PDDL: A Language with a Purpose?

T. L. McCluskey
Department of Computing and Mathematical Science,
School of Computing and Engineering,
University of Huddersfield, UK
email: lee@zeus.hud.ac.uk

Abstract

In order to make planning technology more accessible and usable the planning community may have to adopt standard notations for embodying symbolic models of planning domains. In this paper it is argued that before we design such languages for planning we must be able to evaluate their quality. In other words, we must clear for what *purpose* the languages are to be used, and by what criteria the languages' effectiveness are to be judged. Here some criteria are set down for languages used for theoretical and practical purposes respectively. PDDL is evaluated with respect to them, with differing results depending on whether PDDL's purpose is to be a theoretical or practical language. From the results of these evaluations some conclusions are drawn for the development of standard languages for AI planning.

Introduction

Good planning algorithms are hard to devise, but fairly easy to evaluate; on the other hand, modelling languages are fairly easy to devise, but hard to evaluate. Language extension is similar: it is relatively easy to add arbitrary features to a language, but adding the tools to manipulate the enhanced language, or perfecting a semantic definition of the extension, is much more difficult. Having devised a language[1], how can we evaluate it's quality? One way is to use *practical* methods. Experiments can be set up to test the effectiveness of a language, using engineers in a controlled environment. This is a time consuming and costly business, however, and the tests are prone to extraneous variables as people act differently when on their own to when they are being experimented on.

For reasons such as these, more analytical methods of evaluating languages are popular. This involves generating a list of criteria, usually called *design criteria*, that have been devised when considering the *purpose* of the language. Sometimes these criteria are well developed a priori, and sometimes old languages are subject to being evaluated with new criteria. A well-used language does not necessarily mean it will score highly on a desired set of criteria; it may be that one feature of the language makes it uniquely usable by a community. That feature may be that it is similar to a set of languages it was designed to replace, making it easy to migrate to. Or as another example, consider the old language FORTRAN IV. It was well respected by engineers of mathematical applications because of its compilers' efficiency and its wealth of mathematical primitives. But given it should embody desirable software engineering criteria such as *strong typing* and *structured programming* then it was quite obvious that it scored poorly. Thus languages like FORTRAN were either re-invented (hence environments such as 'MatLab') or they evolved to score higher against the new criteria (hence FORTRAN 77 with its structured control constructs).

In this paper I discuss the kinds of criteria against which an AI planning language might be judged, making a distinction between them depending on the purpose of the language. I apply them to version 1.2 of PDDL, and draw some conclusions for the future development of planning language standards.

Criteria for Evaluating Languages

The study of languages for machine as well as human consumption (ie ones that people have to manipulate or understand in some way) encompasses three aspects: syntax, semantics and pragmatics. A fundamental question about a language arises when considering these three aspects: *is it going to be used theoretically or is it going to be used generally by people to encode complex algorithms or knowledge?*

Theoretical formal languages: Considering theoretical languages, in computer science we have the Lambda Calculus, the Pi-Calculus, the Turing Machine, first order logics etc. They are often used to theorise about concepts (e.g. sequential or concurrent computation), or are used as the meaning domain for the semantical definition of practical languages. Considering the well-known languages which are used in theoretical research, the intrinsic criteria that underlie their success appear to be the following:

- (1) *simple, clear, precise syntax and well-researched semantics*

  For example, in Lambda Calculus the syntax is defined in a few BNF rules, with syntactic sugar being added when needed. The semantics have been studied in depth: for example, recursive functions in Lambda Calculus have a

[1] It is assumed in this paper that the languages considered are for domain models *input* to a planner, rather than 'plan' languages used to represent the output of a planner.

clear and precise operational semantics (using conversion rules and normal order reduction) and fixed point semantics. Research has showed that these two kinds of semantics co-inside.

- (2) *adequate expressiveness*

  Can the language adequately represent the range of its targeted application domains? For Lambda Calculus this is the domain of computable functions, and it is a well known (though unproven) conjecture that it is adequate for this.

- (3) *clear mechanisms for reasoning*

  Can a user (perhaps with tool support) reason with parts of a formula in the language? In Lambda Calculus one uses the conversion rules to transform one expression into another, equivalent expression.

Applied formal languages: Theoretical languages, however, tend to have *little or no* pragmatic features. At the other extreme are formal languages which have complex syntax which support many useful pragmatic features. For example we have Java in the field of programming, Z in formal specification of software, RML in requirements modelling (Greenspan *et al.* 1994) or $(ML)^2$ in knowledge-based systems (van Harmelen *et al.* 1996). Often, pragmatic features are present at the expense of clarity. For example, the amount of extra syntactic baggage employed by JAVA tends make it much less clear that the older, simpler PASCAL programming language. In AI planning there are a spectrum of languages between these two extremes. Some planning systems require complex practical-oriented features in their input languages, such as hierarchically structured objects and operators (McCluskey 2000), or Condition Types (Tate *et al.* 1994); some researchers need to use an input language that minimally models the dynamics of the domain, for example when exploring the theoretical complexity of planning (e.g. (Bylander 1991)).

I now consider some criteria that have been found useful for evaluating the pragmatic aspects of formal languages. A quite general framework for the evaluation of languages and their environments is Green's Cognitive Dimensions (Green 2000). This involves using a set of criteria as 'discussion points' to focus on the various dimensions of a language, and may result in an informal evaluation (Green admits his method in not analytic, and the dimensions are not mutually independent). He devised fourteen criteria which have been used to evaluate various types of language and environments, including theorem proving assistants, UML and programming languages. Although these criteria have been quite widely used, they have been successful for languages which are embedded in an environment rather than a language itself. Some of these criteria are aimed at the visual aspects of environments in which the language is embedded. Thus they would be better applied to a planning knowledge acquisition environment than the language used to represent the knowledge only. However, I have extracted and enhanced three criteria which are particularly related to the language itself, and have been used elsewhere in the literature:

- (4) *maintenance* (also referred to as *hidden dependencies* or *locality of change*)

  After changing one part of the notation, will this have any invisible knock-on effects on other parts? Do changes to a part of a model just have a local effect, or will they have global connotations? Can the model be easily and consistently updated to reflect changes? (from the viewpoint of maintenance, it is desirable that all changes have minimal global effects).

- (5) *closeness of mapping / customisation*

  How natural is the mapping between the domain and the model? how small is the 'semantic gap'? Is the language customisable in some sense so that it can fit in well with applications?

Since there is a whole range of assumptions involved in planning which may or may not hold in an application (for example to do with action duration, resources, closed world) it may be that the modelling language will have "variants" to deal with different assumptions. Related to this is the need to have 'hooks' in the language to allow extension: if the scope or depth of requirements of the domain are increased, can the formalism be likewise extended?

- (6) *error-proneness*:

  does the design of the language discourage errors, or are there any parts where it is hard to avoid errors? Is the construction of domain models error prone in a particular way?

Criteria (4) - (6) are analogous to those used to evaluate programming languages: (4) reflects the idea that languages should embody structures to promote loose coupling between sub-parts, and strong coherence. The 'object' in object-oriented programming scores highly in this respect, as implementations of object behaviour are insulated from other parts via the object interface. (5) reflects the dominance of 'high-level' languages - those that are more problem-oriented than machine oriented, and are equipped with user-defined structures for customisation. Finally, (3) has influenced programming language design in order to eliminate common errors; for example, languages which are *not* strongly typed are particularly prone to errors resulting from variable misuse and misspelling.

To investigate more criteria we use Van Harmelen et al's evaluation of $(ML)^2$, a formal KBS specification language, and hence relevant to AI planning languages. They use six criteria to evaluate this formal language used for formalising KADS expertise models (van Harmelen *et al.* 1996). Although objectiveness may be compromised when a group sets out their own criteria for evaluating their own product, the criteria they use are clearly worked out in response to considering the purpose of the language. They use criteria similar to those above (in particular (1), (4) and (6)), as well as the following:

- (7) *reusability*

  Can models or parts of models be easily reused to construct models for new domains?

- (8) *guidelines and tool support*

Is there a useful method to follow to build up a model, and are there tools to support this process?

With respect to the last point, in all areas in computer science involving some kind of non-trivial knowledge capture, methods have been developed to support this. For example in formal specification of software there is the B method(Schneider 2001), or in the acquisition of knowledge for KBS there is the KADS method (Wielinga *et al.* 1992), and some methods have also been developed for acquiring AI planning knowledge (Tate *et al.* 1998; McCluskey & Porteous 1997). The method will give a set of ordered steps to be carried out in order to capture and debug the domain model, thus guiding the knowledge engineer throughout the process. Ideally, the tools will be available in an integrated environment, and will support the steps in the method. Using the structure of the model language, the tools should be able to provide powerful support for statically validating, analysing and operationalising the model.

Finally, I draw on the guidelines for the design of domain model languages as recorded in the Knowledge Engineering for AI Planning Roadmap (McCluskey *et al.* 2003). This was written in the context of planning domain modelling, with the purpose of the language being to assist the process of knowledge acquisition and domain model validation. The criteria included several similar to those discussed above (in particular (1), (2), (5), (8)), and additionally the following:

- (9) *structure*

  It should provide mechanisms that allow complex actions, complex states and complex objects to be broken down into manageable and maintainable units. For example, the dynamic state of a planning application could be broken down into the dynamic state associated with each object. On this structure can then be hung ways of checking the model for internal consistency and completeness.

- (10) *support for operational aspects*

  The language's framework should include a set of properties and metrics which can be evaluated to assess a model's operationality and likely efficiency. It should be possible to predict whether the model can be translated to an efficient application, and what kind of planner should be used with the model.

To sum up, the criteria for practical formal languages are based around the idea that the structure of the language should support initial model acquisition and debugging, and subsequent model maintenance and re-use. Also, although criteria (1) - (3) are aimed specifically for theoretical languages they are often thought desirable for practical languages also.

Design Criteria for a Planning Language

What are the design criteria for an AI planning language? As mentioned above, it depends for what *purpose* the language is set, and a particular concern is whether the language is for theoretical or practical use. In the case of PDDL, this 'purpose' seems to have grown and changed as the language is used more widely. From the initial PDDL report(AIPS-98 Planning Competition Committee 1998), it appears that the language was designed to represent the syntax and semantics of domain models that were currently available to the authors, and that were used as input languages to many of the published planners of the time. Not all planners were expected to use all PDDL's features, and on the other hand planners were expected to have requirements that would mean a user extending PDDL in a controlled way. Its initial purpose, therefore, appears to have been as a communication language - a basic common denominator for planners of the STRIPS-tradition at the time so that (a) they could be compared in competition and (b) problem sets could be shared and planning algorithms independently validated. In this respect, as a communication language it has clearly been successful.

Nowadays the Planning Domain *Definition Language* is sometimes described as a 'modelling language', which has quite different ramifications than its originally expressed purpose. If its purpose is to support theoretical study, eg to help compare the capabilities of new planning algorithms, then it should be evaluated with respect to a restricted set of criteria such as (1) - (3). If its purpose (now) is to be a practical language, to help an engineer accurately and efficiently encode an application domain into a planning domain model then additionally it should be subject to evaluation by a range of the criteria such as (3) - (10).

Evaluation of PDDL with respect to stated criteria

In PDDL we have a family of languages to suit planners with different capabilities. The basic requirement in PDDL is ':strips' which indicates the underlying semantics of the language worlds are considered as sets of situations (states), where each state is specified by stating a list of all predicates that are true. Firstly, I evaluate the language using criteria (1) - (3) given above. I concentrate here on version 1.2 of the language, and remark on the extensions later.

Clear syntax and semantics: The syntax is clear and precisely defined within the manual, and parsing tools that embody this definition are publically available. The semantics of PDDL version 1.2, however, are informal and appear to be distributed among the manual itself, the pre-existing languages/systems that PDDL replaced (eg ucpop), PDDL's language processors, and the LISP interpreter. Although the fact that PDDL's syntax is LISP-like appears a superficial observation, the meaning of several of the primitive functions is given in terms of LISP functions. For example, the manual often relies on the reader using his intuition (p9: 'Hopefully, the semantics of these expressions is obvious'). As the language becomes more complex, then the natural language semantics are less obvious (for example, consider the meaning of domain axioms and their relationship with action definitions on page 13 of the manual). These extensions need to be defined precisely, as if two systems use these extensions, then they ought to do so in a uniform way, otherwise the standard is not preserved.

Adequate expressiveness: That PDDL a very expressive language for a range of planning applications has been shown by the range of problem domains used in competitions and in benchmark sets. Further, the ability to change some of the environmental assumptions is also present, although the semantics of some of these extensions is not clear.

Clear mechanisms for reasoning: A domain definition in PDDL is a 'model' in the sense that we have a representation that can be used to perform operations in the same manner that occur in the domain; and that there is a well-known operational semantics for constructs in the model. The declarative features of the notation - pre- and post-conditions, logic expressions, and named objects within the model which correspond directly to named objects in the domain, make reasoning about the notation feasible. However, problems to do with semantics, particularly to do with its extensions, restrict the success of this language with respect to the criterion.

Summary

In terms of the criteria for a language used for theoretical purposes, PDDL scores well on some aspects. There are problems with the lack of a clear semantics but these tend to be more to do with the non-basic parts such as the domain axioms. Also, the temporal and resource extensions of version 2.1 seem to have addressed the semantic issues more thoroughly (Fox & D.Long 2001).

PDDL: a modelling language?

Here I briefly evaluate PDDL using (3) - (10), the criteria reserved for languages aimed at practical application.

structure and error-proneness: PDDL has features such as ':timeless' - which allow the statement of static knowledge, and ':domain-axioms' which allow left-to-right rules that form invariants on situations. A domain definition is structured into components by Keywords e.g. :constants :actions etc. A special keyword is :requirements which tells a process which blend of PDDL features are used in the domain definition. Further, the manual devotes several pages to a hierarchical action notation; unfortunately, perhaps related to the fact that it was not subsequently used, version 2.1 of PDDL excludes this. On the negative side, whereas PDDL (v1.2) has features for hierarchically structuring actions, it does not have sufficient features for giving structure to objects or states. Further, the language lacks structures for setting up internal consistency criteria such as the completeness or validity of world states or actions.

maintenance and re-usability: PDDL's declarative form makes adding and changing operators a local task, and reusing operators in new domains feasible. The ':extends' feature allows a form of modularisation - one can import previously written components into a new model. However, no help is given in dealing with the natural dependency of actions on each other: the requirement that pre-conditions should be achievable by the execution of other actions or the initial state causes global interference and is the cause of many errors when defining domains.

guidelines and tool support: there are parsers, solution checkers and domain analysis tools available publically, but PDDL was not designed to be associated with a method for model building. This one point alone seems to make it currently ineffective as a practical 'modelling language' for complex applications.

closeness of mapping / customisation: Clearly PDDL's encodings shares the same 'high level' aspects as do propositional encodings in general. Also, one can pose domain axioms to model invariants in the domain. Reflecting domain *structure* (as mentioned above) by for example creating composite objects is not possible. Customisation does appear to be addressed in PDDL with features such as ':requirements' where fundamental assumptions about the model of the domain can be set.

support for operational aspects: The PDDL manual makes it clear that this area is not one that fits in with PDDL's aim - to model the physics of a domain. It does recommend a convention by which such extensions can be made in a controlled way, such that the model with the extensions stripped away will make sense to a pure PDDL interpreter.

Summary

For a language whose initial purpose was one of domain model communication, and which aspired to include only feature which capture dynamics, PDDL has in fact several features to help domain builders. These include HTN operators, domain axioms, modularisation through the 'extends' keyword etc. On the other hand, it fails to meet the criteria is in not being associated with a model building method, and in its lack of structure for objects, predicates and states. Structuring devices are present in several modelling languages (e.g. DDL.1 (Cesta & Oddi 1996; McCluskey & Porteous 1997)); these allow the state-space of objects to be modelled independently of the actions, and hence are useful in removing errors from action representations.

Conclusions

Both to help the Planning field mature, *and* to help engineers apply the technology, language conventions have to be achieved. The requirements of the future Semantic Web in particular will demand a common model for planning knowledge. This paper has argued that before conventions are devised there must first be an agreement on the purpose of a language, and secondly a set of criteria to be used to help form and develop the language.

Two broad purposes for an AI planning domain language were outlined - one as a theoretical device, to be used for exploring the properties of planning algorithms, and one as a practical language, to be used to help an engineer efficiently and accurately encode an application domain.

I performed an initial evaluation of PDDL with respect to the criteria formed from both purposes, with mixed results. The evaluation leads me to the following conclusions:

- in standardising a form of PDDL for theoretical purposes, more attention needs to be devoted to precisely defining its semantics, and that of any of its extensions;

- in standardising a form of PDDL for practical domain model building, then more structure, guidelines and tool support is required.

For the future, I feel that the community needs to settle on the purpose of PDDL, decide on the criteria that can be used to evaluate PDDL's quality, and perform a thorough evaluation using the language's most recent version. This will lead, I believe, to a sound path for its future development.

References

AIPS-98 Planning Competition Committee. 1998. PDDL - The Planning Domain Definition Language. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control.

Bylander, T. 1991. Complexity Results for Planning. In *Proc. IJCAI'91*.

Cesta, A., and Oddi, A. 1996. DDL.1: A Formal Description of a Constraint Representation Language for Physical Domains. In Ghallab, M., and Milani, A., eds., *New Directions in AI Planning*. IOS Press. 341–352.

Fox, M., and D.Long. 2001. PDDL2.1: An extension to PDDL for expressing temporal planning domains . In *Technical Report, Dept of Computer Science, University of Durham*.

Green, T. 2000. Instructions and descriptions: some cognitive aspects of programming and similar activities. In *Advanced Visual Interfaces*, 21–28.

Greenspan, S.; Mylopoulos, J.; and Borgida, A. 1994. On formal requirements modeling languages: RML revisited. In *Proceedings of the 16th International Conference on Software Engineering*, 135 – 148. IEEE Computer Science Press.

McCluskey, T. L., and Porteous, J. M. 1997. Engineering and Compiling Planning Domain Models to Promote Validity and Efficiency. *Artificial Intelligence* 95:1–65.

McCluskey, T. L.; Aler, R.; Borrajo, D.; Haslum, P.; Jarvis, P.; I.Refanidis; and Scholz, U. 2003. Knowledge Engineering for Planning Roadmap. *http://scom.hud.ac.uk/planet/*.

McCluskey, T. L. 2000. Object Transition Sequences: A New Form of Abstraction for HTN Planners. In *The Fifth International Conference on Artificial Intelligence Planning Systems*.

Schneider, S. 2001. *The B-Method: An Introduction*. Palgrave.

Tate, A.; Drabble, B.; and Levine, J. 1994. The Use of Condition Types to Restrict Search in an AI Planner. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*.

Tate, A.; Polyak, S. T.; and Jarvis, P. 1998. TF Method: An Initial Framework for Modelling and Analysing Planning Domains. Technical report, University of Edinburgh.

van Harmelen, F.; Aben, M.; Ruiz, F.; and van de Plassche, J. 1996. Evaluating a formal KBS specification language. *IEEE Expert* 11(1):56–62.

Wielinga, B. J.; Schrieber, A. T.; and Breuker, J. 1992. KADS - a modelling approach to knowledge engineering. *Knowledge Acquisition* 4(1):5 – 53.