Planning Domains and Plans: Validation, Verification and Analysis

Derek Long and **Maria Fox**
{first.last@cis.strath.ac.uk}
University of Strathclyde, UK

Richard Howey
{richard.howey@ed.ac.uk}
University of Edinburgh, UK

Abstract

In this paper we discuss the nature of the verification and validation problems as they apply to planning domain specification, construction and use. We consider the extent to which it is a real problem and highlight a few examples of problems in simple domains as evidence of the ways that bugs can appear in domain models. We then proceed to examine tools that can be used to help in resolving the problem of domain specification validation and verification, before illustrating the verification of a model of a battery domain.

1 Introduction

A considerable part of the research in AI Planning is focussed on finding and improving heuristics that can guide domain-independent planners through a search space, regardless of the domain the search space represents. One consequence of this focus is that there are many benchmark domains that have been used by a wide part of the community for testing and evaluation, but which have not been closely inspected by very many more people than the original developers. Furthermore, it is common for evaluation to consist of timing the process of production of a plan, without necessarily confirming the validity of the plan — it is often assumed that a planner is producing valid plans, since the planner is believed to be sound. This is interesting from a validation point of view, since it means that domains are frequently subject only to the inspection of the original designers, often with supposed plans for problem instances associated with the domains being accepted without formal validation.

In this paper we examine some of the problems that can arise in planning domains, problem instances and plans and the role of automated tools, static analysis and planners themselves in the process of validating and verifying domains and plans.

2 Verification of Formal Specifications

Verification involves comparing two (or more) separate specifications of a target in order to confirm that they describe the same thing. To do this assumes that the specifications can be given a common semantics, so that the meaning of the different specifications can be usefully compared. It also assumes that the specifications describe the target in the same way (same level of abstraction, same granularity and so on). In case one specification is more abstract than the other it can be possible to identify an abstraction mapping that can be applied to the more detailed specification to raise its level of abstraction to match that of the other specification. In this case, the comparison can, of course, only compare the specifications at the higher level of abstraction, so the separate question of whether the realisation of the more concrete specification is valid cannot be addressed directly by this comparison. A challenge that faces most efforts at verification is to acquire two or more specifications of a target written in languages with sufficiently precisely defined semantics to allow a direct comparison. In planning, for example, it is common to have a formal specification of the domain written in a language such as PDDL (Fox & Long 2003), but very unusual to have any other formal specification to compare it with.

If it is hard to find multiple formal specifications of the same target, what other options are open? A common approach is to use a formal specification to infer consequences, often adopting additional hypotheses describing initial conditions, in order to compare these consequences with those expected. This can be easier that constructing multiple formal specifications, since the expected inferences can frequently be characterised far more compactly and efficiently. In the case of software verification, this approach includes testing, where unit tests, for example, specify particular input contexts and the expected output of the program unit for each test. More complex software tests can specify conditions that must be satisfied by the output, rather than the value of the output itself. The output of a test run can then be checked to confirm it meets these tests. In essence, the comparison of two formal specifications of outputs corresponding to a particular input becomes a surrogate for the comparison of two formal specifications of the target software module. More generally, this approach to verification can be seen illustrated in figure 1: a single formal specification of a model combined with the much simpler specification of a test context allow the automatic construction of an implied result that can be checked against formally specified properties of the expected result.

In the case of planning, a domain model might be tested by verifying that plans generated by a planner using the

Copyright © 2009, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

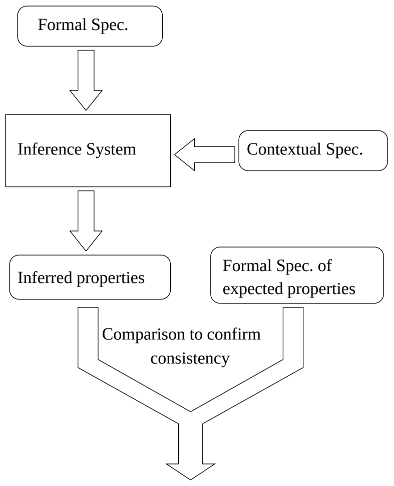
Formal Spec.
Inference System
Contextual Spec.
Inferred properties
Formal Spec. of expected properties
Comparison to confirm consistency

Figure 1: Verification against a formal specification of properties.

model satisfy specified properties. This process rests on an assumption that the planner itself is sound. It is important to observe that this plan testing process is not the same as validating plans, since a sound planner will generate valid plans for the domain model it has been given, even if the model is an inadequate image of the intended target. Therefore, the verification-by-testing process can only be automated for planning domain models if there is an additional specification of the properties of plans that solve a particular problem instance.

As an aside, we observe that verification-by-testing is always limited, since the inference process can be seen as analogous to an abstraction, so that many different models could lead to the same inferred results, despite disagreeing about the details from which the inferences are drawn. Systems can agree on a collection of inferences drawn from different specifications, despite the specifications describing entirely different targets. However, the most sophisticated verification techniques infer complex properties of specifications, such as termination guarantees and correctness of outputs for a wide range of inputs, so that the details that these techniques leave abstracted can be reasonably described as implementation choices. Techniques used to check properties such as termination include well-known model-checking approaches (Holzmann 2004; Behrmann, David, & Larsen 2004; Cimatti *et al.* 2002). The key to being able to apply these techniques for such systems is that the desirable properties, such as termination, can be specified in concise formal statements. A challenge for planning domain modelling is to find similarly concise formal statements that can express the desirable properties of planning domains.

3 Verification of Planning Domain Models

Although planners are finding increasing application, the methodology supporting the development of domain models is still in its infancy. Despite the efforts in the International Competition for Knowledge Engineering for Planning Systems (ICKEPS) (Barták & McCluskey 2007), there are few tools providing engineering support and even fewer that attempt to provide any kind of validation or verification functionality. In section 5 we will consider some of the tools and techniques that can provide support in the verification process, but here we review the precise nature of the range of verification tasks that are of interest in planning.

Planning is traditionally taken to be a generic problem solving problem, in which the input to a planning system specifies a domain model, an initial state and a goal. A planner is then expected to deliver a solution plan that would coerce the initial state into one that satisfies the goal if executed in the domain the model describes. The verification tasks of interest include determining that a planning system itself is sound or complete with respect to arbitrary planning tasks (termination on unsolvable problems can also be of interest). This is actually a standard software verification task, since a planner is simply an algorithm for solving the planning problem.

A further validation task is to confirm that a domain model describes the domain dynamics it is intended to capture. Closely related to this (and possibly not even separable from it) is the task of confirming that a given initial state is a consistent state for the domain model and one that conforms to the target being modelled. Finally, there is the task of validating or verifying solution plans for a given domain and initial state (and confirming that they satisfy specified goals). This last task is clearly redundant if the plans are produced by a planner that has been proved sound. Unfortunately, in planners we see the same gap between formally specified algorithms (whether or not they are proved sound) and the details of an implementation that undermines the power of verification in other areas of software development. In fact, a further separation between proofs of soundness and the production of sound plans lies in the fact that there are often differences (usually only minor) between the semantics of domains assumed in the determination of planner soundness and the semantics assumed in validating plans. Such discrepancies lie in subtle corners of the modelling language, affecting such things as the precise constraints on when actions can be executed simultaneously or by how much interfering actions must be separated.

Of these tasks, validating domain models remains the most challenging. The source of the challenge is that, as with software systems, it is almost impossible to find resources to support separate development of independent formal specifications of the same target. In practice, planning is still at a stage where a single domain model is constructed, iteratively, by a planning expert, perhaps consulting with a domain expert (where relevant). The equivalent of unit testing for planning domains is the creation of test scenarios and plan inspection to confirm the outputs meet the expected criteria, although this merges validation and verification.

Although this seems rather a weak position, a natural question is, given that planning domain models are often declarative in style (although this ignores HTN planning domain models and models that depend on control rules), might it be the case that domain bugs are relatively uncommon compared with software bugs? We now offer some examples of simple benchmark domains, developed by various planning experts, that contain bugs that have remained undiscovered for years. One of the reasons for this long lag before bugs are uncovered is that most planning researchers treat benchmarks as evaluation material, assuming that since the domain has been used for evaluation in many cases in the past, it must be valid.

3.1 Related work

The problem of planning domain validation and verification is one that has been confronted by several researchers, particularly those working in domains with safety-critical requirements. For example, Bedrax-Weiss *et al* (Bedrax-Weiss *et al.* 2005), Khatib *et al* (Khatib, Muscettola, & Havelund 2001) and Penix *et al* (Penix, Pecheur, & Havelund 1998) are all examples of work examining the problem of planning domain model verification in the space domain. This work includes attempts to compile domain models into forms where model checkers can be used to verify formal properties and other work exploring techniques for supporting inspection-based approaches to model validation.

McCluskey and co-authors (McCluskey & Porteous 1997; McCluskey & Simpson 2004) are amongst a group of researchers who have considered the process of engineering domains and ways in which tools can be used to support the construction of valid domains. This strand of work mirrors the efforts in software development to automate aspects of code-production in order to reduce the scope for error. In the most recent work, this research has led to the development of a tool to support the construction of planning domains, providing some visualisation and state-based modelling of finite-state machines for the capture of domains. Edelkamp *et al* (Edelkamp & Mehler 2003) have also explored a similar tool design.

Although automated approaches are sought by many researchers, in practical application much of the validation and verification of real systems remains a largely manual process, exploiting aid from simulations and physical models. Cichy *et al* (Cichy *et al.* 2005) describe this process as it applied in the validation of the EO-1 Science Agent.

Despite the efforts of researchers such as these, the validation of planning domains remains limited and underexplored.

4 War Stories: Simple Examples of Flawed Domains

The first domain we examine is the Settlers domain from IPC3 (Long & Fox 2003a). This domain models an infrastructure development and production problem, based on an computer game of the same name. The actions allow basic resources (wood and stone) to be collected and used to develop more complex production buildings which can allow access to more refined or advanced resources (iron, cut timber and coal). Finally, housing and transportation infrastructure can be built. The last of this includes carts, trains and railway networks and also ships.

This domain is rich in numeric fluents which are used to record quantities of stock at different locations, making it very challenging for most planners and few researchers have experimented very far with the domain. Nevertheless, it has been available since 2002 and has been examined and used for testing since then. It serves to highlight a weakness of PDDL, which is that the closed collection of object names and lack of functions makes it necessary to decide in advance of solving an instance, just how many vehicles of each type might be needed for construction. The interest in the domain encoding for this work lies in actions that manage coastal development.

Examination of the domain reveals that the development process at a coastal location begins with construction of a dock and then with a wharf (which would have been more appropriately named a shipyard). However, the build ship action is very strange: it requires *only* that sufficient iron be available and as a consequence it *creates a wharf* and a ship at a location that might not even be coastal! Clearly, the effect of creating a wharf was intended to be a precondition that should have ensured that a ship could only be built on the coast at appropriately equipped locations. Further, the action of moving a ship requires only that locations be connected by sea, whereas it seems more reasonable that the destination should have a dock (the start would always have a dock if ships could only be built at wharfs and only sail to docks). This strange behaviour passed apparently unnoticed and unchallenged for 6 years, despite planners having been used on the domain.

There are other features of the Settlers domain that are less errors than oddities. The resource cost to construct trains is very significant, because of the need to build up rail links in order to use them. Furthermore, a train must carry coal to fuel its own movement, using its own storage space to hold it. Since the capacity of a train is only 5, it is almost impossible to imagine circumstances in which building a train to transport materials will be more efficient than using carts. Similarly, ships have a capacity of only 10 but demand a great deal of infrastructure to support them, so that it is unlikely that ships will be a wise investment unless materials have to be transported to locations that are otherwise inaccessible. The fact that these oddities have not been observed seems to be linked to the fact that planners can currently solve only small instances where it is unsurprising that significant infrastructure is needed. This serves to emphasise that inspection-based debugging of planning domain models depends on the creation of problem instances that exercise all of the actions in the domain.

A second example can be seen in the MPrime domain, originally designed by Drew McDermott for the 1st IPC. This domain was a deliberately disguised version of a transportation problem using vehicles that consume fuel from stores at the locations they travel from. The domain included an action that allowed fuel to be transferred from one location to another, consuming two fuel units from the source to generate one at the destination. However, the transfer action could be instantiated to transfer fuel from a location to itself which, combined with the fact that the domain used propositional encodings of the fuel counts, made it possible for a location to end up simultaneously having one fewer and one greater fuel units than it started with. While the domain was disguised it was not apparent that this was a mistake: the model describes a behaviour that is syntactically valid, even if inconsistent with the intended interpretation.

Two final examples serve to illustrate the greater difficulties that arise in capturing temporal domains. Firstly, Match-Lifts, a domain created by Keith Halsey for the original tests on the CRIKEY planner (Coles *et al.* 2009), represents an extended version of the fuse fixing domain designed by Fox and Long for testing LPGP (Long & Fox 2003b). The extension has multiple electricians moving between floors on lifts to fix fuses at different places in the building. The problem with this domain is that the action that allows an electrician to leave a lift is a durative action, requiring the electrician to be in the lift at the start of the action, but deleting this fact only at the end of the action when the new location of the electrician is asserted. This is a problem because the same electrician can start to move from the lift to a different room on the same floor after starting a first move, but before finishing it. The result is that when the second move completes the electrician is no longer in the lift, but deleting a false condition is not an error, so the electrician can end up being in multiple locations at the same time. A clever planner can exploit this to have the same electrician fixing fuses in many rooms using the light of the same match!

The second temporal example is the Timed Rovers domain from IPC3. In this domain there is a recharging action that allows a rover to increase its energy. The effect of the action is to increase the charge at the end of the action by the recharge rate multiplied by the duration of recharging, which is fixed at the outset of the action to be exactly long enough to raise the charge level back to maximum capacity. Unfortunately, the planner can apply multiple recharge activities in parallel with one another, allowing the rover to charge far beyond its maximum capacity. This bug looks as though it might be fixed by simply asserting at the end of the action that the rover has its maximum charge, but things are not so simple: the reason why the effect was described as an increase effect is that the rover might consume charge while it is recharging, so that after recharging for enough time to completely recharge based on its charge level at the *start* of the action would not in fact be sufficient to completely recharge it at the end.

5 Tools and Techniques for Verification and Validation

Having seen that existing benchmark planning domains can contain bugs, despite being exposed to a relatively wide audience and being models of lower complexity than we might expect in real applications, we now turn to the question of what can we actually do about this. First, it is worth considering the source of complexity in the structure of planning domain specifications. It is interesting to observe that planning domain descriptions are very small by software engineering standards. A domain description of a few thousand lines would be considered huge — more typical is a few hundred lines or less, certainly for benchmark domains — compared with the multi-million lines of source code in large modern software systems or thousands of lines for small individual projects. The source of complexity in the planning descriptions arises from the highly declarative form they take combined with the complex interactions between the behaviours of different component subsystems within a domain: where modern software engineering approaches pursue modularity and reduction of interaction between components, planning domains decompose behaviours, but depend on interaction between components. Thus, where software engineers can attempt to tackle the complexity of verification by exploiting the decomposition of software into functionally independent units, planning domain engineers must confront the opposite requirement, considering exactly the most complex interactions between multiple components. Interestingly, the most similar verification tasks in software engineering appear to be those involving multithreaded and parallel systems, where deadlock and race conditions are notoriously difficult to find and eliminate (eg (Eytani *et al.* 2007)).

Although it is often too much to expect that multiple specifications be constructed for the same target domain, it can nevertheless be a powerful technique for supporting verification by inspection if a specification for a target is presented back to its writer in different forms. This process enhances inspection by requiring the reader to go through a process of mental evaluation. The familiar problem of reading what is expected rather than what has been written can be tackled by the process of translation into new formalisms. There are several possible formalisms that can act as candidates for the expression of specification of planning domains: an action-centred representation such as PDDL (McDermott 2000), a constrained-timeline-based representation such as DDL (Muscettola 1994), or one of its variants, and an object-centred representation such as SAS+ (Bäckström & Nebel 1995) or OCL (McCluskey & Porteous 1997). These differ as follows:

1. An action-based representation focuses on the mechanisms of change within a domain, providing pre- and post-condition specifications for each possible state transition that can be enacted within the domain. In this case, state is treated holistically, viewed as the entire configuration of the world. This representation emphasises the causal relationships within a domain (which actions *cause* particular changes and what do they depend on?), also highlighting the relationships between objects that enable the transitions between states.

2. A constrained-timeline-based representation focuses on the temporal relationships that govern the intervals over which individual objects are in particular states, placing constraints on the ways in which these intervals interact, sequencing, overlapping or enveloping one another to support the behaviours of the individual objects.

3. An object-centred representation focuses on the individual objects of the domain, emphasising the values of properties of these objects, including states that they may be in and the transitions that are possible between these states or changes in associated property values. The roles of

supporting objects are denoted as constraints on the transitions made by the dependent objects.

Other formalisms are used to define planning domains, such as temporal logic (Bacchus & Kabanza 2000) or hierarchical task networks (Nau *et al.* 2003), but these languages are not used to capture planning domains in the same ways as those listed above, so translations between specifications in these languages and those written in any of the other languages (or vice versa) are either currently impossible or else create highly artificial compiled forms that are not useful for the manual inspection process. The artificial form of compiled representations makes pure constraint formalisations of planning domains and SAT representations equally inappropriate as media to support manual inspection.

The three formalisms listed above are very similar in terms of their expressive power and in the content of the domain specifications they each support. Automatic translations between them have been explored in various different pieces of work, sometimes directly with the intention of providing a translation between formalisms and sometimes because one or other perspective on a domain offers an opportunity to exploit different heuristic planning techniques. Examples of these are Helmert's PDDL-to-SAS+ translator (Helmert 2006), a PDDL-to-DDL translator (Bernardini & Smith 2008) (which also exploits the PDDL-to-SAS+ translation), a (partially completed) DDL-to-PDDL translator [1] and an OCL-to-PDDL translator in GIPO (McCluskey & Simpson 2004; Simpson, Kitchin, & T.L.McCluskey 2007). Several of these translators are either under development or are only partially complete, but they still provide useful tools to support the revisualisation of specifications by translating from one formalism to another and show how these techniques could be developed into fully fledged domain engineering support tools. The key to using these translations is to present specifications to their writer in new ways, forcing the writer to reconsider what has been specified and to decide whether it captures the intended aspects of the domain behaviour.

Another family of tools that can be used to help in reformulating elements of the behaviour of a planning domain is that of invariant synthesisers such as TIM (Fox & Long 1998) and DISCOPLAN which analyse planning domains to extract invariant constraints on the behaviours of objects. This process is a precursor to reformulating domains in other forms (SAS+ or DDL) but presenting the invariants explicitly to a domain engineer can offer a new perspective on the behaviours the domain describes that can highlight domain errors that are less easy to identify once the domain has been translated and the invariants become implicit.

TIM proved to be useful in analysing the MPrime domain and allowed us to discover the bug in the behaviour described in it.

5.1 Temporal and Metric Domains

The translation approaches outlined above are generally less capable when confronting domains describing richer temporal and metric behaviours. In this level of expressiveness for modelling the planning community has both fewer well-developed tools (including planners) and less collective experience in engineering and debugging domains. However, the challenges in managing these domains are more daunting than those that we face in handling propositional domains, since the forms of interactions are both more subtle and can fail in more complex ways. TIM has been extended to find invariants in temporal domains expressed in PDDL, while consideration of rich temporal domains as timed automata (Fox & Long 2006) might offer new ways to present the formalisations of temporal domains. Metric behaviours have been explored very little. One possibility that might offer ways to analyse these behaviours for verification purposes is to borrow from program analysis techniques, exploiting ideas such as abstract interpretation and resource profiling.

TIM could have offered a clue about the unintended behaviour in the Rovers domain, since it shows that the recharge action is not self-mutex. The MatchLifts domain revealed its bug when we applied CRIKEY3 (Coles *et al.* 2008) to it and it exhibited a strange behaviour. This, of course, is a less-than-ideal way to find the bug! The Settlers domain bug became apparent after a prolonged inspection in order to remodel the problem in a constraint formulation. This process was carried out manually, but it hints at the possibility that translation of this kind of domain could lead to discovery of bugs such as lay dormant in Settlers.

6 Plan Validation

The problem of confirming a plan is correct with respect to a formal domain model has traditionally been called *plan validation*, although it might be seen as a verification task, since it is not concerned with whether the formal model correctly captures the target domain. The most well-developed tool for PDDL domain encodings is the Validator (Howey, Long, & Fox 2004), which can check the executability of a plan and whether it achieves the appropriate goals. It evaluates the plan according to the indicated plan metric and can also test plans against trajectory constraints and preferences. It will give advice about what might be wrong with a plan when a plan fails and can report on many other features of a plan, such as its robustness to temporal or metric variability, its precise use of resources and the profile of this use over time. It can also handle continuous change, using analytic techniques to find roots of polynomials and numeric techniques for some other functions such as exponential and logarithmic functions.

As an illustration of the power of the Validator in providing insight into the behaviour of a domain encoding, we have used it to explore the behaviour of a simple model of batteries under load. We took as the starting point for our model a two-charge-well model described in (Jongerden *et al.* 2009) (illustrated in figure 3). In that paper the model was developed for use with a model checker (UPPAAL), using discretised time. The idea behind the model is that a battery delivers charge from its free charge well, according to the demand of a load placed on the battery, while charge passes from the bound-charge well of the battery into the free-charge well,

[1] Unpublished work pursued by M. Fox and D. Long during a visit to NASA Ames in 2000.

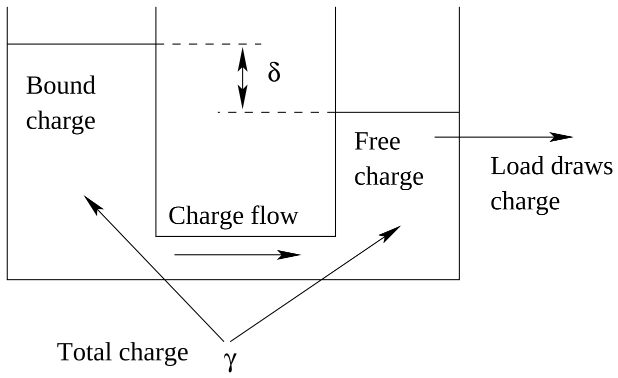

Figure 2: The kinetic battery model.

flowing like a liquid to equalise the heights of the two wells. Properties of the battery determine the relative widths of the two wells, the flow rate between the wells and the initial charge levels. The model offers a simple explanation for the observed phenomenon that a battery regains charge if it is allowed to rest between loads. More sophisticated models are possible, but the two-well model offers a reliable first order approximation of battery behaviour in practice.

This system illustrates an important aspect of the process of modelling a physical system: the first stage in building a model is to arrive at a conceptualisation of the physical system that is sufficiently precise that it can be captured in a formal specification. The question that verification and validation seeks to address is how well the specification captures the conceptualisation. The broader question of how well the conceptualisation captures the physical system cannot be addressed by verification, but only by empirical study of the physical system itself.

In our PDDL model of batteries we used PDDL+ (Fox & Long 2006), creating a process modelling the flow of charge from the bound-well to the free-well and a process of discharging under load initiated by turning on the battery load and ended by turning it off. The model includes an event triggered by the free-charge well becoming empty, which is that the battery is dead (even though it might contain remaining charge in the bound-charge well). The most complex element of this model is the interaction of the processes: discharging is a linear process, proportional to the load, but the rate of flow of charge from the bound-charge to the free-charge well is proportional to the difference in the heights of the charge in each of the two wells (not quite the same as the charge, since the height is dependent on the width of the well as well as the total charge). The combination of the these behaviours yields a function that includes a negative exponential decay in the height difference.

It general, the problem of interest is to determine how long one or more batteries can be made to last using different policies. Using the Validator we can determine the time at which the batteries become dead within the model and compare it with the behaviour of the specification used in (Jongerden *et al.* 2009) with UPPAAL-Cora. The figures for a selection of scenarios (loads and policies) are shown in

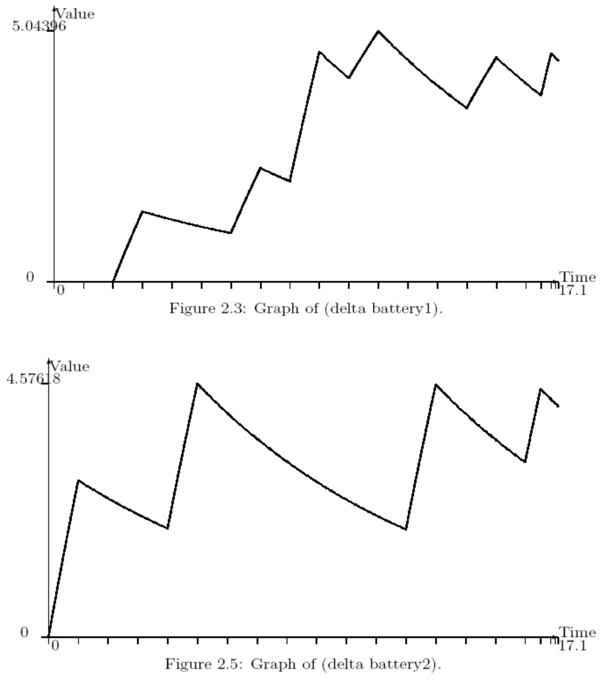

Figure 3: Plots of the differences in well heights for two batteries operating under different loads. These plots are generated automatically by the Validator.

Scenario	Lifetime KiBaM (min)	Lifetime U-C Model	Lifetime PDDL+ model
CL_250	4.53	4.56	4.526
CL_500	2.02	2.04	2.017
CL_alt	2.58	2.60	2.580
ILs_500	4.30	4.32	4.304
ILl_500	6.53	6.56	6.532

Table 1: Comparison of results from the mathematical kinetic battery model (KiBaM), the UPPAAL-Cora model and the PDDL+ model.

table 1. It is worth acknowledging that this problem is an optimisation problem rather than a natural planning problem, but the evaluation of policies is a necessary precursor to construction of policies and the use of the Validator to compare the behaviour of the specification in PDDL with that of the UPPAAL-Cora specification is an indication of the way that the Validator can be used as a tool to support the validation of domain models.

7 Conclusion

Validation and verification of planning domain models is still an poorly developed area of research. The construction of planning domain models is still the preserve of a relatively tiny group of researchers, limiting our experience of the problems that are typically encountered in the development of domain models and the kinds of errors that arise. The tools that software engineers use to help in the verification of software models can provide some insights into the areas that are likely to help in the verification of planning domain models. The software engineer still depends heavily on inspection-based techniques, but these can be enhanced by automatic analysis and translation into different formalisms. A further support for verification is to verify plans for domains, where the plans have been built by hand. This can be compared directly with performing unit tests in software engineering. The Validator provides a powerful profiler for determining the behaviour of plans and resources in domains. The Validator is probably the only PDDL-based tool that offers such comprehensive coverage of the metric and temporal elements of the language and can give deep insights into the behaviour of these aspects of a domain encoding.

References

Bacchus, F., and Kabanza, F. 2000. Using temporal logic to express search control knowledge for planning. *Artificial Intelligence* 116(1-2):123–191.

Bäckström, C., and Nebel, B. 1995. Complexity results for SAS+ planning. *Computational Intelligence* 11(4):625–655.

Barták, R., and McCluskey, L. 2007. Introduction to the special issue on knowledge engineering tools and techniques for automated planning and scheduling systems. *Knowledge Eng. Review* 22(2):115–116.

Bedrax-Weiss, T.; Frank, J.; Iatauro, M.; and McGann, C. 2005. Inspection and Verification of Domain Models with PlanWorks and Aver. In *Proceedings of ICAPS'05 Workshop on Verification and Validation meets Planning and Scheduling*.

Behrmann, G.; David, A.; and Larsen, K. G. 2004. A tutorial on UPPAAL. In Bernardo, M., and Corradini, F., eds., *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, 200–236. Springer–Verlag.

Bernardini, S., and Smith, D. 2008. Translating PDDL2.2. into a Constraint-based Variable/Value Language. In *Proceedings of ICAPS Workshop on Knowledge Engineering for Planning and Scheduling*.

Cichy, B.; Chien, S.; Schaffer, S.; Tran, D.; Rapideau, G.; and Sherwood, R. 2005. Validating the Autonomous EO-1 Science Agent. In *Proceedings of ICAPS'05 Workshop on Verification and Validation meets Planning and Scheduling*.

Cimatti, A.; Clarke, E. M.; Giunchiglia, E.; Giunchiglia, F.; Pistore, M.; Roveri, M.; Sebastiani, R.; and Tacchella, A. 2002. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In *Proceeding of International Conference on Computer-Aided Verification (CAV 2002)*.

Coles, A. I.; Fox, M.; Long, D.; and Smith, A. J. 2008. Planning with problems requiring temporal coordination. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI 08)*.

Coles, A. I.; Fox, M.; Halsey, K.; Long, D.; and Smith, A. J. 2009. Managing concurrency in temporal planning using planner-scheduler interaction. *Artificial Intelligence* 173(1):1–44. Available online August 2008.

Edelkamp, S., and Mehler, T. 2003. Knowledge acquisition and knowledge engineering in the modplan workbench. In *Proceedings of Workshop on Knowledge Engineering for Planning (ICKEPS)*, 22–29.

Eytani, Y.; Havelund, K.; Stoller, S. D.; and Ur, S. 2007. Towards a framework and a benchmark for testing tools for multi-threaded programs. *Concurrency and Computation: Practice and Experience* 19(3):267–279.

Fox, M., and Long, D. 1998. The Automatic Inference of State Invariants in TIM. *Journal of Artificial Intelligence Research* 9:376–421.

Fox, M., and Long, D. 2003. PDDL2.1: An Extension of PDDL for expressing Temporal Planning Domains. *Journal of Artificial Intelligence Research* 20:61–124.

Fox, M., and Long, D. 2006. Modelling mixed discrete-continuous domains for planning. *J. Art. Int. Research* 27:235–297.

Helmert, M. 2006. The Fast Downward Planning System. *Journal of Artificial Intelligence Research* 26:191–246.

Holzmann, G. 2004. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley.

Howey, R.; Long, D.; and Fox, M. 2004. VAL: Automatic plan validation, continuous effects and mixed initiative planning using PDDL. In *Proceedings of 16th IEEE International Conference on Tools with Artificial Intelligence*.

Jongerden, M. R.; Haverkort, B.; Bohnenkamp, H.; and Katoen, J.-P. 2009. Maximizing system lifetime by battery scheduling. In *Proc. DSN 2009, IEEE Computer Society*.

Khatib, L.; Muscettola, N.; and Havelund, K. 2001. Verification of Plan Models Using UPPAAL. In *Formal Approaches to Agent-Based Systems*, volume 1871/2001 of *Lecture Notes in Computer Science*. Springer. 114–122.

Long, D., and Fox, M. 2003a. The 3rd International Planning Competition: Results and analysis. *Journal of AI Research* 20.

Long, D., and Fox, M. 2003b. Exploiting a graphplan framework in temporal planning. In *Proceedings of ICAPS'03*, 51–62.

McCluskey, T. L., and Porteous, J. M. 1997. Engineering and compiling planning domain models to promote validity and efficiency. *Artificial Intelligence* 95(1):1–65.

McCluskey, T. L., and Simpson, R. 2004. Knowledge formulation for ai planning. In *Proc of EKAW'04: Engineering Knowledge in the Age of the Semantic Web*, 449–465.

McDermott, D. 2000. The 1998 AI planning systems competition. *AI Magazine* 21(2):35–56.

Muscettola, N. 1994. HSTS: Integrating planning and scheduling. In Zweben, M., and Fox, M., eds., *Intelligent Scheduling*. San Mateo, CA: Morgan Kaufmann. 169–212.

Nau, D.; Au, T.-C.; Ilghami, O.; Kuter, U.; Murdoch, J.; Wu, D.; and Yaman, F. 2003. An HTN planning environment. *J. AI Res.* 20.

Penix, J.; Pecheur, C.; and Havelund, K. 1998. Using Model Checking to Validate AI Planner Domain Models. In *Proceedings of the 23rd Annual Software Engineering Workshop*.

Simpson, R.; Kitchin, D.; and T.L.McCluskey. 2007. Planning domain definition using GIPO. *The Knowledge Engineering Review* 22:117–134.