Real-Time Planning for Video-Games: A Purpose for PDDL

Olivier Bartheye and Éric Jacopin
MACCLIA
CREC Saint-Cyr
Écoles de Coëtquidan
F-56381 GUER Cedex
{olivier.bartheye,eric.jacopin}@st-cyr.terre-net.defense.gouv.fr

Abstract

We intend this paper as a position paper on the missing features of PDDL as a domain definition language for real-time planning in the video-gaming area. After a very brief overview of PDDL, we discuss its quality, its adaptation to real-time issues and to the needs of the video-games domain. We present a few further issues from our experience developing real-time pddl-based planning for fast pace video-games.

Introduction

Could a newly developed planner today not accept some subset of the Planning Domain Description Language (PDDL)? The answer certainly is an obvious "No" if this planner aims at the International Planning Competition for which PDDL is mandatory since 1998 (AIPS 98 Planning Competition Committee 1998); and if not, why would this planner take the time to be PDDL-compliant? Because reusing PDDL, its features and semantics (Gerevini, Long, and Haslum 2009) and its planners, could just save you time and money.

Then what would you use PDDL for? Well, as the Planning Domain Description Language, you would expect PDDL to provide key features in the process of solving a planning problem. But what about video-games and, moreover, fast pace arcade games? Well, if you would love to have a planner for your next game, then PDDL might just be the choice for you. This position paper enumerates several features not available in PDDL which we believe are nevertheless useful for your Real-Time AI Planning needs for video-gaming.

If Planning is about plan construction, then there are features PDDL does not provide. In PDDL there little place for Hierarchical Task Networks, no way to declare procedural attachments to actions, no mean to express heuristics on the selection on goals or preconditions and the solution space is invisible: how many solutions do you want for your problem? The first one you find? Five? Would it be useful to keep on searching after one solution has been found?

However, Planning for fast pace video-games is not just about plan construction. For instance, Planning also has to do with plan execution, handling execution exceptions and failures of various kinds: plan construction failure (no plan exists for the problem, a plan exists but shall be available too late for execution, a plan is found but only part of it is valid for execution, . . .), action execution failures (preconditions totally or partially unfulfilled, impossibility to detect action execution success, no replacement action is available, . . .) and plan execution recovery (once an action failure has been successfully handled, how would you get back to the original plan). Planning for video-gaming also has to do with planning domain maintenance (various development versions for debugging and performance tuning) incremental loading of domain data (bonus or power-up actions should be available only when the corresponding game actions are available) and patches (new characters, new behaviours and new levels certainly imply new PDDL domains files).

We obviously admit that some parts of the previous enumerations might not be of direct concern for a Planning Domain Description Language; however, we strongly believe that PDDL should include well-defined features to *easily* extend the language to various domains and in particular to video-games.

The rest of this position paper is organized as follows. After a brief overview of the language, we discuss the evaluation of PDDL through McCluskey's criteria. We then give details on the meanings of real-time and report some of the needs of the game developers community through (Nareyek 2005). We further enumerate several features we believe are useful to apply PDDL to an industrial domain.

PDDL

We here give a very brief overview of PDDL while taking the time to point some limitations of PDDL. We thus invite the non-familiar-with-PDDL reader to refer to the official PDDL literature (AIPS 98 Planning Competition Committee 1998; Gerevini and Long 2005; Gerevini, Long, and Haslum 2009) which served as our PDDL reference for the details of this paper; we suggest as well the familiar-with-PDDL reader to skip the next subsection to the second subsection where presents a framework (McCluskey 2003) to evaluate various purposes of PDDL.

Overview

The Planning Domain Description Language (PDDL) is a planning problem specification language which has been widely available since the first International Planning Competition (AIPS 98 Planning Competition Committee 1998).

A planning problem is made of a set of actions, an initial state and a final state. The plan construction activity looks for organizing actions so that the initial state is transformed into the final state. As actions manipulate states, both actions and states are made of the same basic element: logical formulas which describe (interesting) properties of your domain.

For instance, in PDDL you could write

```
(coordinates oz-wizard 12-i 10-j)
```

to describe that the bidimensional `coordinates` of the `oz-wizard` are `12-i` and `10-j`. Numbers can only appear as parameters of the four usual operations (`+`, `*`, `/` and `-`) and of the usual binary comparison operators (`>`, `<`, `=`, `>=` and `<=`); for instance, in PDDL you could write

```
(>= (size oz-wizard-team) 4)
```

to describe that, for some reason, there must be at least `4` members of the `oz-wizard-team`. More complex properties can be expressed with quantifiers:

```
(forall (?w wizard)
   (and (member ?w oz-wizard-team)
        (>= (size oz-wizard-team) 4)
        (coordinates front-door 12-i 10-j)
        (coordinates ?w 12-i 10-j)))
```

could describe that at least `4` members of the `oz-wizard-team` must be at the `front-door` of some building for some reason; note that a variable is prefixed with a `?` and can be typed to restrict the interpretation of the formula: for instance, the above (universally quantified) formula states only `wizard`s must be at the `front-door`.

As PDDL descends from the LISP programming language (Winston and Horn 1989), it makes heavy use of parentheses; although it is expected that video-game developers will hardly be familiar with LISP, we argue that PDDL is *not* a programming language but a description language and that its LISP roots can safely be forgotten while keeping in mind that parentheses go by pairs: for each opening `(` there must be a closing `)`.

A state is a set of logical formulas and actions are made of two states: the first denotes the preconditions which must be achieved when the action is executed and the second, the effects, denotes the properties of your domain which are realized once the execution of the action is terminated. As an example, here is a PDDL action which trivially describes the bidimensional `move`ment of a character in a game;

```
(:action move
    :parameters
         (?c - character
          ?x1 - coord-i ?y1 - coord-j
          ?x2 - coord-i ?y2 - coord-j)
    :precondition
         (and (coordinates ?c ?x1 ?y1))
    :effect
         (and (coordinates ?c ?x2 ?y2)
         (not (coordinates ?c ?x1 ?y1))))
```

where keywords prefixed with a `:` (e.g. **`:effect`**) introduce a specific section of an action. Consequently,

```
(move oz-wizard 1-i 1-j 12-i 10-j)
```

describes the action of moving the `oz-wizard` from one location to another; this expression, called `<action-term>` in the very first version of PDDL (AIPS 98 Planning Competition Committee 1998), has neither been modified nor enriched in the subsequent versions of PDDL.

PDDL evolved to version 3 (Gerevini and Long 2005), with new notable expressive power at each step such as durative actions. As it is not our objective to detail the expressiveness of PDDL, we refer the reader to the literature for the complete story (Gerevini, Long, and Haslum 2009) while noting that the semantics of PDDL, i.e. how a language feature must be interpreted, are not always trivial (e.g. the use of Büchi automata to express the semantics of state trajectory constraints (Gerevini, Long, and Haslum 2009, pages 13–16)).

Quality

One of the key feature of PDDL is to provide a family of languages; (Gerevini and Long 2005, page 7) enumerates 16 distinct values for the keyword **`:requirements`**. Each requirement defines a subset of PDDL with various features; for instance, the **`:strips`** requirement is the default requirement when no requirement is specified while some requirements, such as **`:quantified-preconditions`** or **`:adl`** implies others. These subsets allows for separate tracks in the International Planning Competitions in order to compare planners with similar features because not all planners are expected to support all the features of PDDL. But although PDDL started with the purpose of comparing AI Planners and still serves this purpose, its languages-within-a-language feature rapidly pushed PDDL to the area of modelling languages.

As argued by McCluskey (McCluskey 2003) a language is not only a matter of features but also a matter of quality, specially when the evolution of the language blurred its initial purpose. Thus, (McCluskey 2003) presents several criteria to evaluate the quality of PDDL version 1.2 (AIPS 98 Planning Competition Committee 1998) for various purposes:

1. Simple, clear, precise syntax and well-researched semantics,
2. Adequate expressiveness,
3. Clear mechanisms for reasoning,
4. Maintenance (also referred to as hidden dependencies or locality of changes),
5. Closeness of mapping/customisation,
6. Error-proneness,
7. Reusability,
8. Guidelines and tool support,
9. Structure,
10. Support for operational aspect.

Let us give a short explanation for the following four criteria: Closeness questions the ease of customization of language so that “it can fit well in applications”; Error-proneness relates to the language design decisions in order to discourage errors, could for instance lead PDDL to first support several build configurations of game projects and consequently debugging features (West 2005) (e.g. spying of variables, formulas and actions just as in Prolog); Structure questions the "mechanisms that allow complex actions, complex states and complex objects to be broken down into manageable and maintainable units"; Support for operational aspect questions the predictability of the PDDL model of the domain: can it be translated efficiently? What kind of planner should be used with the model?

Criteria (1) to (3) are classically related to a computer science programming language and (Gerevini, Long, and Haslum 2009) addresses criteria (1) and (2) for today's version 3 of PDDL.

Criteria (4) to (10) are related to software engineering and thus are of concern to the video-game industry as a software industry. (AIPS 98 Planning Competition Committee 1998, page 14) admits that "it is often convenient to break the definition [of a domain] into pieces" and introduces the addendum construct and the :extends keyword to allow to add data to an existing domain thus addressing criteria (4), (7) and (9). These constructs, however, do not seem to survive the subsequent versions of PDDL, leaving the users with the development and maintenance of large planning domain files which simply take both time and memory to load. Really worse in our experience, is a large set of actions which must be loaded in one time as part of a planning problem definition file: if some of these actions reveal themselves to be unnecessary for a given phase of the game, they unnecessarily increase the search space and consequently increase the planning runtimes.

Real-Time

Performance is an issue for Real-Time software and so it is for the fast pace games of today, from arcade games to First Person Shooters. Let us first explain what we mean by real-time (Douglass 1999) in the video-games domain:

1. Real-time does not mean real fast but means the speed of the game: AI Planning does not lower the quality of entertainment/playability of the game,
2. Real-time means hard deadlines: late data is at best worthless data and at worst bad/wrong data. Missing a hard deadline constitutes a system failure of some kind: for instance, the death of a non-playable character or even worse, the death of the player's character,
3. Real-time means soft deadlines: late data, e.g. a plan of actions, may still be good data if execution coupled to exception handling can handle it,
4. Real-time means embedded: AI Planning exists inside a larger system, i.e. the game engine. The computation resources are shared among many important modules,
5. Real-time means interaction with sensors and actuators: it is the mean to acting in the game, to assess game situations for planning problem generation, to represent game level objectives for game trust.

Taken literally, none of these real-time issues are handled by PDDL which represents time in actions and then consequently in plans, but neither in the search space nor in the solution space: heuristics which could prune the search space (Botea, Müller, and Schaeffer 2003) are ignored and there is no way to express constraints on the number of solutions.

Embedded-ness and real-time interaction may not be of any concern to PDDL, but as AI Planning only is a module into a larger system, interfacing with this system or other modules is an essential feature; (Orkin 2006), for instance, uses procedural attachment to actions in order to connect to the Finite State Machines of the game.

Video-Games

Taken from (Nareyek 2005), here are some of the AI Planning requirements from the working group on Goal-Oriented Action Planning from the International Game Developers Association:

1. HTN/Hierarchical planning functionality is considered as very useful,
2. About the basic ingredients of what constitutes a planning problem and a planning domain: more complex data type than simple `TRUE/FALSE` for state variables should be possible, like numbers and object/links,
3. We also need the power of temporal qualifier for states. For now, temporal intervals seem to be the best choice for this,
4. We certainly have to pay more attention to the potential integration with scripting, the need to have a very first action determined as quickly as possible, and graded quality results for plan optimization.

These four requirements all directly relate to PDDL which uses various formulas (e.g. predicates, universal and existential) to describe states and actions. As Hierarchical Task Networks (HTN) are networks of actions, their use in a PDDL-based game entails a description syntax and semantic in PDDL. The reader is referred to (Armano, Cherchi, and Vargiu 2003; Botea, Müller, and Schaeffer 2003) for propositions on extending PDDL to hierarchical planning. As, previously mentioned (cf. the previous PDDL overview) numbers in PDDL can appear as function parameters and not as constant parameters for predicates; instead, string-like constants are used (see the very first PDDL sentence above). But complex data structures, for instance regarding durations and various temporal qualifiers for states, certainly are provided by PDDL.

However, PDDL does not provide a direct interface to "the outside" through links of any kinds (this includes the impossibility for procedural attachment) and, consequently, scripting. It might not be obvious that scripting for video-games is related to PDDL, but if Scheme (Abelson, Sussman, and Sussman 1997) was as popular as LUA in the video-game community (see (Millington and Funge 2009, Section 5.10) about Scheme-based AI scripting in games), both scripting and procedural attachment would be immediate in PDDL. As both compilation of Scheme and interfacing Scheme with procedural languages (such as C) are well-mastered topics (Queinnec 1997), various steps could be taken from Scheme to LUA (or Scheme to various virtual machines).

Further issues

We mentioned build configurations and debugging, but here are several other issues PDDL does not address:

1. Documentation, version control,
2. Patches (new objects, new levels, new behaviours, . . .),
3. Failure recovery (what must be done when no plan is found? When an action cannot be executed? When one cannot recognize whether an action has successfully been executed, . . .),
4. Exceptions (responsiveness to unexpected events).

C++ has header files for class declaration and source files for operation definitions and accordingly defines two file extensions: ".h" for header files and ".cpp" for source files. In PDDL, there are files for domains (types, predicates, functions, actions, . . .), for problems (constants and their possible types, **:init**ial and **goal** states, . . .), and for solutions; but PDDL does not define any (specific) file extension, leaving their management to version control systems. This indeed is a superficial requirement for PDDL; however, could you imagine an application with no file extension today? In the same superficial spirit, do you know of any language which does not include a syntax for comments? Well, although LISP comments are probably the most appropriate (in LISP, a comment begins with two "**;**"), PDDL does not even propose a syntactic form for comments. In the same spirit, no specific textual documentation is suggested although debugging and release configurations definitely concern modelling languages as well as programming languages; although clearly rudimentary, LISP provides documentation of LISP functions with a simple string and the **:documentation** and **:verbose** keywords to record and print this string (Steele 1990).

Patches with new game objects, levels, characters and behaviours might appear as only concerning game content, but does in fact concern PDDL: for instance, new behaviours means new actions, new game objects and new levels mean new formulas to describe them. The incremental loading of PDDL files would greatly ease the gathering of such domain descriptions; however, as noted earlier, both the addendum construct and the :extends keyword are longer PDDL members.

According to (AIPS 98 Planning Competition Committee 1998) PDDL only seems to be interested in solutions: failure is not an option for PDDL. But as we discuss with hard real-time deadlines, late data may just be wrong data: the returned solution simply is no longer a solution. Or some characters may just die waiting for a solution from the AI Planner. Consequently, and this is related to heuristics, we believe new kinds of solution files should be introduced: such files would contains plans but would just come from a library of plans and could be executed prior to AI Planning (taking cover is not only relevant for human soldiers), during search or after search when it fails or when execution fails. That is, if the game takes actions to keep a character alive, why not put these actions in a PDDL file? Then, we need to distinguish such files from solution files created by the search.

Priorities

If we could pull the strings for PDDL, we would certainly begin with the introduction of heuristics: ordering of the preconditions of actions and ordering of actions during search.

Secondly, we would allow for Scheme-based procedural attachment and would favour the development of a Scheme to LUA compiler.

Third, despite HTNs are interesting complex plan structures, we would rather look at introducing execution :pragmas in PDDL solutions files to describe interruptions and execution alternatives: if the AI planner spent some effort to propose a plan of actions, it might worth considering adding information to this plan to back up the failure of actions. An exception-like PDDL construct is what actually would suit us the most. Accordingly, we would find debugging directives to print information in a console during plan execution very useful.

Finally, as superficial as it may look, we would be very happy with file extensions. We currently have megabytes of PDDL files all with the same ".pddl" extension although these files have clearly not the same semantic (a domain file has nothing to do with a solution file).

Conclusion

The reader might reach this point with the belief that we are pessimistic about PDDL and its use in video-games. On the contrary, we believe that the AI Planning community is lucky to possess PDDL and we would have been unable to produce such an enumeration if PDDL had not helped us succeed in connecting an AI Planner to various video-games (Bartheye and Jacopin 2007; 2008; 2009a; 2009b): PDDL actually can be used for video-games. That's not the question. The question is how can we make it popular?

In a recent survey of the readers of the Game Developer Magazine and of attendees of the Game Developers Conference, game developers "cited ease of development as the top criteria than any other consideration" (Remo 2009) to pursue the development on a given platform. We consequently believe it is important to *ease* each step in the mastering of PDDL, not by providing new AI Planning features, but by first reinforcing the software engineering features of PDDL and second by introducing plan execution features into PDDL.

We intended this paper as a position paper, thought provocative, and we hope that at least, it shall provoke discussions and debates.

Acknowledgements

We wish to thank our anonymous reviewers for their helpful remarks and Michael Buro and Carlos Linarès for giving us the opportunity to improve this position paper.

References

Abelson, H.; Sussman, G. J.; and Sussman, J. 1997. *Structure and Interpretation of Computer Programs*. MIT Press.

AIPS 98 Planning Competition Committee. 1998. PDDL – the Planning Domain Definition Language (version 1.2).

Technical Report Tech Report CVC TR 98-003/DCS TR-1165, Yale Center for Computational Vision and Control.

Armano, G.; Cherchi, G.; and Vargiu, E. 2003. An extension to pddl for hierarchical planning. In *Proceedings of the International Workshop on PDDL*, 1–6.

Bartheye, O., and Jacopin, É. 2007. Planning as a software component : A report from the trenches. 26^{th} Workshop of the UK Planning and Scheduling Special Interest Group, Prague, CZ, Dec. 17-18.

Bartheye, O., and Jacopin, É. 2008. Connecting pddl-based off the shelf planners to an arcade game. ECAI Workshop on AI in Games, Patras, GR, Jul. 21.

Bartheye, O., and Jacopin, É. 2009a. A planning plug-in for virtual battle space 2: A report from the trenches. In *Proceedings of the Spring Simulation MultiConference*, 4. ACM Press.

Bartheye, O., and Jacopin, É. 2009b. A real-time ppdl-based planning component for video games. In *Proceedings of the* 5^{th} *AIIDE*, 130–136. AAAI Press.

Botea, A.; Müller, M.; and Schaeffer, J. 2003. Extending PDDL for hierarchical planning and topological abstraction. In *Proceedings of the International Workshop on PDDL*, 25–32.

Douglass, B. 1999. *Doing Hard Time*. Addison-Wesley.

Gerevini, A., and Long, D. 2005. BNF description of PDDL 3.0.

Gerevini, A.; Long, D.; and Haslum, P. 2009. Deterministic planning in the fifth international planning competition: PDDL 3 and experimental evaluation of the planners. *Artificial Intelligence* ***173*** 619–668.

McCluskey, L. 2003. PDDL: A language with a purpose? In *Proceedings of the International Workshop on PDDL*, 82–86.

Millington, I., and Funge, J. 2009. *Artificial Intelligence for Games*. Morgan Kaufmann.

Nareyek, A. 2005. (Group Coordinator), The 2005 AI Interface Standards Committee report – working group on Goal-Oriented Action Planning. International Game Developers Association.

Orkin, J. 2006. Three States and a Plan: The A.I. of F.E.A.R. In *Proceedings of the Game Developper Conference*, 17 pages.

Queinnec, C. 1997. *Lisp In Small Pieces*. Cambridge University Press.

Remo, C. 2009. State of development 2009. *Game Developer* ***17****(2)* 20–21.

Steele, G. 1990. *Common LISP – The Language (2nd Edition)*. Prentice Hall.

West, M. 2005. Debug and release. *Game Developer* ***12****(9)* 34–36.

Winston, P., and Horn, B. 1989. *Lisp 3rd Edition*. Addison-Wesley.