location - location)...)
; repair item by a robot at an inspection area area
(:durative-action repair
:parameters(?robot - robot
?item - (either electronics furniture)
?inspection_area - inspection)...)
; dummy action
(:durative-action dummy
:parameters(?item - (either electronics furniture)
?storage_area - storage)...)

• (:goals (and
(repaired tv_ID101)
(repaired tv_ID02)
(repaired refrigerator_ID03)
(repaired sofa_ID04)
(delivered tv_ID101)
(delivered tv_ID02)
(delivered refrigerator_ID03)
(delivered sofa_ID04)
(be av area_storage)))
```

Figure 6: $\mathcal{V}$, $\mathcal{A}$, and $\mathcal{G}$ in *repair agency* Φ

storage area. Initially, the system has a set of electronics and furniture categories $\mathcal{T}$ as shown in Figure 3, On the other hand, $\mathcal{V}$, $\mathcal{A}$, and $\mathcal{G}$ are shown with comments in Figure 6, respectively. The simulator that we are using does not deal with conditional effects or derived predicates, therefore we added a dummy action with a zero duration that asserts that an item is delivered if it is repaired and it is at the storage area.

The information of $\mathcal{S}$ includes the three areas of the warehouse (`area_transit`), (`area_inspect`), and (`area_storage`) the robot start location (`be av area_storage`), the robot state (`empty av`), the robot operational hours between 10:00 and 23:00, the electronic and furniture items that require repairing and their locations, and the durations of movement between the warehouse areas, maintenance time, loading time, and unloading time).

The plan to solve Φ (PLAN1 is shown in Figure 7) is calculated by the planner and consists of 36 actions; the robot av moves from its start location (`area_storage`) to (`area_transit`), loads an item, moves to (`area_inspect`), unloads the item to be repaired, repairs the item, loads the item, moves to (`area_storage`), unloads the item, and the item is delivered; the previous rotation (8+1 dummy) applies for each of the four items that require repairing (shown in the goals in Figure 6); `tv_ID101`, `tv_ID02`, `refrigerator_ID03`, and `sofa_ID04`.

```
0.0003:   (MOVE AV AREA_STORAGE AREA_TRANSIT) [5.0000]
5.0005:   (LOAD AV REFRIGERATOR_ID03 AREA_TRANSIT) [1.0000]
6.0008:   (MOVE AV AREA_TRANSIT AREA_INSPECT) [5.0000]
11.0010:   (UNLOAD AV REFRIGERATOR_ID03 AREA_INSPECT) [1.0000]
12.0013:   (REPAIR AV REFRIGERATOR_ID03 AREA_INSPECT) [80.0000]
91.0015:   (LOAD AV REFRIGERATOR_ID03 AREA_INSPECT) [1.0000]
92.0017:   (MOVE AV AREA_INSPECT AREA_STORAGE) [5.0000]
97.0020:   (UNLOAD AV REFRIGERATOR_ID03 AREA_STORAGE) [1.0000]
• 98.0023:   (DUMMY REFRIGERATOR_ID03 AREA_STORAGE) [0.0000]
98.0025:   (MOVE AV AREA_STORAGE AREA_TRANSIT) [5.0000]
103.0027:   (LOAD AV SOFA_ID04 AREA_TRANSIT) [1.0000]
104.0030:   (MOVE AV AREA_TRANSIT AREA_INSPECT) [5.0000]
109.0033:   (UNLOAD AV SOFA_ID04 AREA_INSPECT) [1.0000]
110.0035:   (REPAIR AV SOFA_ID04 AREA_INSPECT) [80.0000]
189.0038:   (LOAD AV SOFA_ID04 AREA_INSPECT) [1.0000]
190.0040:   (MOVE AV AREA_INSPECT AREA_STORAGE) [5.0000]
195.0043:   (UNLOAD AV SOFA_ID04 AREA_STORAGE) [1.0000]
196.0045:   (DUMMY SOFA_ID04 AREA_STORAGE) [0.0000]
196.0047:   (MOVE AV AREA_STORAGE AREA_TRANSIT) [5.0000]
201.0050:   (LOAD AV TV_ID101 AREA_TRANSIT) [1.0000]
202.0052:   (MOVE AV AREA_TRANSIT AREA_INSPECT) [5.0000]
207.0055:   (UNLOAD AV TV_ID101 AREA_INSPECT) [1.0000]
208.0058:   (REPAIR AV TV_ID101 AREA_INSPECT) [80.0000]
287.0060:   (LOAD AV TV_ID101 AREA_INSPECT) [1.0000]
288.0063:   (MOVE AV AREA_INSPECT AREA_STORAGE) [5.0000]
293.0065:   (UNLOAD AV TV_ID101 AREA_STORAGE) [1.0000]
294.0067:   (DUMMY TV_ID101 AREA_STORAGE) [0.0000]
294.0070:   (MOVE AV AREA_STORAGE AREA_TRANSIT) [5.0000]
299.0073:   (LOAD AV TV_ID02 AREA_TRANSIT) [1.0000]
300.0075:   (MOVE AV AREA_TRANSIT AREA_INSPECT) [5.0000]
305.0078:   (UNLOAD AV TV_ID02 AREA_INSPECT) [1.0000]
306.0080:   (REPAIR AV TV_ID02 AREA_INSPECT) [80.0000]
385.0082:   (LOAD AV TV_ID02 AREA_INSPECT) [1.0000]
386.0085:   (MOVE AV AREA_INSPECT AREA_STORAGE) [5.0000]
391.0088:   (UNLOAD AV TV_ID02 AREA_STORAGE) [1.0000]
392.0090:   (DUMMY TV_ID02 AREA_STORAGE) [0.0000]
```

Figure 7: PLAN1

For the purpose of simulating the execution of the plan we reutilised the simulation system of our previ-

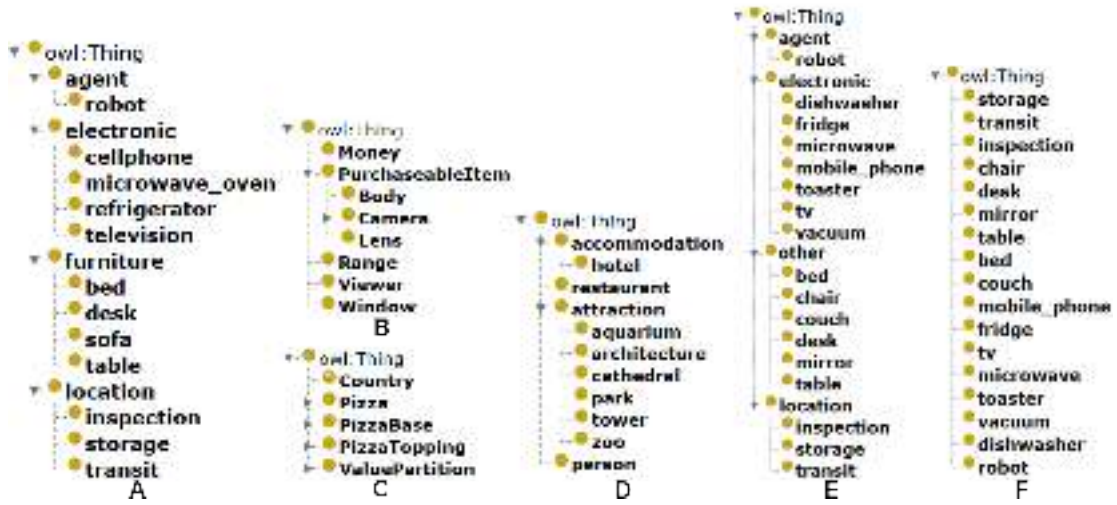


Figure 8: Several remote ontologies

ous work (Babli et al. 2016). The simulator creates a timeline (as shown in Figure 1) and starts PLAN1 execution simulation. Let us assume that after repairing and delivering the item `refrigerator_ID03` (after the ninth action in PLAN1, Figure 7), a new information is received from a different delivery agent `(be bosch_ID3400 area_transit)`, that includes a new object o `bosch_ID3400` $\notin \mathcal{O}$. The system requests the type of o and find t `dishwasher` $\notin \mathcal{T}$. The system creates n_Φ (shown in Figure 8 A), then it accesses several ontologies available in online-repositories $\mathsf{R'} = \{B, C, D, E, F\}$ (shown in Figure 8).

The ontologies are augmented using ConceptNet. VSM distance is calculated between A, and each ontology in $\mathsf{R'}$, the distances are respectively: 0.01, 0.01, 0.52, 0.87, and 0.81. $\mathsf{R'} = \{D, E, F\}$ is recognised as the set of most similar ontologies. The system tries to find `dishwasher` in $\mathsf{R'}$ and creates $\mathsf{R'_t} = \{E, F\}$, the set of ontologies out of $\mathsf{R'}$ that contain `dishwasher`. The SV distance is measured to find whether E or F describes more accurately the semantics of the application domain, the values are respectively 0.24 and 0.16; therefore, $n_t = E$. Next, the system attempts to position $c_{\mathsf{dishwasher}}$ in A by aligning A and E. The system finds that $c_{\mathsf{parent(dishwasher)}}$ in E is $c_{\mathsf{electronic}}$ and it matches the class $c_{\mathsf{electronic}}$ in A; therefore $c_{\mathsf{electronic}}::c_{\mathsf{dishwasher}}$ inside A A new entry `dishwasher - electronic` is added to $\mathcal{T}$, and `bosch_ID3400` is added to $\mathcal{O}$. The information required for integrating `bosch_ID3400` in Φ is automatically identified, requested, and added to $\mathcal{S}$. Since `dishwasher` is a sibling of a type that is involved in a goal $g \in \mathcal{G}$ thus, the system formulates new g'_1 = `(repaired bosch_ID3400)`, g'_2 = `(delivered bosch_ID3400)` and $\mathcal{G}' = g'_1 \cup g'_2 \cup \mathcal{G}$. Finally the system updates $\mathcal{S}$ with the current state at the time the new information was received. The planner is called to generate a new plan (PLAN2 shown in Figure 9); allowing the robot to repair and deliver the original set of items plus the new item. The simulation contin-

```
0.0003:   (MOVE AV AREA_STORAGE AREA_TRANSIT) [5.0000]
5.0005:   (LOAD AV SOFA_ID04 AREA_TRANSIT) [1.0000]
6.0008:   (MOVE AV AREA_TRANSIT AREA_INSPECT) [5.0000]
11.0010:   (UNLOAD AV SOFA_ID04 AREA_INSPECT) [1.0000]
12.0013:   (REPAIR AV SOFA_ID04 AREA_INSPECT) [80.0000]
91.0015:   (LOAD AV SOFA_ID04 AREA_INSPECT) [1.0000]
92.0017:   (MOVE AV AREA_INSPECT AREA_STORAGE) [5.0000]
97.0020:   (UNLOAD AV SOFA_ID04 AREA_STORAGE) [1.0000]
98.0023:   (DUMMY SOFA_ID04 AREA_STORAGE) [0.0000]
98.0025:   (MOVE AV AREA_STORAGE AREA_TRANSIT) [5.0000]
103.0027:   (LOAD AV TV_ID101 AREA_TRANSIT) [1.0000]
104.0030:   (MOVE AV AREA_TRANSIT AREA_INSPECT) [5.0000]
109.0033:   (UNLOAD AV TV_ID101 AREA_INSPECT) [1.0000]
110.0035:   (REPAIR AV TV_ID101 AREA_INSPECT) [80.0000]
189.0038:   (LOAD AV TV_ID101 AREA_INSPECT) [1.0000]
190.0040:   (MOVE AV AREA_INSPECT AREA_STORAGE) [5.0000]
195.0043:   (UNLOAD AV TV_ID101 AREA_STORAGE) [1.0000]
196.0045:   (DUMMY TV_ID101 AREA_STORAGE) [0.0000]
196.0047:   (MOVE AV AREA_STORAGE AREA_TRANSIT) [5.0000]
201.0050:   (LOAD AV BOSCH_3400 AREA_TRANSIT) [1.0000]
202.0052:   (MOVE AV AREA_TRANSIT AREA_INSPECT) [5.0000]
207.0055:   (UNLOAD AV BOSCH_3400 AREA_INSPECT) [1.0000]
208.0058:   (REPAIR AV BOSCH_3400 AREA_INSPECT) [80.0000]
287.0060:   (LOAD AV BOSCH_3400 AREA_INSPECT) [1.0000]
288.0063:   (MOVE AV AREA_INSPECT AREA_STORAGE) [5.0000]
293.0065:   (UNLOAD AV BOSCH_3400 AREA_STORAGE) [1.0000]
294.0067:   (DUMMY BOSCH_3400 AREA_STORAGE) [0.0000]
294.0070:   (MOVE AV AREA_STORAGE AREA_TRANSIT) [5.0000]
299.0073:   (LOAD AV TV_ID02 AREA_TRANSIT) [1.0000]
300.0075:   (MOVE AV AREA_TRANSIT AREA_INSPECT) [5.0000]
305.0078:   (UNLOAD AV TV_ID02 AREA_INSPECT) [1.0000]
• 306.0080:   (REPAIR AV TV_ID02 AREA_INSPECT) [80.0000]
385.0082:   (LOAD AV TV_ID02 AREA_INSPECT) [1.0000]
386.0085:   (MOVE AV AREA_INSPECT AREA_STORAGE) [5.0000]
391.0088:   (UNLOAD AV TV_ID02 AREA_STORAGE) [1.0000]
392.0090:   (DUMMY TV_ID02 AREA_STORAGE) [0.0000]
```

Figure 9: PLAN2

ues. After the robot has repaired `tv_ID02`, a new information is received `(be mirror_ID510 area_transit)`, that includes a new object o `mirror_ID510` $\notin \mathcal{O}$. The system requests the type of o and find t `Mirror` $\notin \mathcal{T}$. Similarly, and depending on the siblings using the neighbourhood constraint during the alignment, the system deals with the new information and extends the knowledge of the planning task, a new plan is obtained (PLAN3 shown in Figure 10), and the simulation continues. At the end of the day, the robot ends up in repairing six items instead of four.

```
0.0003:   (LOAD AV TV_ID02 AREA_INSPECT) [1.0000]
1.0005:   (MOVE AV AREA_INSPECT AREA_STORAGE) [5.0000]
6.0008:   (UNLOAD AV TV_ID02 AREA_STORAGE) [1.0000]
7.0033:   (DUMMY TV_ID02 AREA_STORAGE) [0.0000]
7.0010:   (MOVE AV AREA_STORAGE AREA_TRANSIT) [5.0000]
12.0013:   (LOAD AV MIRROR_ID510 AREA_TRANSIT) [1.0000]
13.0015:   (MOVE AV AREA_TRANSIT AREA_INSPECT) [5.0000]
18.0018:   (UNLOAD AV MIRROR_ID510 AREA_INSPECT) [1.0000]
19.0020:   (REPAIR AV MIRROR_ID510 AREA_INSPECT) [80.0000]
98.0023:   (LOAD AV MIRROR_ID510 AREA_INSPECT) [1.0000]
99.0025:   (MOVE AV AREA_INSPECT AREA_STORAGE) [5.0000]
104.0027:   (UNLOAD AV MIRROR_ID510 AREA_STORAGE) [1.0000]
105.0030:   (DUMMY MIRROR_ID510 AREA_STORAGE) [0.0000]
```

Figure 10: PLAN3

Conclusion

Context and context awareness are crucial for any intelligent agent that operates in a dynamic environment. To develop context-aware ambient intelligence planning service, suitable context models and reasoning approaches are necessary. In this paper we have presented a domain-independent approach that may be considered as a context model and a first step towards a context aware ambient intelligent planning service. Our approach bolsters an autonomous agent with the capability of extending its planning task to accommodate new information on the fly; to learn information about the planning task and to introduce relating information such as new objects whether of existing types or more importantly new types during the execution of the initial plan that solves the original planning task, that in turn may trigger the formulation of new goals and produce new plans to achieve the new goals in addition to the original set of goals. On the other hand, for future work we intend to focus on two aspects, the first aspect is related to goal directed behaviour, more specifically the goal reasoning approaches rather than goal generation, to endow the system with the ability to perform goal reasoning and management instead of delegating that task to a planner; and the second aspect is to investigate how integrating deep learning image recognition with the simulation system, ontologies, and planning would scale, and to check whether it is viable for a real time application.

References

Babli, M.; Ibáñez, J.; Sebastiá, L.; Garrido, A.; and Onaindia, E. 2016. An intelligent system for smart tourism simulation in a dynamic environment. In *2nd Workshop on Artificial Intelligence and Internet of Things*, 15–22.

Cashmore, M.; Fox, M.; D., L.; Magazzeni, D.; and Ridder, B. 2017. Opportunistic planning in autonomous underwater missions. *IEEE Transactions on Automation Science and Engineering* PP(99):1–12.

Cox, M. T. 2007. Perpetual self-aware cognitive agents. *AI Magazine* 28(1):32–46.

Dannenhauer, D., and Muñoz-Avila, H. 2013. Luigi: A goal-driven autonomy agent reasoning with ontologies. *Advances in Cognitive Systems* 2:1–18.

David, J., and Euzenat, J. 2008. Comparison between ontology distances (preliminary results). In *The 7th International Semantic Web Conference*, 245–260.

Doan, A.; Madhavan, J.; Dhamankar, R.; Domingos, P. M.; and Halevy, A. Y. 2003. Learning to match ontologies on the semantic web. *VLDB J.* 12(4):303–319.

Edelkamp, S., and Hoffmann, J. 2004. PDDL2.2: The language for the classical part of the 4th international planning competition. Technical report, 195, University of Freiburg.

Horridge, M., and Bechhofer, S. 2011. The OWL API: A java API for OWL ontologies. *Semantic Web* 2(1):11–21.

Klenk, M.; Molineaux, M.; and Aha, D. W. 2013. Goal-driven autonomy for responding to unexpected events in strategy simulations. *Computational Intelligence* 29(2):187–206.

Patel-Schneider, P. F.; Hayes, P.; and Horrocks, I. 2014. Owl web ontology language semantics and abstract syntax. World Wide Web Consortium.

Russell, S. J., and Norvig, P. 2010. *Artificial Intelligence - A Modern Approach (3. internat. ed.)*. Pearson Education.

Sánchez, D.; Batet, M.; Martínez, S.; and Domingo-Ferrer, J. 2015. Semantic variance: An intuitive measure for ontology accuracy evaluation. *Engineering Applications of Artificial Intelligence* 39:89–99.

Shvaiko, P.; Euzenat, J.; Srinivas, K.; Mao, M.; and Jiménez-Ruiz, E., eds. 2013. *Monolingual and Cross-lingual Ontology Matching with CIDER-CL: evaluation report for OAEI 2013*, volume 1111.

Speer, R.; Chin, J.; and Havasi, C. 2017. Conceptnet 5.5: An open multilingual graph of general knowledge. In *Proc. of the 31 Conference on Artificial Intelligence*, 4444–4451.

Vattam, S.; Klenk, M.; Molineaux, M.; and Aha, D. W. 2013. Breadth of approaches to goal reasoning: A research survey. Technical report, DTIC Document.

Discovering Numeric Constraints for Planning Domain Models

Alan Lindsay[1] and Peter Gregory[2]
[1]School of Computing and Engineering, University of Huddersfield, UK
[2]Digital Futures Institute, School of Computing, Teesside University, UK
a.lindsay@hud.ac.uk

Abstract

In this work we take a first look at domain model acquisition in planning domains with numeric constraints. We begin by constructing a typology of the numeric constraints represented in numeric domains of the third International Planning Competition (IPC). We then propose a numeric constraint representation that can capture several of these types. Our approach assumes that any condition is constructed by bounding a linear formula over the preceding ground actions. For example, driving a truck may rely on there being sufficient fuel. This can be modelled as a bound on the sum over the specific move actions that have been made in the plan. There are a large number of potential numeric constraints and as such we have proposed a heuristic layered approach to exploring the space. The implementation is still ongoing, but we present initial results for several forms of numeric constraint. This supports a discussion of the key issues surrounding our approach and the problem in general.

Introduction

Modelling is considered to be a bottleneck in the process of tackling combinatorial problems, due to the skills required to develop these models. Model generation is a crucial process within the planning community and modelling support tools have been developed to aid domain modellers, for example, the GIPO (Simpson, Kitchin, and McCluskey 2007), itSIMPLE (Vaquero et al. 2007) and KIWI (Wickler, Chrpa, and McCluskey 2014) systems. Another avenue of research to aid in the modelling process is based on learning models from example solutions: namely that of domain model acquisition, which is the core topic of this work.

Domain model acquisition has been applied across a range of research and application areas. For example within the business process community (Hoffmann, Weber, and Kraft 2012) and space applications (Frank et al. 2011). An extended version of the *LOCM* domain model acquisition system (Cresswell, McCluskey, and West 2009) has also been used to help in the development of a puzzle game (Ersen and Sariel 2015) based on spatio-temporal reasoning. Web Service Composition is another area in which domain model acquisition techniques have been used (Walsh and Littman 2008). However, there is relatively little work in the area of domain model acquisition that targets the numeric fragment of PDDL, which is perhaps surprising given that many commercial and industrial applications of automated planning technology rely on numeric state variables. For example, in constructing policies for the use of batteries (Fox, Long, and Magazzeni 2011), the construction of machine tool calibration plans (Parkinson et al. 2012) and spacecraft orbit planning (Surovik and Scheeres 2015). Within both board games and video games, numeric models are crucial in order to encode scoring systems, resource use, etc. Within interactive narrative settings, numeric variables represent varied structures, such as strength of relationships in social networks (Porteous, Charles, and Cavazza 2013; 2015) and the level of tension (Porteous et al. 2011) within a certain scene.

In recent work an approach for distributing plan cost amongst its contributing action costs has been presented (Gregory and Lindsay 2016). This has extended domain acquisition tools with the ability to identify the aspects of the planning model that are involved in accumulating cost during planning. However, numeric state variables are also used to capture numeric constraints, such as resource use, which play an important role in capturing an accurate model.

In this work we investigate how numeric constraints can be identified and used to extend the domain definition with the numeric variables that are necessary to represent them. The space of possible constraints is of course large and therefore we have exploited observations from benchmark models (those encoded for third IPC, in this case) combined with a layered heuristic approach, based on (Gregory and Lindsay 2016), in order to guide the exploration.

The paper is structured as follows: we present a background in domain model acquisition; we present a typology for number constraints of benchmark domains and develop a template for numeric constraints. We present a heuristic approach for exploring the space; finally we describe the current implementation, we present some initial results, outline the related and future work and conclude.

Background

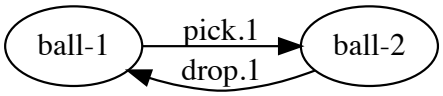
Domain model acquisition is the problem of learning a formal domain model of a system from some form of input data. The domain model acquisition system that we introduce in this paper assumes the *LOCM* family of algorithms and the generated structures that they create. In order to describe these algorithms, and also our own, we introduce

the Gripper domain as a running example. In this problem domain a robot moves between two rooms picking up and dropping off balls. The robot is typically constrained in how many balls can be carried. Although this constraint was originally encoded using a collection of gripper objects, it can naturally be encoded using a numeric constraint.

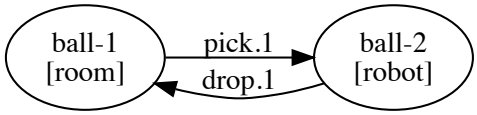
The Family of *LOCM* Algorithms

LOCM (Cresswell, McCluskey, and West 2009) is a system for learning domain models from example action sequences. Its distinguishing feature is that it uses no other information besides the action sequences, i.e., no information about types, predicates, initial or final states. This is possible because it is based on restricting assumptions about the form of the domain model.

The domain model construction has two aspects: Firstly, the action sequences are analysed to obtain a simple state machine for each type. The sequences of transitions experienced by individual objects are tracked through example plan sequences and the objects that are observed performing the same transitions are grouped into types. A finite state machine is generated for each type, which captures the transitions for that type. E.g., for a ball in Gripper, the following machine is revealed, comprising two transitions and two states.



Secondly, the action sequences are further analysed to establish whether a given state for a given type has a temporary association with another object. If so, the state is qualified with a parameter which records the association. For the Gripper balls, they have an association with a room in one state, and a robot in the other state.



It is then possible for the learned model to be translated into the STRIPS fragment of PDDL. Each state is represented by a PDDL predicate having its associated object as first argument, with further arguments formed from state parameters. Operators are constructed from the transitions and their parameters, using the binding constraints discovered between action parameters and state parameters.

The *LOCM2* system (Cresswell and Gregory 2011) generalises the approach and allows types to be represented by multiple state machines, each containing a subset of the full transition set for the type. The *LOP* system (Gregory and Cresswell 2015) learns static relations by comparing optimal input plans with the optimal plans found using the induced domain model of *LOCM2*. Assuming that *LOCM2* has detected the dynamics of the problem correctly, then if the induced plan is shorter, then this provides evidence to support the hypothesis that some static relation has gone undetected.

NLOCM (Gregory and Lindsay 2016) is a domain model acquisition system for domains with action costs. As well as example plans, the system also requires the final plan costs as input. The approach then creates a hypothesis of how the cost was accumulated from action costs and how these costs can be attributed to specific sets of parameters. The parameter sets are defined as *templates*, which each represent a possible parameter combination for an operator that might have a value associated with it.

Definition 1. A *template*, T_O, for an operator, O, from the set of all the operators, $\mathcal{O}$, is defined as:

$$T_O \subseteq \mathbb{P}(\text{args}(O)) \qquad (O \in \mathcal{O})$$

Similarly, we define $T_A = T_O$, where A is an instantiating action of operator O.

For example, the template $\{2,3\}$ for a move action, e.g., (move _ $?r_0$ $?r_1$), defines a collection of features for each combination of rooms that can be traversed between, e.g., (move _ room1 room2). Here the symbol '_' indicates a wildcard in the template that matches any instantiation. *NLOCM* both identifies a selection of templates that must be active to explain the cost and identifies the costs that each feature contributes to the plan costs (such as the cost for moving between room1 and room2).

In this work we aim to identify the specific language features in a domain that cause numeric effects and the conditions that act on those effects. We therefore will use templates as a means of indicating a feature that contributes to a numeric variable, and we also use them as constraint templates, which capture the important parameters for the constraint.

Numeric Constraints

The space of possible numeric constraints is very large and in this section we identify a restricted subset that still allows us to explore many numeric constraints. To start the section, we present an illustrative example of the use of a numeric constraint in the Gripper domain. In order to best choose an appropriate subset, we have surveyed the domains from the third IPC and developed a typology of the numeric constraints represented. After discussing this we present the representation that defines the subset of numeric constraints that we have focussed on in this work.

A Numeric Constraint in the Gripper Domain

As an example, consider a general two room Gripper domain, where a robot transfers balls between two rooms. It is common that there is a restriction on the number of balls that can be held by the robot. One way of encoding this is to use a numeric variable and maintain the count through careful manipulation in the effects of actions. For example, the drop action is extended with the following effect:

(increase (held-balls-count ?r) (-1))

The pick action is then constrained by exploiting the value maintained in this variable, through the precondition:

(> (upper-problem-limit) (+ (held-balls-count ?r) 1))

Capacity limit	Notes
Satellite	limited data capacity
Zeno-travel	max capacity for zooming
Rover	data capacity to store photo
Settlers	limited capacity for resources
UMT2	weight and volume constraints
Fuel	Notes
Satellite	slew time
Zeno-travel	zoom and fly
Rover	charge of rover
Settlers	e.g., coal for train
Static	Notes
UMT2	dimension constraints

Table 1: A typology of numeric constraints for the third IPC domain models.

Typology of Numeric Constraints

The numeric constraints from domains of the third IPC (Long and Fox 2003) can be broadly grouped into capacity limits and fuel constraints. Table 1 categorises the domains: Rovers, Satellite, Settlers, UMT and Zeno-travel, by the types of numeric constraint that they contain. Although there are numeric versions of Driverlog, the numeric variables are only used for accumulating cost and not as part of any constraint.

Capacity constraints are either constraints that limit an object, o's, accumulation of a certain relationship (e.g., a robot can only pick up 2 balls), or they are constraints that limit what o can do once it has been loaded (e.g., the robot can move and carry 1 ball, but not 2 balls). A variable is typically used to monitor the current load of the object. The value added, v, when the relationship is made between o and some other object o' is usually a function of o', such as weight (or simply a counter, as in the Gripper example). The object, o, is associated with a capacity and any action that would lead to the variable having a larger value than the capacity is not permitted. When the relationship is broken then the variable is decremented by v.

Fuel constraints represent the depletion of finite resources and limit the number of certain transitions that occur in the plan (at least without refuelling). For example, a truck may only be able to carry n units of fuel, limiting its transitions around a map. This sort of constraint typically acts on a transition where an object, o, transfers a relationship with an object, o' for the same type of relationship with an alternative object, o''. The amount of resource used by the action will commonly depend on a property of the pair (o', o''). For example, the amount of fuel consumed by a truck traversing between two locations might depend on the distance of the objects. In Zeno-travel the amount of fuel also depends on the mode of travel used, e.g., the zoom action uses more fuel than the fly action.

We have also included static numeric constraints, such as height, or width constraints, which can be represented in *LOP* as static facts and are not explored further.

Plan steps	v_{r_0}	v_{r_1}
s_0	0	0
a_1: (**pick** b_1 $\mathbf{r_0}$ l_1)		
s_1	**1**	0
a_2: (**pick** b_2 $\mathbf{r_1}$ l_2)		
s_2	1	**1**
a_3: (move r_0 l_2 l_1)		
s_3	1	1
a_4: (**drop** b_2 $\mathbf{r_1}$ l_1)		
s_4	1	**0**

Table 2: An illustration of two numeric state variables being maintained during planning.

A Model of Numeric Constraints

In this work we focus on a formalism for numeric constraints that allows us to capture an interesting subset of the constraints from the third IPC. We assume that all of the constraints can be represented as a bound on a (possibly parameterised) numeric variable, such as ($<$ (held-balls-count r_0) 2). Furthermore, we assume that for each numeric constraint, a collection of action costs can be defined and that the value of the numeric variable is the sum of the costs for the applied actions.

In the Gripper example, the held-balls-count variable can be calculated using the operator costs: 1 for pick operators, and -1 for drop operators. Notice that in the case of a parameterised constraint a connection is needed so that a specific numeric variable is only effected by the relevant action costs. E.g., Table 2 demonstrates how these values can be maintained, using v_{r_0} for held-balls-count for robot r_0 and v_{r_1} for r_1. This will be made more precise in the following section.

At this stage it is helpful to consider the value of the variable in terms of counts of contributing terms (i.e., as a linear function). For some plan fragment we can count the number of times a certain feature matches in the plan. A cost can then be associated with each feature. For example, in the Gripper example, we can say that:

(held-balls-count ?r) = $1 \times (pick\_?r\_) + -1 \times (drop\_?r\_)$

These features therefore match to pick and drop plan steps that are applied to the relevant robot. This value can then be used in order to constrain the valid actions by selecting a bound that separates the valid values, where an action should be applicable, from invalid values. An example of a constraint for the pick operator in Gripper is: ($>$ (upper-problem-limit) (+ (held-balls-count ?r) 1)),[1] where held-balls-count are a set of numeric variables that maintains a count of the number of balls that are being held for each robot. The numeric constant upper-problem-limit captures the upper bound of the function and defines how many balls can be held. Of course in general the cost and upper bound may be parameterised. The valuation is maintained

[1]Notice that the upper bound is set assuming that the numeric effect of the current action has been applied. This is very typical and is therefore assumed in our model.

Constraint Template	pick ?r _
Variables	$v_{?r} = v_{r_0}, \ldots$
View Options	pick:?r↦drop:?r pick:?r↦pick:?r pick:?r↦move:?r
Action Costs	C(pick _ _)=1 C(drop _ _)=-1 C(move _ _)=0

Figure 1: The constraint template: (pick ?r _), indicates that the pick operator is constrained and that the constraint is defined in terms of individual robots. As such a set of variables: $v_{r_0}, \ldots$, maintain the current valuation for each robot, r_i. The view options make a mapping between the robot variable of the pick operator to variables of the other operators. There is only 1 such mapping for robots in this domain. Finally a set of action costs are defined (only operator costs in this example). E.g., each time a drop action is applied then the relevant numeric variable is determined through the parameter mapping and the cost (-1) is added.

depending on those matching instantiations of the action parameters.

This model allows us to explore capacity constraints and some fuel type constraints (only a limited form of refuelling can be captured and considering richer representations has been left for future work).

Selecting Numeric Constraints

In this section we describe our approach for exploring the space of possible numeric constraints. We begin by defining the input data that our approach relies on and then define the space of numeric constraints that we explore in this work. We define the metric that we use to select between explanations and define an ordered exploration of increasingly more complex explanations. The final part presents the simple loop that we use in order to incrementally build a collection of numeric constraints from the presented approach that only identifies single constraints.

The Space of Numeric Constraints

Our representation of numeric constraints has four elements: a constraint template and their associated numeric variables, the upper bounds for the variables, the costs associated with each *feature* (an instantiated template used for cost) and a mapping between the constraint template and the features. Figure 1 presents an example of the main components of a numeric constraint in the Gripper domain. The constraint template, denoted $\tau^{\mathbf{NC}}$, determines the operator that is constrained and the parameters of the action that are involved. In the Gripper example, the constraint template, $\tau^{\mathbf{NC}}$ is pick$\{1\}$, which is defined for the robot parameter of the pick action.

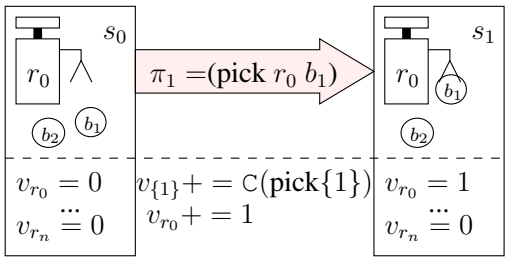


Figure 2: An illustration of the maintenance of a variable for each tuple of the control template (in this case: pick$\{1\}$). Each variable, v_{r_i}, records the number of balls that the robot r_i is holding. The view option links the first variable of the pick action to the parameter of the numeric variable. The evaluation of $\mathtt{C}(\text{pick}\{1\}) == 1$ increments the variable to maintain a count of the number of balls being held.

An important aspect of the approach is in defining a supporting numeric variable, which is effected by the relevant operators so that it maintains the valuation of the desired function over the preceding plan fragment. This variable is then compared to the accepted upper bound value for the function. An example of a constraint for the pick operator in Gripper is: ($<$ (held-balls-count ?r) (upper-problem-limit)), where held-balls-count are a set of numeric variables that each maintain a count of the number of balls that are being held for each robot. The numeric constant upper-problem-limit captures the upper bound of the function and in this case, it is the same for all robots. This constraint is illustrated for upper-problem-limit ==1 in Figure 3.

The templates and their sets of features are similar to those in (Gregory and Lindsay 2016) and they capture the values associated with the active ground templates. For example, in the Gripper domain, the following feature values might be defined: C(pick _ _ _)=1 and C(drop _ _ _)=-1. If the pick feature is active then one is added to the relevant variable each time a pick action is applied (see Figure 2). In the case of the drop feature, then 1 is removed each time a drop action is applied. In a more complex domain, the robot might be limited by the total weight of the balls that it could carry and features parameterised by the ball could be used to encode the ball's weight, e.g., C(pick b1 _ _)=30.

In order to connect the templates used to accumulate cost, τ, with a constraint template, $\tau^{\mathbf{NC}}$, we define the set of view options: $\mathbb{VO}_{\tau^{\mathbf{NC}},\tau}$, which define all mappings between the two templates (ignoring incompatible types). This dictates the parts of the plan step that should match before the plan step is deemed relevant for $\tau^{\mathbf{NC}}$. If a variable is expected to maintain a count for a specific object then it should only match with plan actions that describe this object in certain transitions. For example, consider a constraint on the number of entrances to a room and the number of balls dropped in the room. In order to correctly accumulate the correct features requires careful matching of the parameters between the variable used to maintain the function's evaluation and the relevant feature values.

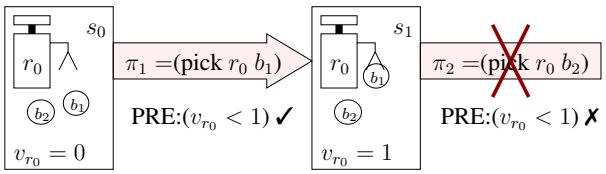


Figure 3: An illustration of how the control template (in this case: pick{1}) uses the maintained variables (e.g., v_{r_0}) in order to constrain the applicability of actions. The robot can only pick up one ball at a time (i.e., a precondition of $v_{r_0} < 1$) and therefore the action π_2 is not applicable.

For a particular constraint, we can calculate the evaluation of the function at each plan step. This is computed as the linear function of the active and relevant feature counts, which we denote, $\mathcal{C}_\pi(i, \tau)$, multiplied by their associated value, $\mathcal{W}_{\pi_i,\tau}$:

$$\mathcal{F}_\pi(i) = \sum_\tau \mathcal{C}_\pi(i, \tau) \times \mathcal{W}_{\pi_i,\tau}$$

If we assume a finite number of possible values for $\mathcal{W}_{\pi_i,\tau}$ then the set of all possible constraints can be enumerated. This is a large set and some of these constraints will be meaningless and others less desirable. The rest of this section describes how we have selected useful constraints and organised our exploration of the space.

Input Data Our approach relies on a set of positive example plans, Π^+ and a set of negative example plans, Π^-. In this work we assume that the plans are each generated for the same initial state, which allows us to determine the appropriate bound for a constraint using a set of example plans, rather than just one example plan. Each plan in the negative examples, $\pi^- \in \Pi^-$, provides a valid action sequence, up to the last step. In fact, for each negative example, $a_0, ..., a_n = \pi^-$, we add the plan fragment $a_0, ..., a_{n-1}$ to the positive examples. This can potentially lead to a more accurate constraint by not underestimating the bound.

Useful Numeric Constraints The selection of useful constraints relies on Π^+ and Π^-, the sets of positive and negative examples. The positive examples are used in order to establish an upper bound for the variable (that is the valuation of $\mathcal{F}_\pi(i)$, for plan step i, which is highest amongst all relevant plan steps in the plans of Π^+) during valid transitions. [2] By its definition, all positive examples must be explained by any generated constraint.

A negative example is covered by a feature costing when the final plan index matches the condition-template and the constraint variable is higher than the upper bound. As each plan in the negative examples minus its final action is added to the positive examples, there is no opportunity for the sequence being regarded as a negative example before its end.

[2] It interesting to note that the variable may in fact become larger than this upper bound. The upper bound captures the highest value that is observed at a plan step that satisfies the condition-template.

Optimisation Function

We use a tiered evaluation function to first promote the functional generality of the explanation and then a cascade of tie-breaking, preferring less complex explanations:

1. Maximise: # covered negative examples
2. Minimise: sum of active template costs
3. Minimise: # of features used
4. Minimise: sum over feature weight magnitudes

The first tier selects the explanations that maximise the number of negative plans whose evaluation is higher than the upper bound calculated for the positive examples. The first tie-breaking tier breaks ties on the complexity of the active templates required in the explanation. This follows (Gregory and Lindsay 2016) and is based on the arity of the template. We then select explanations with lower numbers of non-zero feature costs in the explanation. The final tier promotes explanations that are expressed with smaller magnitudes.

Layered Exploration of the Space of Constraints

The space of all constraints that we have defined is large. In order to reduce the size of the learning task, we propose to shape the exploration through this space by iteratively extending the complexity of the constraints that we consider. In order to support a more organised exploration of the space, we have focussed on feature costs at the operator level. For example, this allows allocating a cost of 1 to move actions and not distinguishing between possible differences in cost between different robots. We are also so far focusing on constraint templates that have a single parameter.

Given the large search space, it is prudent to exploit any additional structure where it can be of benefit. One specific way this is done is by exploiting typical properties of constraint formulation in the context of the transition system inferred by LOCM. Structures, generated as part of the LOCM domain inference process, have been exploited in the context of learning functions of accumulating action costs (Gregory and Lindsay 2016). In that work, the use of state parameters provides insight into useful subsets of the parameters that could be relevant for allocating action costs. However, monotonic accumulation is not as common when describing numerical constraints and instead there is commonly a balance of increasing and decreasing effects as resources are claimed and released. In this context we can exploit different structures that are generated by LOCM. In particular, in several layers we will only allow certain transitions from an object's FSM (see the background section). The layers are described here:

1. Operator based constraint templates. E.g., appropriate for capturing capacity constraints in the Gripper domain when there is only 1 robot.
2. Single object focussed constraint templates and a focussed subset of the object's FSM features. At this layer a constraint template can be learned for a single object (e.g., a robot) and only transitions that change that object's FSM state, or the FSM state parameters can be part of the constraint. For example, a parameter (the room) is changed

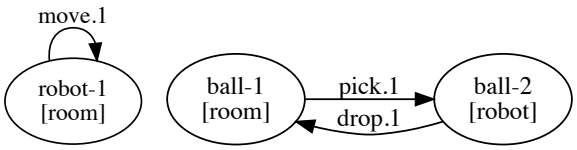


Figure 4: The reduced FSMs for Gripper, illustrating only the transitions that change the state of the objects.

Algorithm 1 GROWCONSTRAINT: a method for incrementally growing a numeric constraint.

```
1:  procedure GROWCONSTRAINT(𝕃, Π⁺, Π⁻)
2:      constraint=[]
3:      for layer_i ∈ 𝕃 do
4:          repeat
5:              notchanged=true
6:              exp = findBestExplanation(layer_i, Π⁺, Π⁻)
7:              if len(exp.covered) > 0 then
8:                  notchanged=false
9:                  constraint.add(exp)
10:                 Π⁻.removeAll(exp.covered)
11:             end if
12:         until notchanged
13:     end for
14:     return removeSubsumedExplanations(constraint)
15: end procedure
```

when a robot is moved and therefore the move 1 template can be allocated cost. However, dropping a ball does not effect the robot's state and therefore cannot be allocated cost.[3] The reduced systems are presented in Figure 4 for the Gripper domain.

3. Collation of objects that have shared parameter. At this layer we acknowledge that many constraints act across sets of objects that each share a relationship with a single object. For example, a robot exists as a parameter in the space of each ball that is held by the robot. The active features at this layer are those that either add, remove or maintain the parameter in each of the objects that can achieve the parameter.
4. Single object focussed constraint templates and FSM features. At this layer all of the object's FSM features are active and can be used in the definition of a constraint.

We have implemented layers 1,2 and 4 and have used these in the evaluation. Although layer 3 is more complex than layer 4, we predict it will have fewer instantiations.

Incrementally Growing the Constraint

In this work we have adopted a simple approach in order to construct a set of numeric constraints that cover as many of the negative examples as possible. The pseudo-code is presented in Algorithm 1. The inputs are a list of layers ($\mathbb{L}$) and the positive and negative training examples (Π^+ and Π^-). The procedure loops through the layers through increasing complexity. At each layer a best explanation for the examples is created. While the proposed explanations can still cover negative examples (the explanation returned, exp has a list of the negative examples that it covers) then the procedure remains at the same layer, removing the covered examples after each new constraint is added. Once no more constraints can be found at a layer then the procedure continues to the next layer.

A final step is used to clean out some overfitting constraints made in the less expressive layers. We start with the constraints created for the more expressive layers and then we only add those constraints from previous layers that explain examples that have not been explained by the constraints in the layers above. This approach removes some occurrences where partial explanations are generated at layers that cannot explain the constraint completely.

Initial Evaluation

An initial prototype was developed that has allowed us to test the proposed approach on the layers 1,2 and 4 of the layered strategy presented above. At each tier all of the possible constraints are enumerated and the best constraint selected, using the presented scoring function. The covered negative examples are removed and the process repeated until no more examples can be explained, where the next tier is attempted. For this evaluation we have constrained all features to have costs within certain small bounds.

We have tested the approach on simple versions of the Zeno-travel domain and a transportation domain. The transportation domain involves redistributing packages using trucks and captures a capacity constraint, as well as constraint on the number of exits from a location (i.e., the Mystery domain encoding of fuel). In Zeno-travel, airplanes transport passengers between cities, traversing by either flying or zooming. The domain defines constraints on flying and zooming in terms of fuel and also only allows zooming when there are a small number of occupants. The former constraint distinguishes between the amount of fuel used by each of the operators.

Two sets of examples: positive and negative, were generated for each domain. The negative examples were generated by performing a 20 step random walk, $a_0, ..., a_{19}$, in the partial model (the model without the numeric constraints). Each step of the walk was then simulated using the complete model. In this way the index of the first inapplicable action, a_i, was identified and the fragment, $a_0, ..., a_i$ was then used as a negative example. This acts in place of an action sequence validator. Positive examples are random walks generated using the complete model. 50 positive and 50 negative examples were generated for each domain.

Table 3 presents counts of both possible combinations and the number of explanations generated at each layer of complexity. The table makes it clear how important it is to exploit the leverage of structures that are more likely to be used for capturing numeric constraints. In particular, consider the different number of combinations generated in the case of layers 2 and layers 4.

[3] The intention of layer 2 is that the object's that are impacted by a change should record its cost, which we expect will help reduce redundancy. For example, in the case of dropping a ball, it is more natural that this cost relates to the ball and not the robot. Indeed this is often the case, e.g., weight or size.

Domain	Range	COMB(L1)	EXP(L1)	COMB(L2)	EXP(L2)	COMB(L4)	EXP(L4)	Ok
Trans	-1,..,1	162	9	42	1	846	23	✓
Trans	-3,..,3	2058	147	210	3	21462	847	✓
Trans	-5,..,5	7986	605	506	5	125598	5472	✓
Zeno	-1,..,1	324	0	144	9	42444	1045	✗
Zeno	-3,..,3	9604	0	588	47	715596	5917	✓
Zeno	-5,..,5	58564	0	1452	123	10688172	96150	✓

Table 3: The number of generated combinations, COMB(L) and explanations, EXP(L), for the three layers explored and different feature domains (Range). The Ok column indicates that the constraints matched those of the domain.

Transportation

In the first layer an explanation is generated that constrains the total number of move actions (in each move action 1 is added to a numeric variable with no parameters). The upper bound for this constraint sums the number of times all locations can be exited. In the second layer an explanation is generated that constrains the number of a move actions that a specific truck can perform. The bound given to this constraint is lower than the total number of possible moves and therefore this constraint overfits the training data. However, these constraints are both structurally valid and when correctly parameterised with a problem specific upper bound they capture part of the fuel constraint.

In the fourth layer the system identified two constraints that covered the negative examples. The first captured the constraint on the number of times a location has been left. It was represented by defining a numeric variable for each location, which counts (+1) each time a move action is applied with the location in the second parameter. The costs were as follows: `cost`(move.3)=0, `cost`(drop.3)=0, `cost`(move.2)=1 and `cost`(pickup.3)=0. This constraint subsumes the constraints learned in the previous layers and the system therefore removes them from the constraint and retains this constraint.

The second learned constraint at this level captures the capacity constraint on the truck as a condition on the pickup action. Each time a truck loads a package, a variable associated to the truck is incremented and each time a package is unloaded it is decremented, maintaining a count of the number of packages on the truck. The cost allocations were as follows: `cost`(pickup.2)=1, `cost`(drop.2)=-1 and `cost`(move.1)=0.

Zeno-Travel

The approach correctly identified the constraints for the simplified Zeno-travel domain. No constraints were found at the first layer, but two were found at the second layer. These constraints are similar: each captures the fuel constraint, one for the zoom action and the other captures the constraint for fly. The active features for airplane objects (at the second layer) are zoom.1 and fly.1. In each case the approach allocates feature values of `cost`(zoom.1)=2 and `cost`(fly.1)=1, correctly identifying that the zoom action uses more fuel. The constraint identified for the zoom action is presented in Figure 5. A numeric state variable (in this case zoom-bound-accum) is created in order to maintain

(**:action** `zoom`
:parameters (?a1 - airplane ?c1 ?c2 - city)
:precondition (> (zoom-bound ?a1) (+ (zoom-bound-accum) 2) ..
:effect ... (increase (zoom-bound-accum) **2**)..)

(**:action** `fly`
:parameters (?a1 - airplane ?c1 ?c2 - city)
:precondition ...
:effect ...(increase (zoom-bound-accum) **1**)..)

Figure 5: PDDL fragments presenting the constraint identified for the zoom action, which captures the restriction on fuel. The system was able to identify the increased fuel consumption used by the zoom action.

the valuation of the function at each state. The learnt constraint is then represented using a precondition, which relies on the maintained variable. This illustrates the potential for redundancy in the approach. Each constraint is described in isolation and we do not currently exploit the possibility of sharing variables for several constraints.

In the fourth layer all of the transitions defined in the airplane FSM are available, which are: zoom.1, fly.1, debark.2 and board.2. The system learns a constraint for the zoom action, which correctly captures that the zoom action is only applicable when there are up to a maximum number on-board (i.e., with respect to a specific airplane). The values are as follows: `cost`(zoom.1)=0, `cost`(fly.1)=0, `cost`(debark.2)=-1 and `cost`(board.2)=1. This constraint is interesting because the operator with the precondition (zoom) is not involved in changing the numeric variable.

For the range {-1,..1} the system found an alternative explanation for the input data. This is because the actual constraint was not expressible given the range.

Future Work

There are several key limitations of the presented work that must be examined in order to better understand the nature of this problem. Numeric state variables are used to encode distances, resource limits and other arbitrary values. However, exploring features with large domains within a large space of possible template choices is impractical and will lead to overfitting. One possible connection that could be exploited is between the feature's contribution to a constraint and its contribution towards the plan cost. For example, driving between A and B may use x units of fuel, but also contribute x

to the total pollution of the plan. In this way the feature costs discovered by *NLOCM* may also provide some indication of feature values for constructing constraints.

We have assumed in this work that the part of the model that can be represented by static relations is correctly discovered by *LOP*. However, the identification of static relations in *LOP* is based on the assumption that it can explain the missing constraints. We suspect that it may infer spurious constraints when faced with input data that it cannot correctly explain. In future work we will both investigate this suspicion and if it is correct then investigate learning the constraints together.

It is common that the current fuel level cannot be captured in a linear sum over the preceding consumption and refuelling actions. The extension of the model with a single reset (or refuel) feature, which has the effect of setting the variable equal to zero would be a possible extension.

Perhaps the most important consideration is the input data that we have used. In this work, the negative examples indicate the actions that contain numeric constraints in their preconditions, however, they do not provide much information regarding the underlying cause for the failure or the objects and actions that have contributed. Currently example sets must also all share a common initial state.The problem is that if different initial states are used then different initial values for the numeric state variable would be possible and identifying a bound for it would be meaningless. It will be important future work to examine alternative data sets in order to best consider the trade-off between the quality of the output and the effort of defining the input data.

Conclusion

In this work we have investigated the problem of identifying numeric constraints in planning models. Our approach attempts to identify functions that separate a given set of good and bad plan fragments. The functions are defined in terms of a linear sum over the preceding steps in the plan fragment. In our evaluation we show that it is possible to identify common numeric constraints, such as resource restrictions. However, there is a very large space of possible numeric constraints, which this work only begins to explore. In the future work we have outlined some of the more important open questions in this area.

References

Cresswell, S., and Gregory, P. 2011. Generalised Domain Model Acquisition from Action Traces. In *International Conference on Automated Planning and Scheduling*, 42 – 49.

Cresswell, S.; McCluskey, T. L.; and West, M. M. 2009. Acquisition of object-centred domain models from planning examples. In Gerevini, A.; Howe, A. E.; Cesta, A.; and Refanidis, I., eds., *International Conference on Automated Planning and Scheduling*. AAAI.

Ersen, M., and Sariel, S. 2015. Learning behaviors of and interactions among objects through spatio-temporal reasoning. *Computational Intelligence and AI in Games, IEEE Transactions on* 7(1):75–87.

Fox, M.; Long, D.; and Magazzeni, D. 2011. Automatic construction of efficient multiple battery usage policies. In *International Conference on Automated Planning and Scheduling*, 74–81.

Frank, J. D.; Clement, B. J.; Chachere, J. M.; Smith, T. B.; and Swanson, K. J. 2011. The Challenge of Configuring Model-Based Space Mission Planners. In *International Workshop on Planning and Scheduling for Space*.

Gregory, P., and Cresswell, S. 2015. Domain Model Acquisition in the Presence of Static Relations in the LOP System. In *International Conference on Automated Planning and Scheduling*, 97–105.

Gregory, P., and Lindsay, A. 2016. Domain Model Acquisition in Domains with Action Costs. In *Proc. of the 26th Int. Conf. on Automated Planning and Scheduling (ICAPS)*.

Hoffmann, J.; Weber, I.; and Kraft, F. M. 2012. SAP speaks PDDL: Exploiting a software-engineering model for planning in business process management. *Journal of Artificial Intelligence Research* 44:587–632.

Long, D., and Fox, M. 2003. The 3rd international planning competition: Results and analysis. *Journal of Artificial Intelligence Research* 20:1–59.

Parkinson, S.; Longstaff, A.; Crampton, A.; and Gregory, P. 2012. The Application of Automated Planning to Machine Tool Calibration. In *International Conference on Automated Planning and Scheduling*.

Porteous, J.; Teutenberg, J.; Pizzi, D.; and Cavazza, M. 2011. Visual programming of plan dynamics using constraints and landmarks. In *International Conference on Automated Planning and Scheduling*, 186–193.

Porteous, J.; Charles, F.; and Cavazza, M. 2013. NetworkING: using character relationships for interactive narrative generation. In *International Conference on Autonomous Agents and Multi-agent Systems*, 595–602.

Porteous, J.; Charles, F.; and Cavazza, M. 2015. Using Social Relationships to Control Narrative Generation. In *AAAI*, 4311–4312.

Simpson, R. M.; Kitchin, D. E.; and McCluskey, T. L. 2007. Planning domain definition using GIPO. *Knowledge Eng. Review* 22(2):117–134.

Surovik, D. A., and Scheeres, D. J. 2015. Heuristic Search and Receding-Horizon Planning in Complex Spacecraft Orbit Domains. In *International Conference on Automated Planning and Scheduling*, 291–295.

Vaquero, T. S.; Romero, V.; Tonidandel, F.; and Silva, J. R. 2007. itsimple 2.0: An integrated tool for designing planning domains. In *International Conference on Automated Planning and Scheduling*, 336–343.

Walsh, T. J., and Littman, M. L. 2008. Efficient Learning of Action Schemas and Web-Service Descriptions. In *AAAI*, 714 – 719.

Wickler, G.; Chrpa, L.; and McCluskey, T. L. 2014. KEWI - A knowledge engineering tool for modelling AI planning tasks. In *International Conference on Knowledge Engineering and Ontology Development*, 36–47.

Modelling Sequences of Processes in PDDL+ for Efficient Problem Solving

Elad Denenberg and **Amanda Coles**
King's College London
Bush House, 30 Aldwych,
London, United Kingdom WC2B 4BG

Abstract

For many years, the planning community has adopted the mantra 'physics not advice'; indicating that a planning model should describe the properties of the problem, but should not give advice on how to solve the problem. This is in contrast with other, more widely applied, search and optimization technologies, such as Mixed-Integer Programming and Constraint Programming as well as AI technologies, such as Evolutionary Algorithms and Machine learning. As models get more complex, solving planning problems becomes more challenging, and it becomes important for us to understand how to model problems so that they can be solved efficiently by planners; in order to be able to apply planning in real life applications. In this paper we focus on a particular common pattern: the need for several effects to be applied sequentially as the result of a single decision. This may occur, for example, when an action starts a cascade of effects. In this paper, we consider different ways of modelling this in PDDL, and compare the efficiency of solving the problem using each of these models in three state-of-the-art PDDL+ planners: SMT-Plan+, OPTIC and DiNo. Our results show that the more intuitive model is less scalable on the first two, and that a model ensuring fewer happenings is less scalable on the third. By presenting this work we hope to encourage more research in developing efficient planning models for expressive domains in order to allow planning to be applied in a wider range of applications.

Introduction

Many in the Domain Independent Planning community subscribe to the philosophy 'physics not advice'; that is, since we are dealing with the design of planners that should cope with any domain, when generating a testbed the domains must not contain hints or assist the planner in solving a given problem. In contrast, in many other fields of AI and optimization the realization that no tool can perform well on all domains (the No Free Lunch Theorem (NFL) (Wolpert and Macready 1997)) has inspired much work to be carried out in the area of selecting the proper search tool for a given model, or modelling in a way that would facilitate faster search with a given a tool. This area has received relatively little attention in planning, and in particular we do not yet have a good understanding of how modelling decisions affect the performance of the most expressive planning systems: PDDL+ planners.

In this work we will discuss a simple example of matching an expressive PDDL+ model to a search tool in planning problems. The example we discuss is modelling a process sequence: a number of processes which start one after the other. Processes in PDDL+, as defined by (Fox and Long 2002), can be described as effects that act upon a variable regardless of actions taken, as long as a set of predicates is true. An example of a process is gravity acting on a falling object, or a battery being charged by solar energy as the sun rises. A process sequence may appear naturally in a domain, however, one may also encounter such sequence when modelling a series of effects.

Often in real life engineering domains one may come across a series of continuous numeric effects acting one after another upon a variable. This may happen due to an action starting a cascade of such effects. For instance a rover transmitting data back to base. The transmission itself has several phases: linking, transmitting, awaiting confirmation, receiving confirmation data. All these phases have different continuous effects on battery usage, and therefore can be represented as a cascade of continuous numeric effects on the battery charge level. This may also apply to interval constraints (Tran et al. 2017) and (Tierney et al. 2012).

Another reason for effect series may be a result of piecewise linearization of a non-linear domain (Denenberg and Coles 2018; Cao et al. 2011). For instance, power landing using rockets: A robot is landing and is under the force of gravity, at any given point it may fire its engines to reduce the velocity of the fall. The engine firing changes the velocity exponentially while the gravity changes the velocity polynomially. Mixing these functions may prove to be hard, and therefore the user might chose to approximate each of the non-linear effects as a sequence of piecewise linear continuous effects.

This paper discusses three different methods for modelling sequences of continuous effects. One uses only featuers of PDDL 2.1, clips and durative actions. Next, two PDDL+ models are examined: The first of these two is an intuitive one which might be created by an end user. The second model is less intuitive, but ensures a smaller number of happenings. We compare the perofrmance of these models empirically using three different state-of-the-art PDDL+

planners: SMTPlan+ (Cashmore et al. 2016), OPTIC (Benton, Coles, and Coles 2012) and DiNo (Piotrowski et al. 2016). We present results indicating which is the most efficient model to use for each type of planner, and an analysis of why each model is better suited to that type of planner.

Problem Definition

A PDDL+ planning problem (Fox and Long 2002) is a tuple $\langle$F,v,A,P,I,G$\rangle$[1] where:

- F is a set of propositions (facts);
- v is a set of real numeric variables;
- A is a set of actions;
- P is a set of processes;
- I ($\subseteq F$) is the initial state;
- G (a conjunction of facts from F and numeric conditions over v) is the goal.

Each action has three sets of preconditions which must hold at the start of, at the end of, and throughout its execution respectively. Preconditions are conjunctions of propositions (or their negations) and numeric conditions (which for our purposes we will assume can be represented in the form **w.v**$\{>, \geq, <, \leq, =\}$c where (w is a vector of constants and c is a constant)). Actions can have instantaneous effects at their start or end, these can be propositions that are added or deleted, or updates to numeric variables of the form v$\{$+=, -=, =$\}$**w.v** + c where v $\in$ v. In addition to this, actions can have continuous numeric effects that happen throughout their duration, most generally of the form dv/dt $\{$+=, -=, =$\}$**w.v** + c; but in this work we focus on linear continuous change where continuous effects are of the form dv/dt $\{$+=, -=, =$\}$ c. Finally, actions have a duration constraint, defining a permissible range from which the planner can select the duration of the action.

Processes comprise a precondition and a set of continuous numeric effects. They differ from actions in the semantics of their execution: nominally actions model the activities the planner can choose to take; whereas processes model exogenous happenings in the environment. If the preconditions of an action are true in a given state, then the planner can *choose* to apply that action in that state (or not to). In contrast, if the preconditions of a process are true in a given state then that process will execute automatically, the planner has no choice over this.

A solution to this problem is a *plan*: a sequence of actions from A that transforms I into G.

Often in literature PDDL+ models are referred to as *hybrid*, meaning they mix continuous and instantaneous effects. This work does not explore the hybrid property of PDDL+, rather, it aims at making use of the added expressiveness of processes.

[1]We exclude events from our definition as we do not use them in this work

Running Example: Generator Domain

Throughout the remainder of the paper we use the well-known generator domain as a running example. The objects in this domain are a generator, which has a main tank with a given capacity. Generating electricity consumes fuel from the main tank. The generator can be refuelled using an auxiliary tank.

During the plan the Generator is required to work without "choking", that is, the level of fuel in the main tank must not reach zero. When refuelling, the fuel must not spill, i.e. the level of the fuel must not be greater than the capacity of the main tank.

The consumption of fuel by the generator is assumed to be linear. And thus modelled by a single action with a continuous linear effect. Refuelling from an auxilliary tank is approximated by three piecewise linear sections as described in Figure 1. These are the linear effects that become the effects of individual actions or processes that must be sequencesd to model the effect throughout the whole duration of refuelling.

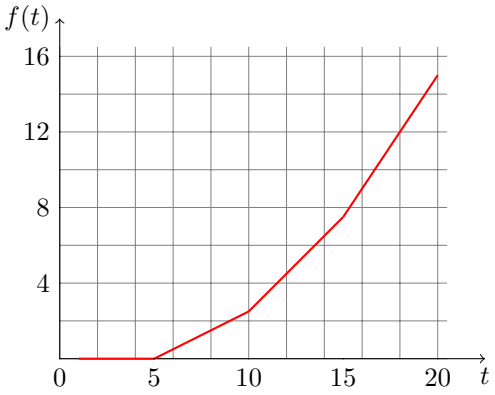


Figure 1: Linearized refuel

Modelling Sequences of Continuous Effects

We now consider the problem of modelling a sequence of conditional effects that occur as a result of a single decision. More formally, we have a physical effect E that acts continuously upon the variable v, and has an effect $f(u)$ that is defined such:

$$\frac{\mathrm{d}v}{\mathrm{dt}} = f(u) = \begin{cases} a_1 & 0 < u \leq u_1 \\ a_2 & u_1 < u \leq u_2 \\ \vdots & \\ a_n & u_{n-1} < u \leq u_n \end{cases} \quad (1)$$

where a_i are a set of contributions to the rate of v, and u_j are a set of values of u. These contributions could be constant, modelling a piecewise linear function (as those we consider in this paper) or could themselves be functions over problem variables. We present 3 different approaches to modelling this type of effect in PDDL.

Durative Action and Clip Model

It is possible to model sequences of continuous effects, in PDDL 2.1, without the need to invoke PDDL+ processes. In order to do this we make use of clips (Fox, Long, and Halsey 2004). Clips are additional actions that are used to bind together the execution of actions in order to ensure one starts as soon as another finishes.

Figure 2 illustrates this compilation: each action A_i has effect `(increase v (* #t) a_i)`. Additionally A_i ($i > 1$) has start and end preconditions that are added at the start, and deleted at the end of, C_i and C_{i+1} respectively. Finally, the start of A_{i+1} ($i < n$) adds an end precondition of ci. In this way, we can force A_{i+1} to happen immediately after Ai ends. In our generator example, A_1,A_2 and A_3 would be actions, with appropriate durations, that when sequenced model refuelling. Each has a continuous effect on *fuel* corresponding to the respective slope in Figure 1. Propositional preconditions and effects would enforce that the actions must be applied in sequence, in this order.

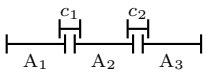


Figure 2: Clip and Durative Action Model

This representation has the advantage that it does not require a PDDL+ capable planner; although it still exhibits a level of expressiveness captuerd by few planners. It is worth noting that this representation requires a large number of actions to be added to the domain, over which the planner must search.

Modelling as Process Sequences

An alternative is to use a set of PDDL+ processes to describe the effects, as demonstrated in (Denenberg and Coles 2018).

The most intuitive way to model such a series of effects using a sequence of processes would be to define a set of processes P_i, each defined on the interval $u_{i-1} < u \leq u_i$, as shown in Figure 3 and visualized in Figure 4a. In our generator example, the refuel action would be a single action that sets a counter u to zero and has effect `(increase u (* #t) 1)`, with a start add/end delete effect of some fact $refuelling$ denoting that refuelling is happening. The processes, with precondition f and their respective u value ranges would then be responsible for updating the fuel level only during their respective ranges.[2]

It was discovered that many planners struggle with processes defined in this fashion. Therefore, we suggest the "stacking" of the processes. That is, instead of defining the process on an interval, we defining a single condition and incorporating the effect of process P_i to the effect of P_{i+1} in the following manner:

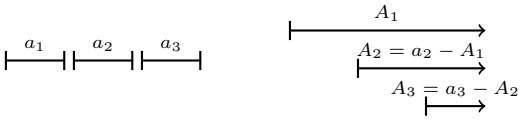
[2]Note that this model is correct for the case where refuelling with the same tank cannot self-overlap.

```
(:process Pi
 :parameters ()
 :precondition (and
        (< u u_{i-1})
        (>= u u_i) )
 :effect ( increase v (* #t a_i) )
)
```

Figure 3: Intuitive Process

$$\frac{\mathrm{d}v}{\mathrm{d}t} = F(u) = \begin{cases} A_1 & 0 < u \leq u_1 \\ A_2 & u_1 < u \\ \vdots & \\ A_n & u_{n-1} < u \\ A_{n+1} & u_n < u \end{cases} \quad (2)$$

Where $A_1 = a_1$ and $A_i = a_i - A_{i-1}$ for all $i > 1$. Only a starting condition is defined for each process PF_i. The effect of the process PF_i incorporates the new change a_i as well as the removal of the previous effect A_{i-1}. This is visualized in Figure 4b and demonstrated in Figure 5. Again, in our generator example we create a refuel action assigning u to zero at the start, and with effect `(increase u (* #t) 1)`, the processes now start in sequence, but overlap, all continuing until the end of the refuel action.



(a) Intuitive Model (b) "Stacking" Model

Figure 4: Visualization of Processes Models

```
(:process PFi
 :parameters ()
 :precondition (< u u_{i-1})
 :effect ( increase v (* #t A_i) )
)

(:process PFiplusone
 :parameters (?b - bar)
 :precondition (< u u_{i})
 :effect ( increase v (* #t A_{i+1}) )
)
```

Figure 5: Intuitive Process

Evaluation

In this section we examine two linearised versions of the well known generator domain. The first, a symmetric domain, where there is no importance as to which tank is to be used when. The second an asymmetric domain, in which the

use of tank number one must precede that of number two and so on. Using the PDDL 2.1 and the two PDDL+ models to describe these versions resulted in six domains: Symmetric and asymmetric Clips, symmetric and asymmetric intuitive model domains and symmetric and asymmetric "stacking" model domains.

On each domain four problems were tested: a generator with one,two, three and four auxiliary tanks. The problem files for all domains and problems are given in the Appendix.

The problems were tested on three state of the art planners: a PDDL+ implementation of OPTIC, SMTPlan+ and DiNo.

The Clips models proved to be inefficient: SMTPlan+ and DiNo timed out on all problems, both symmetric and asymmetric. Only Optic was able to find a valid solution, and the results are presented in Table 1 and Table 2.

The average results of the tests of the PDDL+ symmetrical domain are summarized in Table 1, and the asymmetrical in Table 2.

The results of the symmetric PDDL+ domains are as follows: In both OPTIC, SMTPlan+ the process "stacking" model performed better than the intuitive model. Using the process "stacking" model in the one tank problem proved to be 85% faster than the intuitive model. Using SMTPlan+ the fast model was 98% faster in the two tank problem, while OPTIC was unable to solve the intuitive model within 1000 seconds and timed out. STMPlan+ was able to solve the 3 tank problem within 769 secons.

DiNo was able to solve both PDDL+ models and scale well. The difference between the intuitive and "stacking model" was relatively small. Even though the process "stacking" model ensures fewer happenings, DiNo scaled slightly better while using the intuitive model.

number of tanks			1	2	3	4
OPTIC+	Clips and Struts		0.07	14.16	TO	TO
	Processes	Intuitive	22.86	TO	TO	TO
		Stacking	2.95	922.32	TO	TO
		%diff	12.90	-	-	-
SMTPlan+		Intuitive	0.22	132.39	TO	TO
		Stacking	0.03	0.72	769.33	TO
		%diff	14.09	0.55	-	-
DiNo		Intuitive	3.02	4.04	5.52	7.43
		Stacking	2.78	4.07	5.84	7.64
		%diff	92.05	100.74	105.80	102.83

Table 1: Symmetric Generator Problem Test Results

number of tanks			1	2	3	4
OPTIC+	Clips and Struts		0.07	14.06	TO	TO
	Processes	Intuitive	22.70	TO	TO	TO
		Stacking	2.89	29.51	38.78	49.86
		%diff	12.72	-	-	-
SMTPlan+		Intuitive	TO	TO	TO	TO
		Stacking	0.59	TO	TO	TO
DiNo		Intuitive	2.57	3.58	Failed	Failed
		Stacking	2.58	Failed	Failed	Failed

Table 2: Asymmetric Generator Problem Test Results

The results of the asymmetric PDDL+ domains are: OPTIC showed great improvement when using the "stacking" model, SMTPlan+ operated better on the "stacking" model, and was able to solve the porblem with one tank, though it timed out on the rest of the problems. DiNo performed better on the intuitive model. In addition, DiNo failed to run on any of the "stacking" domains.

Discussion

The problems presented in the previous section implies that the non-intuitive model is more efficient on two planners while less efficient on the third regardless of whether the domain is symmetric. In this section we will propose an explanation for the observed performance.

OPTIC solves a planning problem by transforming durative actions (and processes), to snap actions. The search in OPTIC is over happenings, which are either the application of snap actions, or the decision to start or stop a process. If a process has a precondition that is trivially false (for example, a proposition (e.g. *refuelling*) is known to be false in the state) then there is no need to make a search decision about whether that process executes or not. However, when a precondition is over a continuously changing variable (e.g. $(< u\ u_i)$ where u is currently subject to continuous numeric change), then the planner must make a search decision, whether to start or stop a process conditioning on it. The plan for the intuitive model requires more happenings (start refuel, start p_1, end p_1,start p_2, end p_2,start p_3, end p_3, end refuel) all at different times. For the stacked model, however, the plan: start refuel, start P_1, start P_2, start P_3, end refuel, with only 5 happenings, suffices as the planner can deduce that as soon as it ends refuel all the processes must end P_1,P_2 and P_3 immediately, as one of their preconditions *refuelling* has been deleted. This means that the solution plan appears at a lower depth in the search tree.

A similar thing happens with the SMTPlan+. This planner uses a defined number of happenings, for each happening a set of Satisfiability Problem (SAT) equations is formulated and solved. Because fewer happenings are required to solve the stacked model (the processes can end at the same happening as the refuel action), SMTPlan+ can solve the problem with fewer happenings. Since SMTPlan+ by default starts with a small number of happenings and increases this until it finds a solution; finding a solution with fewer happenings means fewer SAT problems have to be proven unsolvable to find a solution.

The DiNo planner generates states by discretization. The states generated end up being comprised of a large amount of happenings ϵ time units from one another. at each such happening the planner may chose to perform an instantaneous action, start or end a durative action or process, or update the variables using all the currently active effects. Therefore the performance of this planner depends on the number of time units the plan requires, and the user defined size of the ϵ, rather than by the number of happenings required by the domain. Furthermore, since the planner needs to update the variables every ϵ time step, the more effects acting on a variable the more calculations would be required. This is why the "stacking" model scales slightly worse in this planner.

Conclusions

This paper presented three models for each of two variants for the same problem, and the performance of three planners on these models. The PDDL2.1 model proved to be inefficient on all three planners. Two of the planners showed high sensitivity to the type of PDDL+ model, and the third, showed slightly better scalability with one domain over the other. Had this been a real life problem, the engineer solving it would have been required to either chose the planning tools to fit his problem, or to model the problem in a manner that would allow the problem to be solved within reasonable time. This is a clear demonstration of the need to research modelling in planning. In addition to emphasizing the need to study the matching of models to planners, this work also introduces a means of modelling a cascade of effects.

Modelling a cascade of effects in PDDL+ as a sequence of processes was explored: Two models proposed were tested on three planners in symmetrical and asymmetrical representation. Future work must include additional PDDL+ capable planners. In addition, the domains used here were linear, in the sense the effects were of linear change. Since the contribution of the model was in reducing the number of happenings, we do not expect a non-linear effect to show significantly different behaviour scalability wise. However, future tests will include planners performance on such non-linear domains.

Since the use of processes on intervals are slow in planners that depend on the amount of happenings, it might be useful to create a preprocessing tool for these planners, such that would parse a domain and identify the intuitive, yet less efficient model, and replace it with the "stacking" model suggested in this paper.

References

Benton, J.; Coles, A. J.; and Coles, A. 2012. Temporal planning with preferences and time-dependent continuous costs. In *ICAPS*, volume 77, 78.

Cao, J.; Bell, K. R. W.; Coles, A. J.; and Coles, A. I. 2011. Voltage control of distribution network using an artificial intelligence planning method. In *Proceedings of the Twenty First International Conference and Exhibition on Electricity Distribution (CIRED)*.

Cashmore, M.; Fox, M.; Long, D.; and Magazzeni, D. 2016. A compilation of the full pddl+ language into smt. In *AAAI Workshop: Planning for Hybrid Systems*.

Denenberg, E., and Coles, A. 2018. Automated planning in non-linear domains for aerospace applications. In *58th Israel Annual Conference on Aerospace Sciences*.

Fox, M., and Long, D. 2002. Pddl+: Modeling continuous time dependent effects. In *Proceedings of the 3rd International NASA Workshop on Planning and Scheduling for Space*, volume 4, 34.

Fox, M.; Long, D.; and Halsey, K. 2004. An investigation into the expressive power of pddl2. 1. In *ECAI*, volume 16, 338.

Piotrowski, W.; Fox, M.; Long, D.; Magazzeni, D.; and Mercorio, F. 2016. *Heuristic planning for PDDL+ domains*, volume 2016-January. International Joint Conferences on Artificial Intelligence. 3213–3219.

Tierney, K.; Coles, A. J.; Coles, A.; Kroer, C.; Britt, A. M.; and Jensen, R. M. 2012. Automated planning for liner shipping fleet repositioning. In *ICAPS*, 279–287.

Tran, T. T.; Vaquero, T.; Nejat, G.; and Beck, J. C. 2017. Robots in retirement homes: Applying off-the-shelf planning and scheduling to a team of assistive robots. *Journal of Artificial Intelligence Research* 58:523–590.

Wolpert, D. H., and Macready, W. G. 1997. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation* 1(1):67–82.

Acknowledgments

This work was supported by the UK Engineering and Physical Sciences Research Council (EPSRC) grant EP/P008410/1 (AI Planning with Continuous Non-Linear Change).Elad Denenberg was supported in part by the Joan and Reginald Coleman-Cohen Fund.

Appendix

```
(define (problem run-generator2)
(:domain generator2)
(:objects gen - generator tank1 tank2 tank3 -
    tank)
(:init
;(= (fuelLevel gen)  990) ; 1 tank
;(= (fuelLevel gen)  970) ; 2 tanks
(= (fuelLevel gen)  955) ; 3 tanks
;(= (fuelLevel gen) 940) ; 4 tanks
(= (capacity gen)  1000)
(available tank1)
(= (reftime tank1) 0)
(available tank2)
(= (reftime tank2) 0)
(available tank3)
(= (reftime tank3) 0) )
(:goal (generator-ran))
)
```

Figure 6: Symmetric Problems

```
(define (domain generator2)
(:requirements  :fluents  :durative-actions
    :duration-inequalities  :adl  :typing)
(:types generator tank)
(:predicates (refueling ?g - generator) (
    generator-ran) (available ?t - tank))
(:functions (fuelLevel ?g - generator) (capacity
    ?g - generator) (reftime ?t - tank) )

(:durative-action generate
:parameters (?g - generator)
:duration (= ?duration  1000)
:condition (over all (>= (fuelLevel ?g) 0))
:effect (and ( decrease (fuelLevel ?g) (* #t 1))
        (at  end (generator-ran)) ) )

(:durative-action refuel
:parameters (?g - generator ?t - tank)
:duration  (= ?duration  20)
:condition (and ;(at start (not (refueling ?g)) )
        (at  start (available ?t))
(over all (< (fuelLevel ?g) (capacity ?g))))
:effect (and (at  start (refueling ?g))
        (in     rease (reftime ?t) (* #t 1))
        (at  start (not (available ?t)))
        (at  end (not (refueling ?g))) ) )

(:process refueling1
```

```
:parameters (?g - generator ?t - tank)
:precondition (and (refueling ?g)
        (>= (reftime ?t) 5)
        (< (reftime ?t)  10) )
:effect ( increase (fuelLevel ?g) (* #t 0.5)) )

(:process refueling2
:parameters (?g - generator ?t - tank)
:precondition (and (refueling ?g)
        (>= (reftime ?t) 10)
        (< (reftime ?t)  15) )
:effect ( increase (fuelLevel ?g) (* #t 1.0)) )

(:process refueling3
:parameters (?g - generator ?t - tank)
:precondition (and (refueling ?g)
        (>= (reftime ?t) 15) )
:effect ( increase (fuelLevel ?g) (* #t 1.5)) ) )
```

Figure 7: Symmetric Intuitive Model Domain

```
(define (domain generator2)
(:requirements  :fluents  :durative-actions
    :duration-inequalities  :adl  :typing)
(:types generator tank)
(:predicates (refueling ?g - generator) (
    generator-ran) (available ?t - tank))
(:functions (fuelLevel ?g - generator) (capacity
    ?g - generator) (reftime ?t - tank) )

(:durative-action generate
:parameters (?g - generator)
:duration (= ?duration  1000)
:condition (over all (>= (fuelLevel ?g) 0))
:effect (and ( decrease (fuelLevel ?g) (* #t 1))
        (at  end (generator-ran)) ) )

(:durative-action refuel
:parameters (?g - generator ?t - tank)
:duration  (= ?duration  20)
:condition (and ;(at start (not (refueling ?g)) )
        (at  start (available ?t))
        (over all (< (fuelLevel ?g) (capacity ?g)
            )))
:effect (and (at  start (refueling ?g))
        ( increase (reftime ?t) (* #t 1))
        (at  start (not (available ?t)))
        (at  end (not (refueling ?g))) ) )

(:process refueling1
:parameters (?g - generator ?t - tank)
:precondition (and (refueling ?g)
        (>= (reftime ?t) 5)
        ;(< (reftime ?t)  10)
        )
:effect ( increase (fuelLevel ?g) (* #t 0.5)) )

(:process refueling2
:parameters (?g - generator ?t - tank)
:precondition (and (refueling ?g)
        (>= (reftime ?t) 10)
        ;(< (reftime ?t)  15)
        )
:effect (
    increase (fuelLevel ?g) (* #t 0.5));1.0))
)

(:process refueling3
:parameters (?g - generator ?t - tank)
:precondition (and (refueling ?g)
        (>= (reftime ?t) 15)
:effect (
    increase (fuelLevel ?g) (* #t 0.5));1.5))
) )
```

Figure 8: Symmetric Process "Stacking" Model Domain

```
(define (domain generator2)
(:requirements  :fluents  :durative-actions
    :duration-inequalities  :adl  :typing)
(:types generator tank)
(:predicates (refueling ?g - generator) (
    generator-ran) (available ?t - tank) (s0 ?t -
     tank) (s1 ?t - tank) (s2 ?t - tank) (s3 ?t -
     tank) (c01 ?t - tank) (c12 ?t - tank) (c23 ?
    t - tank))
(:functions (fuelLevel ?g - generator) (capacity
    ?g - generator) (reftime ?t - tank) )

(:durative-action generate
:parameters (?g - generator)
:duration (= ?duration  1000)
:condition (over all (>= (fuelLevel ?g) 0))
:effect (and ( decrease (fuelLevel ?g) (* #t 1))
(at  end (generator-ran)) ) )

(:durative-action refuel
:parameters (?g - generator ?t - tank)
:duration  (= ?duration  20)
:condition (and (at  start (available ?t))
(over all (< (fuelLevel ?g) (capacity ?g)))
(at  end (s3 ?t)) )
:effect (and (at  start (refueling ?g))
(at  start (not (available ?t)))
(at  start (s0 ?t))
(at  end (not (refueling ?g))) ) )

(:durative-action strt-refueling1
:parameters (?g - generator ?t - tank)
:duration (= ?duration 5)
:condition (and (at  start (c01 ?t))
(at  end (c12 ?t)) )
:effect (and (
    increase (fuelLevel ?g) (* #t 0.5))
(at  start (s1 ?t))
(at  end (not(s1 ?t))) ) )

(:durative-action strt-refueling2
:parameters (?g - generator ?t - tank)
:duration (= ?duration 5)
:condition (and  (at  start (c12 ?t))
(at  end (c23 ?t)) )
:effect (and ( increase (fuelLevel ?g) (* #t 1))
(at  start (s2 ?t))
(at  end (not (s2 ?t))) ) )

(:durative-action strt-refueling3
:parameters (?g - generator ?t - tank)
:duration (= ?duration 5)
:condition (and (at  start (c23 ?t)) )
:effect (and (
    increase (fuelLevel ?g) (* #t 1.5))
(at  start (s3 ?t)) ) )

(:durative-action clp01
:parameters (?g - generator ?t - tank)
:duration (= ?duration 0.15)
:condition (and (at  start (s0 ?t))
(at  end (s1 ?t)) )
:effect (and (at  start (c01 ?t))
(at  end (not (c01 ?t))) ) )

(:durative-action clp12
:parameters (?g - generator ?t - tank)
:duration (= ?duration 0.15)
:condition (and (at  start (s1 ?t))
(at  end (s2 ?t)) )
:effect (and (at  start (c12 ?t))
(at  end (not (c12 ?t))) ) )

(:durative-action clp23
:parameters (?g - generator ?t - tank)
:duration (= ?duration 0.15)
:condition (and (at  start (s2 ?t))
(at  end (s3 ?t)) )
:effect (and  (at  start (c23 ?t))
(at  end (not (c23 ?t))) ) ) )
```

Figure 9: Symmetric Clips Domain

```
(define (problem run-generator2)
(:domain generator2)
(:objects gen - generator tank1 tank2 tank3 -
    tank) ;
;(:objects gen - generator tank1 tank2 - tank)
;(:objects gen - generator tank1 - tank)
(:init
;(= (fuelLevel gen)  990) ; 1 tank
;(= (fuelLevel gen)  970) ; 2 tanks
(= (fuelLevel gen)  955) ; 3 tanks
;(= (fuelLevel gen) 940) ; 4 tanks
(= (capacity gen)  1000)
(= (last-used gen) 0)
(available tank1)
(= (reftime tank1) 0)
(= (tanknum tank1) 1)
(available tank2)
(= (reftime tank2) 0)
(= (tanknum tank2) 2)
(available tank3)
(= (reftime tank3) 0)
(= (tanknum tank3) 3)
)
(:goal (generator-ran))
)
```

Figure 10: Asymmetric Problems

```
(define (domain generator2)
(:requirements  :fluents  :durative-actions
    :duration-inequalities  :adl  :typing)
(:types generator tank)
(:predicates (refueling ?g - generator) (
    generator-ran) (available ?t - tank))
(:functions (fuelLevel ?g - generator) (capacity
    ?g - generator) (reftime ?t - tank) (
    last-used ?g - generator) (tanknum ?t - tank)
     )

(:durative-action generate
:parameters (?g - generator)
:duration (= ?duration  1000)
:condition (over all (>= (fuelLevel ?g) 0))
:effect (and ( decrease (fuelLevel ?g) (* #t 1))
        (at  end (generator-ran)) ) )

(:durative-action refuel
:parameters (?g - generator ?t - tank)
:duration  (= ?duration  20)
:condition (and (at
    start (= (last-used ?g) (- (tanknum ?t) 1) ))
        (at  start (available ?t))
        (over all (< (fuelLevel ?g) (capacity ?g)
            )))
:effect (and (at  start (refueling ?g))
        ( increase (reftime ?t) (* #t 1))
        (at  start (not (available ?t)))
        (at      end (not (refueling ?g)))
        (at  end (assign (last-used ?g) (tanknum
            ?t))) ) )

(:process refueling1
:parameters (?g - generator ?t - tank)
:precondition (and (refueling ?g)
        (>= (reftime ?t) 5)
        (< (reftime ?t)  10) )
:effect ( increase (fuelLevel ?g) (* #t 0.5)) )

(:process refueling2
:parameters (?g - generator ?t - tank)
:precondition (and (refueling ?g)
        (>= (reftime ?t) 10)
        (< (reftime ?t)  15) )
:effect ( increase (fuelLevel ?g) (* #t 1.0)) )

(:process refueling3
:parameters (?g - generator ?t - tank)
:precondition (and (refueling ?g)
        (>= (reftime ?t) 15)
:effect ( increase (fuelLevel ?g) (* #t 1.5)) ) )
```

Figure 11: Asymmetric Intuitive Model Domain

```
(define (domain generator2)
(:requirements  :fluents  :durative-actions
    :duration-inequalities  :adl  :typing)
(:types generator tank)
(:predicates (refueling ?g - generator) (
    generator-ran) (available ?t - tank))
(:functions (fuelLevel ?g - generator) (capacity
    ?g - generator) (reftime ?t - tank) (
    last-used ?g - generator) (tanknum ?t - tank)
     )

(:durative-action generate
:parameters (?g - generator)
:duration (= ?duration  1000)
:condition (over all (>= (fuelLevel ?g) 0))
:effect (and ( decrease (fuelLevel ?g) (* #t 1))
        (at  end (generator-ran)) ) )

(:durative-action refuel
:parameters (?g - generator ?t - tank)
:duration  (= ?duration  20)
:condition (and ;(at start (not (refueling ?g)) )
        (at  start (= (last-used ?g) (- (tanknum
            ?t) 1) ))
        (at  start (available ?t))
        (over all (< (fuelLevel ?g) (capacity ?g)
            )))
:effect (and (at  start (refueling ?g))
( increase (reftime ?t) (* #t 1))
        (at  start (not (available ?t)))
        (at  end (not (refueling ?g)))
        (at  end (assign (last-used ?g) (tanknum
            ?t))) ) )

(:process refueling1
:parameters (?g - generator ?t - tank)
:precondition (and (refueling ?g)
        (>= (reftime ?t) 5)
        ;(< (reftime ?t)  10)
        )
:effect ( increase (fuelLevel ?g) (* #t 0.5)) )

(:process refueling2
:parameters (?g - generator ?t - tank)
:precondition (and (refueling ?g)
        (>= (reftime ?t) 10)
        ;(< (reftime ?t)  15)
        )
:effect (
    increase (fuelLevel ?g) (* #t 0.5));1.0))
)

(:process refueling3
:parameters (?g - generator ?t - tank)
:precondition (and (refueling ?g)
(>= (reftime ?t) 15) )
:effect (
    increase (fuelLevel ?g) (* #t 0.5));1.5))
) )
```

Figure 12: Asymmetric Process “Stacking” Model Domain

```
(define (domain generator2)
(:requirements  :fluents  :durative-actions
    :duration-inequalities  :adl  :typing)
(:types generator tank)
(:predicates (refueling ?g - generator) (
    generator-ran) (available ?t - tank)
(s0 ?t - tank) (s1 ?t - tank) (s2 ?t - tank) (s3
    ?t - tank) (c01 ?t - tank)
(c12 ?t - tank) (c23 ?t - tank) )
(:functions (fuelLevel ?g - generator) (capacity
    ?g - generator)
(last-used ?g - generator) (tanknum ?t - tank))
```

```
(:durative-action generate
:parameters (?g - generator)
:duration (= ?duration  1000)
:condition (over all (>= (fuelLevel ?g) 0))
:effect (and ( decrease (fuelLevel ?g) (* #t 1))
        (at  end (generator-ran)) ) )

(:durative-action refuel
:parameters (?g - generator ?t - tank)
:duration  (= ?duration  20)
:condition (and
(at  start (available ?t))
        (at  start (= (last-used ?g) (- (tanknum
            ?t) 1) ))
        (over all (< (fuelLevel ?g) (capacity ?g)
            ))
        (at  end (s3 ?t)) )
:effect (and (at  start (refueling ?g))
        (at  start (not (available ?t)))
        (at  start (s0 ?t))
        (at  end (not (refueling ?g)))
        (at  end (assign (last-used ?g) (tanknum
            ?t))) ) )

(:durative-action strt-refueling1
:parameters (?g - generator ?t - tank)
:duration (= ?duration 5)
:condition (and (at  start (c01 ?t))
        (at  end (c12 ?t)) )
:effect (and (
    increase (fuelLevel ?g) (* #t 0.5))
        (at  start (s1 ?t))
        (at  end (not(s1 ?t))) ) )

(:durative-action strt-refueling2
:parameters (?g - generator ?t - tank)
:duration (= ?duration 5)
:condition (and (at  start (c12 ?t))
        (at  end (c23 ?t)) )
:effect (and ( increase (fuelLevel ?g) (* #t 1))
        (at  start (s2 ?t))
        (at  end (not (s2 ?t))) ) )

(:durative-action strt-refueling3
:parameters (?g - generator ?t - tank)
:duration (= ?duration 5)
:condition (and (at  start (c23 ?t)) )
:effect (and (
    increase (fuelLevel ?g) (* #t 1.5))
        (at  start (s3 ?t)) ) )

(:durative-action clp01
:parameters (?g - generator ?t - tank)
:duration (= ?duration 0.15)
:condition (and (at  start (s0 ?t))
        (at  end (s1 ?t)) )
:effect (and (at  start (c01 ?t))
        (at  end (not (c01 ?t))) ) )

(:durative-action clp12
:parameters (?g - generator ?t - tank)
:duration (= ?duration 0.15)
:condition (and (at  start (s1 ?t))
        (at  end (s2 ?t)) )
:effect (and (at  start (c12 ?t))
        (at  end (not (c12 ?t))) ) )

(:durative-action clp23
:parameters (?g - generator ?t - tank)
:duration (= ?duration 0.15)
:condition (and (at  start (s2 ?t))
        (at  end (s3 ?t)) )
:effect (and (at  start (c23 ?t))
        (at  end (not (c23 ?t))) ) ) )
```

Figure 13: Asymmetric Clips Domain

Building Support for PDDL as a Modelling Tool

Derek Long and **Jan Dolejsi** and **Maria Fox**
{dlong6, jdolejsi, mfox2}@slb.com
Schlumberger, UK

Abstract

The paper describes support for use of PDDL as a modelling language in solving real problems. The support is embodied in a Visual Studio plug-in and is being used by engineers in the construction of domain models used in plan-based automation.

1 Introduction

Planning is a venerable branch of AI research, with roots extending back to the 1960s. Despite a few notable successes (Rajan et al. 2000; Chien et al. 2005; Nau et al. 2005; Muscettola et al. 1998), applications of planning have been slow to emerge. However, after 60 years of development, planning is beginning to find an audience: as automation and robotic control is becoming increasingly capable, the role of planning as a way to extend from the limits of scripted actions towards adaptable long-horizon goal-directed activity. It is interesting to begin to see non-academic positions being advertised that explicitly request expertise in planning and in modelling for planning. As planning emerges from the laboratories and code benches of academic researchers into practical roles, it is pressing to consider the extent to which academic research has successfully anticipated the needs of practitioners and what might need to be done to promote additional tools to make the existing research ideas practially useful.

In this paper we briefly outline some of the lessons learned in experiences in promoting PDDL and planning as tools for modelling and solving planning problems, exposing the technology to a wider user base with a very different starting point and motivation to the usual research or student audiences encountered by academics. We also discuss some of the tools we have started to develop to support and enhance the experience of such users in making planners practically valuable for real problem solving.

2 PDDL as a Practitioners' Language

A significant motivation for a large part of research in planning has been to identify and distill the activity of problem-solving, apparently exhibited by humans, that is independent of the particular domain to which it is applied. This has led to the idea of a completely domain-independent planner being parameterised for problem-solving in specific domains by supplying a declarative description of the activities that are possible in those domains. This idea focussed into the means to compare different planning systems, ensuring a clearer separation between the planning capability and the domain descriptions that fuelled it, through a standardised modelling language: Planning Domain Definition Language (PDDL) (McDermott 2000).

PDDL was conceived as an academic tool for researchers and was heavily based on Lisp (McDermott's preferred programming language). Its original specification attempted to support comparison of both the STRIPS-style planners and also of hierarchical planners. The latter ambitious objective failed — largely because hierarchical planning systems use rich and complex modelling languages that are essentially planning-programming languages (see the language used by SHOP (Au et al. 2011) for example), and it is much harder to achieve a consensus on what form such a language should take (compare the language of SHOP with that of ASPEN (Chien 2012), for example). As a result, this part of PDDL did not take root, but the declarative action-centred STRIPS-inspired core quickly led to an explosion in the field of propositional planning and the performance of such planners.

Although propositional STRIPS planning remains the most active sub-field of planner development, many in the research community have argued that practical planning systems must offer expressive power to represent temporal and metric problems. PDDL2.1 (Fox and Long 2003) was a direct response to this perceived need in order to make planning relevant and useful to a wider community.

Other variants of PDDL (PDDL+ (Fox and Long 2006), PDDL2.2 (Edelkamp and Hoffmann 2004) and PDDL3 (Gerevini et al. 2009)) are all attempts to extend the expressive power of the language to capture aspects of problems that are motivated by realistic examples. These include the addition of processes, events (both triggered and timed) and trajectory constraints. Nevertheless, there has been consistent criticism of PDDL as a poorly conceived language for practition-

ers, making modelling both difficult and non-intuitive and often dismissed as relevant only to the planning competition series.

The authors' experience is based on having introduced PDDL-based planning as a tool for problem-solving in an industrial context. Having exposed the language and paradigm to an audience of engineers interested only in the use of the tools in solving their problems, we have direct experience of how easily PDDL can be learned and used as a tool for modelling realistic problems and how effectively it supports users in harnessing planners.

3 Planning as a Research Subject versus Planning as a Users' Tool

As academic researchers in planning, the main focus of interest tends to be in extending the capabilities and functionality of planners. The research community is typically rather uninterested in work that focuses on the use of planning in particular applications other than as a motivation for further extension of capabilities of planners. This focus is reflected in the courses that are taught on planning, which spend great effort on the way that planners do their job but almost no time on the task of building models for planners to exploit. It is an interesting and challenging change of mindset to move from teaching a course on how to automate planning to teaching a course on how to use planners to do planning. The audience for such material is not concerned with how the planner does its job, but with how to harness the planner to do the jobs they care about.

We have now taught a series of short (3 day) courses on modelling for planning to engineers with no prior experience of planning or, in general, of AI modelling. Many of the participants are software engineers, but by no means all of them. Most are familiar with Matlab as a tool for modelling and programming, but few have used optimisation modelling tools such as linear programming.

Our course has been based on an action-centred modelling paradigm, using PDDL as the language (with no wrapper and no apology) and a few planners to illustrate the range of capabilities that the users can expect. Our courses start with a basic introduction to what we mean by a plan, how such a thing can be useful in solving problems that are more familiar to the audience, and then move into exposure of PDDL. We start with propositional models, using simple examples and asking the participants to change existing models to add new actions, or new object types, to achieve more interesting behaviour. The participants end the first day by building a larger model of a domain from a blank sheet start. The second day sees the introduction of numbers, then simple time and some effort is spent in showing the implications of modelling on concurrency and the ways that invariants, initial and end conditions all impact on the behaviour of the planner. On the third day we move on to more complex temporal models and illustrate the use of continuous change models. By the conclusion, participants are confidently building models that involve actions that interact with continuous process effects embedded in durative actions. An important aspect of the course is an intensive hands-on experience with a large number of exercises, in which the tutors spend time discussing answers and problems with participants.

From the outset, it was clear that the support for modellers, as opposed to planning researchers, is far from sophisticated. Even the few tools that exist are limited by being research systems, incomplete, brittle and difficult to deploy to the multitude of environments different participants favour. Much of the research software will deploy in a Linux environment, but prove difficult to transfer to a Windows environment, or to MacOS. We collected the tools we believed would be most useful, based on our own experiences in modelling domains for application, and ensured that these could be provided as binaries for Windows and Linux (MacOs support has not yet been pursued as far). In particular, obviously planners themselves (and there are very few planners that offer robust support for a wide range of the PDDL language), the validator, VAL (Howey, Long, and Fox 2004), and a parser with limited error checking. We have found that these tools provide an adequate basis for participants who have some experience at working with software tools and with technical material. However, most participants find the lack of good editing support an irritation in trying to develop their models and this has led us to spend effort in improving this area, which we discuss in the next section.

Our experience in teaching these courses has been very positive — participants enjoy the experience and we have had success in encouraging a group of users who are starting to make use of planners as one of the techniques they turn to in problem solving. Our most recent version of the course was recorded and has subsequently been used for online training without direct participation of the lecturers. Applications of planning are developing within the company and there is growing need for skilled modellers as well as the software integration experts who can aid in harnessing the plans to successful exploitation.

Amongst the lessons we have learned in teaching these courses is the importance of spending time on the subtleties of temporal models. There are common mistakes that we have now observed in the construction of temporal models. An example is the natural tendency to extend propopsitional models in temporal models by setting preconditions at the start of durative actions and effects at the end, with no consideration of the interactions between the bodies of these actions and the starts or ends of other actions. This often leads to unexpected exploitation of concurrency by the planner, using actions in unintended patterns. Understanding the need to change object states at the start of actions, while the objects are locked to their role in a durative action, and then releasing them at the end is an idiom

that we have learned to discuss in more detail. Another common trait in early modelling is the use of positive effects and the neglect of negative effects: it is common to see models that only have positive effects of actions. This can lead to successful construction of plans from nominal initial states, but peculiar failures when the planner is asked to replan from an intermediate state. Exposing this has led us to explore additional tools to support modelling, discussed below.

3.1 ICKEPS: A Community Perspective

The ICKEPS community has explored the challenges of domain modelling for planning (not only in PDDL). The 5th competition ran most recently (chr) and led the organizers, Chrpa *et al*, to observe the facts that most participants did not use tools other than standard text editors for modelling and that tools do not support collaboration. They also noted the small community of participants relative to the research community that hosts it. In our view, the latter observation is a consequence of the separation of concerns between building and using planners and our own experiences have demonstrated the significance of this distinction. The first of their observations appears to point to the limitations of existing tools and, we believe, is addressed in part by the tool we describe in Section 4.

Collaborative domain construction remains a challenging issue and one that is of interest to us. At Schlumberger, engineers have constructed and maintain what we believe to be one of the largest PDDL domain models developed, containing more than 4000 lines of PDDL code describing hundreds of action schemas. This is maintained by a team of several authors, under strict version control. Collaborative development is restricted to discussion and pair coding, rather than simultaneous editing of common files.

4 Tools for PDDL Editing and Validating

Although there have been efforts amongst the research community to provide tools to support modelling (Vaquero et al. 2013; Muise 2016; Simpson et al. 2000), we have found that these tools do not support the modelling we are most interested in: PDDL temporal and metric domains. Furthermore, these tools do not attempt to exploit integration into the environments that most programmers are familiar with. As part of the development of better support for the modellers, Dolejsi has constructed a VS Code plug-in to aid PDDL domain construction.[1]

The plug-in offers the basic support one might expect: syntax highlighting, parse-error reporting, autocompletion and template generation (for domains, actions, durative actions and problem files). The plug-in is sensitive to the structure of the PDDL domain: it offers hover functionality to report comments from the context of the definition (of types, predicates and functions) automatic navigation between instances of types, predicates and functions and auto-completion of these symbols.

Figure 1: Plan visualisation

Figure 2: Improving plan quality in output from POPF

As noted above, practical modelling for real problems demands temporal and metric modelling and this tool supports these features of PDDL as first-class features. This makes the tool a practical and valuable way for modellers to build and maintain sophisticated models of real domains. Prior editors have offered strong support for propositional planning, but very little for temporal domain modelling.

The plug-in also allows a planner to be hooked into the environment, so that it can be run directly from within VS Code showing console output. A feature of this is that the plan is shown as a Gantt-chart-like structure (Figure 1) in a new window within the environment, showing the actions along a timeline, and also showing swim-lanes of activities for each of the objects used in the plan and plots of the values of numeric state variables across the timeline of the plan. It is straightforward to configure the plug-in to work with different planners, to work with command line flags to the planners and to manage the output to show the plans in the visualisation (Figure 2 shows the sequence of improving plan quality values for output generated using POPF in OPTIC mode).

The tool supports structured initialisation of initial

[1] `https://marketplace.visualstudio.com/items?itemName=jan-dolejsi.pddl`.

Figure 3: Initialising structured initial state facts

state facts for situations where symmetric predicates or functions are required, or where an ordered sequence of objects must be specified through a *next* predicate or similar (Figure 3).

4.1 VAL and related tools

VAL continues to be a fundamental tool supporting modelling: the ability to use manually constructed plans as tests for the capabilities expressed in the domain model allows the user to confirm that plans expected to be valid are actually valid plans in the model, and also to see the trajectories of states visited by plans from the planner in order to understand how unanticipated plans might be falsely supported by a model. We have extended functionality in VAL to include incremental stepping through the plan programmatically, so that it is possible to then extract intermediate states in PDDL format, to be used as the initial state for planning problems in intermediate states. This function also supports the ability to generate and then perturb intermediate states to test replanning functionality.

This functionality is available through VS Code, allowing automatic stepping through a plan, extraction of intermediate states and replanning from those states. Additional information from VAL is used to construct graphs of numeric values of state variables throughout a plan, in order to track values of these variables and see how they change over time.

5 Conclusion

The authors have been jointly introducing and promoting planning technology for problem solving as a tool across multiple segments of their host industry. There has been successful adoption of planning for a variety of purposes (which we will report in due course). Support for modelling is a crucial part of ensuring the successful adoption of the technology and we have learned, in giving courses on modelling, some of the important differences between communicating the modelling skills required to capture particular domain features and presenting the planning techniques that allow planners to plan with these features.

We have put together a collection of tools to help developers build models, using PDDL. We have found that PDDL is far from a barrier to modelling — in fact, it is largely as intuitive and accessible as had been hoped and envisaged by those who contributed to the series of extensions and modifications of the language. The important gap in the tool suite has been an editor support tool that works within the environment most familiar to developers and this has been created and is now a technically supported tool for the community to use. The plug-in has so far been installed well over 1000 times and we anticipate its continuing growth and extension to further capability. Provision and development of these tools is a key ingredient in accelerating the adoption of planning as a standard tool for regular problem-solving.

References

Au, T.; Ilghami, O.; Kuter, U.; Murdock, J. W.; Nau, D. S.; Wu, D.; and Yaman, F. 2011. SHOP2: an HTN planning system. *CoRR* abs/1106.4869.

Chien, S. A.; Sherwood, R.; Tran, D.; Cichy, B.; Rabideau, G.; Castaño, R.; Davies, A.; Mandl, D.; Trout, B.; Shulman, S.; and Boyer, D. 2005. Using autonomy flight software to improve science return on earth observing one. *JACIC* 2(4):196–216.

Chien, S. 2012. A generalized timeline representation, services, and interface for automating space mission operations. In *SpaceOps 2012*. 1275459.

Edelkamp, S., and Hoffmann, J. 2004. PDDL2. 2: the language for the classical part of the 4th international planning competition. Technical Report 195.

Fox, M., and Long, D. 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *J. Artif. Int. Res.* 20(1):61–124.

Fox, M., and Long, D. 2006. Modelling mixed discrete-continuous domains for planning. *J. Artif. Intell. Res.* 27:235–297.

Gerevini, A.; Haslum, P.; Long, D.; Saetti, A.; and Dimopoulos, Y. 2009. Deterministic planning in the fifth international planning competition: PDDL3 and experimental evaluation of the planners. *Artif. Intell.* 173(5-6):619–668.

Howey, R.; Long, D.; and Fox, M. 2004. VAL: automatic plan validation, continuous effects and mixed initiative planning using PDDL. In *16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2004), 15-17 November 2004, Boca Raton, FL, USA*, 294–301.

McDermott, D. V. 2000. The 1998 AI planning systems competition. *AI Magazine* 21(2):35–55.

Muise, C. 2016. Planning.Domains. In *The 26th International Conference on Automated Planning and Scheduling - Demonstrations*.

Muscettola, N.; Nayak, P. P.; Pell, B.; and Williams, B. C. 1998. Remote agent: To boldly go where no AI system has gone before. *Artif. Intell.* 103(1-2):5–47.

Nau, D.; Au, T. C.; Ilghami, O.; Kuter, U.; Wu, D.; Yaman, F.; Munoz-Avila, H.; and Murdock, J. W. 2005.

Applications of SHOP and SHOP2. *IEEE Intelligent Systems* 20(2):34–41.

Rajan, K.; Bernard, D. E.; Dorais, G.; Gamble, E. B.; Kanefsky, B.; Kurien, J.; Millar, W.; Muscettola, N.; Nayak, P. P.; Rouquette, N. F.; Smith, B. D.; Taylor, W.; and Tung, Y. 2000. Remote agent: An autonomous control system for the new millennium. In *ECAI 2000, Proceedings of the 14th European Conference on Artificial Intelligence, Berlin, Germany, August 20-25, 2000*, 726–730.

Simpson, R. M.; McCluskey, T. L.; Liu, D.; and Kitchin, D. E. 2000. Knowledge representation in planning: A PDDL to OCLh translation. In *ISMIS*.

Vaquero, T. S.; Silva, J. R.; Tonidandel, F.; and Christopher Beck, J. 2013. itsimple: towards an integrated design system for real planning applications. *The Knowledge Engineering Review* 28(2):215–230.