# Planning Modulo Theories: Extending the Planning Paradigm

**Peter Gregory**
School of Computing and Engineering
University of Huddersfield, UK
p.gregory@hud.ac.uk

**Derek Long** and **Maria Fox**
Department of Informatics
King's College London, UK
firstname.lastname@kcl.ac.uk

**J. Christopher Beck**
Mechanical and Industrial Engineering
University of Toronto, Canada
jcb@mie.utoronto.ca

**Abstract**

Considerable effort has been spent extending the scope of planning beyond propositional domains, for example to include time and numbers. Each of these extensions has been designed as a separate specific semantic enrichment of the underlying planning model, with its own syntax and customised integration into a planning algorithm. Inspired by work on SAT Modulo Theories (SMT) in the SAT community, we develop a modelling language and planner that treat arbitrary first order theories as parameters. We call this approach Planning Modulo Theories (PMT). We introduce a modular language to represent PMT problems and demonstrate its benefits over PDDL in terms of expressivity and compactness. We present a generalisation of the $h_{max}$ heuristic that allows our planner, PMTPlan, to automatically reason about arbitrary theories added as modules. Over several new and existing benchmarks, exploiting different theories, we show that PMTPlan can significantly out-perform an existing planner using PDDL models.

## 1 Introduction

Classical planning states are valuations over propositional variables and transitions between states are actions. Each action has a precondition formula, which is an arbitrary formula over the state variables that must be satisfied for the action to be applied, and effects which assign new values to a subset of the state variables. The remaining variables are assumed to remain unchanged (the STRIPS assumption).

In its simplicity, classical planning shares a lot with SAT, which is also concerned with valuations of propositional variables. An interesting development in the SAT community is recent interest in the extension of the propositional language of SAT via SAT Modulo Theories (SMT) (Nieuwenhuis, Oliveras, & Tinelli 2006). SMT is the following family of problems:

**Instance:** A first-order formula, $F$, including constant, predicate and function symbols, and a theory, $T$, defining the meanings of the symbols.

**Question:** Is $F$ satisfiable, subject to the interpretations of the symbols imposed by $T$?

SMTlib (Barrett, Stump, & Tinelli 2010) is a language developed for describing SMT problems. More than 18 solvers have been built to solve problems for several values of $T$, including Difference Logic, Linear Arithmetic, Arrays and Non-Linear Real-valued Functions (de Moura & Bjørner 2008; Nieuwenhuis, Oliveras, & Tinelli 2006; Dutertre & de Moura 2006).

We present a planning formalism for Planning Modulo Theories (PMT), allowing classical planning to be arbitrarily extended in a modular way, analogously to SMT. In PMT, new types (e.g. sets, vectors) can be added as modules and functions operating over those types can then be used in actions. This approach allows exploitation of core planning techniques, while decomposing problems into components that depend on specialised solvers. PMT problems could be solved by compilation into SMT, relying on SAT to support the core planning. Although Rintanen has reported some success in making SAT-planning competitive (Rintanen 2011), there is strong evidence to suggest that dedicated approaches to planning remain dominant. We present an implementation of a planner that solves PMT problems directly. We show how the $h_{max}$ heuristic can be extended to arbitrary new types, using a technique based on association of types with abstractions. We also describe how abstractions can be constructed automatically, allowing the automatic construction of an $h_{max}$ heuristic in PMT domains. We demonstrate some of the advantages PMT offers over PDDL modelling and present benchmark problems that cannot be efficiently represented or solved in PDDL, but which are naturally expressed in PMT. We then show the performance of our planner on these domains.

## 2 Planning Modulo Theories

We now formally define Planning Modulo Theories, which is analogous to the classical planning problem extended in the same way that SMT extends SAT. Recall that a (first order) theory is a set of first order sentences, usually constructed as the deductive closure of a set of axioms, that defines the behaviour of a (possibly infinite) collection of symbols: constants, functions and predicates. Examples are the theories of arithmetic and of set theory.

**Definition 2.1 — State** A *state* is a valuation over a finite set of variables, $V$, where each variable, $v \in V$, has a corresponding domain of possible values, $D_v$. The expression $s[v]$ denotes the value state $s$ assigns to variable $v$ and

$s[v = x]$ is the state that is identical to $s$ except that it assigns the value $x$ to variable $v$. The *state space* for $V$ is the set of all valuations over $V$.

Classically, all planning state variables have boolean domains. In SAS+ encodings, variables have finite domains, while, in metric planning, domains can be integers or real numbers. In PMT we allow arbitrary domains, such as sets, lists, multi-dimensional vectors over reals and so on. Use of types such as sets greatly increases the ease with which certain kinds of problems can be modelled (in some cases offering an exponential compression of the domain encoding), increasing the range and reach of potential planning applications. It is worth noting that richly expressive languages for expressing change over time are far from new (eg the Situation Calculus (McCarthy 1963)), but what we propose here actually constrains the model to make explicit the action structures that cause change in a way that makes them easier for a planner to reason about.

**Definition 2.2 — First order sentence over a state space $S$ modulo $T$** A *first order sentence over a state space $S$ modulo $T$* is a first order sentence over the variables of the state space, constant symbols, function symbols and predicate symbols, where $T$ is a theory defining the domains of the state space variables and interpretations for the constants, functions and predicates. A *term* over $S$ modulo $T$ is, similarly, an expression constructed using the symbols defined by $S$ and $T$.

The key to extending classical planning into PMT is to support first order sentences modulo theories in the preconditions of actions.

**Definition 2.3 — Action** An *action*, $A$, for a state space $S$ modulo $T$, is a state transition function, comprising:

- A first order sentence over $S$ modulo $T$, $Pre_A$, called the *precondition* of $A$.
- A list, $\mathit{Eff}_A$, of assignments to a subset of the state variables, each setting a variable to a value defined by a term over $S$ modulo $T$.

One of the most important ideas in early planning was the STRIPS assumption: that variables not explicitly affected by an action are assumed unchanged. This dramatically simplifies the description of actions compared with complete axiomatisations. We adopt the same assumption in our definition of action application:

**Definition 2.4 — Action application** An action, $A$, for state space $S$ modulo $T$, is applicable in state $s \in S$ if $T, s \models Pre_A$ (that is, the theory together with the valuation $s$ satisfies the precondition of $A$). Following application of $A$, $s$ is updated by the assignments in $\mathit{Eff}_A$ to the variables that they affect, leaving all other variables unchanged.

Definition 2.4 indicates the role that subsolvers play in PMT: to confirm that an action, $A$, can be applied in a state, $s$, it is necessary to check $T, s \models Pre_A$. Doing so involves interpreting the symbols from $T$ that appear in $Pre_A$. $T$ might, in practice, comprise the deductive closure of several theories (for example, $T$ might include sets and numbers, relying on arithmetic and set operations and interaction between them through the cardinality function), in which case multiple subsolvers might be required to interpret $Pre_A$, perhaps with interactions between them. We assume that logical connectives and structural equality are interpreted by $\models$.

We now complete the definition of the PMT problem.

**Definition 2.5 — Planning Modulo Theory** A *Planning Modulo $T$* problem is the tuple $\langle S, A, I, G\rangle$ where:

- $S$ is a state space in which all of the variable domains are defined in $T$;
- $A$ is a collection of actions for $S$ modulo $T$;
- $I$ is an initial valuation (the *initial state*) and
- $G$ is a first order sentence over $S$ modulo $T$ (the *goal*).

A *plan* is a sequence of actions, $\langle a_1, ..., a_n\rangle$ such that $a_i$ is applicable in state $s_{i-1}$ and $s_i$ is the result of applying $a_i$ to $s_{i-1}$, where $s_0 = I$ and $T, s_n \models G$.

PMT supports a lifted representation (as with PDDL) allowing parameterised families of variables and actions. Variable families must be parameterised with finite types to ensure that the set of variables remains finite. We note that it is straightforward to generalise the above definitions to include parallel plans and temporal planning problems and their plans (Fox & Long 2003; 2006).

The idea of extending classical planning to include interpreted predicates and functions was proposed by Geffner (2000) as Functional STRIPS, which is very similar in intention to our proposal. However, while Functional STRIPS is powerful enough to express computations on complex data structures and relationships between these data structures and the propositional part of a planning problem, the idea has not been pursued. Furthermore, the Functional STRIPS proposal does not make theories an explicit parameter of the encodings. In this paper we propose a different approach which shares some of the same ideas, but goes further in terms of the way theories are integrated as parameters to planning problems and we go on to show how these ideas can be supported in implementation. Dornhege *et al.* (2009a) proposed the extension of PDDL with modules that are linked to semantic attachments. Others have also used semantic attachments to extend the range of different kinds of planning technology, such as control-rule guided search (Bacchus & Kabanza 2000) and hierarchical planning (Currie & Tate 1991). The idea of exploiting semantic attachments is related to PMT, although the mechanisms supporting use of these attachments make the management of interacting theories potentially harder to control.

In Table 1 we note some of the theories that have been used in existing planning systems. The table shows several planners that can be seen as having been designed to solve planning problems modulo specific theories. In each case, the interpretation of the theory is managed via a specialised subsolver that can evaluate terms and predicates over terms within the theory, supporting the associated planner in solving problems that use the theory. This support varies from a simple consistency check for sentences in the theory (e.g. the STN in CRIKEY3) through to production of no-goods for the associated planner (e.g. LP-SAT).

Other authors have also observed the opportunities that SMT offers for planning with specific theories. In particu-

| Planning problems | Theory | Planner | Subsolver |
|---|---|---|---|
| Simple durative actions (PDDL2.1) | Difference Logic | Sapa (Do & Kambhampati 2003)<br>CRIKEY3 (Coles *et al.* 2008b) | STN solver |
| Continuous effects (PDDL+) | Linear Programs | COLIN (Coles *et al.* 2009) | LP solver |
| Metric resources | Presburger Arithmetic | Metric-FF (Hoffmann 2003)<br>LP-RPG (Coles *et al.* 2008a)<br>Filuta (Dvorak & Barták 2010)<br>LP-SAT (Wolfman & Weld 1999) | Interval bounded search<br>LP solver<br>Specialised solver<br>SAT modulo LP |
| Axioms | Datalog | FF variant (Thiébaux, Hoffmann, & Nebel 2005) | Specialised |
| 3d manufacturing problems | 3d geometry | IMACS (Gupta, Nau, & Regli 1998) | CAD-based |
| Physical system models | Non-linear functions | ASPEN (Chien *et al.* 2000)<br>SHOP (Nau *et al.* 1999)<br>Europa (Reddy *et al.* 2008) | External code attachments |
| Data products | Ontological theories | WSC planner (Hoffmann *et al.* 2008) | Inference over ontologies |
| Motion planning | 2d and 3d geometry | TFD/M and FF/M (Dornhege *et al.* 2009a; 2009b) | Semantic attachments |

Table 1: Some of the existing examples of planners that use sub-solvers to work with theories.

lar, Wolfman and Weld's LP-SAT (1999) uses a compilation approach to achieve solutions to planning with metric resources and TM-LPSAT (Shin & Davis 2005) uses a similar approach to convert PDDL+ problems into SAT modulo LP problems. TLP-GP (Maris & Régnier 2008) also compiles to SMT to handle temporal problems, using SAT modulo temporal problems (solved with STNs). In all of these examples the researchers have treated the planning problems as classical planning supplemented with specific fixed theories and have exploited the SMT compilation as a way to access the power of SMT to solve SAT modulo the specific theories involved. This is distinct from the idea of theories as *parameters* of the planning problem.

Research on SMT has focussed on theories that have application in the kinds of program and hardware verification tasks that are common application targets of the SAT technology, such as theories of arithmetic and of arrays. Similarly, planning can benefit from theories of spatial and kinematic reasoning to support integrated task and motion planning (Cambon, Alami, & Gravot 2009), data-operations for planning pipeline operations such as image processing (Golden 1998) or web-services (Hoffmann 2008), or of data-structure variables (arrays, sets, lists) to support, for example, bioinformatics applications such as DNA analysis. There are also many examples of systems that integrate programmed functions into planning models to support efficient modelling of complex subsystems, such as batteries, thermal behaviour and so on (Chien *et al.* 2000). PMT can subsume all of these examples and incorporate them into a single uniformly defined problem structure.

## 3 Describing PMT Problems

To support convenient description of PMT domains, we propose a modular variant of PDDL. Donhege *et al.* (2009a) also propose a modular extension of PDDL, but where their extension is based on attaching external libraries to the predicate and function terms they introduce, our variant focusses on the addition of theories for types. The idea of a modular language mirrors SMTlib (Barrett, Stump, & Tinelli 2010), allowing easy extension by addition of new theories to provide new types, with new interpreted constants and predicates and functions over these types. Module files contain declarations of the types of functions manipulating both new and existing types and are written in Module Definition Description Language (MDDL). The definitions of the variables and actions that model a planning domain are provided in core files, which are written in Core Domain Description Language (CDDL).

In CDDL and MDDL all constants and functions are typed. Two types are defined in the core language: `boolean` and `object`. The type `object` defines enumerated domains, such as the set of locations in a transportation domain. CDDL provides two functions: a polymorphic structural equality and a polymorphic assignment function. The return type of equality is `boolean`. Assignment takes a variable and an expression of the same type and updates the state by assigning the value of the expression (evaluated in the current state) to the variable.

Each module can define a (single) new type, constants of the type and functions. The functions can be over the new type and types defined in other modules. The set module in Figure 1 defines a polymorphic set type. Its functions can be used in action preconditions, the initial state and goals. The `cardinality` function returns a value of type `integer`, a type defined in a separate module. Sets can be constructed in the initial state with either the empty-set constant or the construct-set function. The parameter of the construct-set function is `?x+ - a'`, meaning that the function takes one *or more* constants of type `a'`. Since it is often useful to have variable numbers of arguments in function calls, we use the '+' syntax to indicate a variable number of constants of the final parameter type in the function prototype. Strictly, this means that such definitions define a family of functions with overloaded names, differentiated by their arities.

A CDDL file contains a header, defining names and values for enumerated types, the required modules and the functions that define the state variables for the domain (these can only use parameters from the enumerated types). Also defined in CDDL are the planning operators. A planning operator is defined in two parts: a precondition list and an effect list. Each precondition is a boolean function over the state variables and the list is interpreted as a conjunction. Each effect is an assignment of an expression to a variable. An example CDDL operator is shown in Figure 3, exploiting

```
(define (module set)
 (:type set of a')
 (:functions
 (construct-set ?x+ - a')                   - set of a'
 (empty-set)                                - set of a'
 (cardinality ?s - set of a')               - integer
 (member      ?s - set of a' ?x - a')       - boolean
 (subset      ?x - set of a' ?y - set of a') - boolean
 (union       ?x - set of a' ?y - set of a') - set of a'
 (intersect   ?x - set of a' ?y - set of a') - set of a'
 (difference  ?x - set of a' ?y - set of a') - set of a'
 (add-element ?s - set of a' ?x - a')       - set of a'
 (rem-element ?s - set of a' ?x - a')       - set of a'
))
```

Figure 1: The definition of a set module in MDDL.

```
(define (domain setdomain)
  (:types
    location locatable - object
    truck obj          - locatable)
  (:modules integer set)
  (:functions
    (at ?loc - location)        - set of package
    (loc-of-truck ?tru - truck) - location
    (in ?tru - truck)           - set of package
    (linked-to ?x - location)   - set of location)
```

Figure 2: Header of a CDDL domain file.

the interaction of integers, object fluents and sets. In comparison to a similar PDDL operator, this operator has fewer parameters because the location of the truck is determined functionally by an object fluent expression. In order to enforce a capacity constraint on each truck, it is only necessary to use the cardinality function. In PDDL it would be necessary to maintain a numeric fluent to count the items in a truck.

Variables can be assigned initial values or left undefined. The *undefined* value is considered to be a member of every type except `boolean`, may not be used in preconditions, but may be assigned as an effect. The reason that `boolean` has no undefined value is that we adopt the usual closed-world assumption and treat anything that is not true as false.

## 4 Comparing CDDL/MDDL and PDDL

We now examine an example domain that is natural to model using sets but which is difficult to model in PDDL, even with quantification and numbers. Consider a logistics-style domain where trucks are required to deliver packages between different locations. Packages are loaded into trucks one at a time, but all packages in a truck are unloaded at the same time (they are dump-trucks).

In the PMT framework, this unload action (along with relevant state functions) can be modelled as follows:

```
(:functions
  (pos ?tru - truck)    - location
  (at ?loc - location)       - set of package
  (in ?tru - truck)          - set of package
  (connects ?loc - location) - set of location)

(:action UNLOAD-TRUCK
  :parameters  (?t - truck)
  :precondition  (true)
  :effect
   ((at (pos ?)) := (union (at (pos ?t)) (in ?t))
    (in ?t) := (empty-set)))
```

The function `pos` defines object fluents each denoting the location of a truck. The `at` and `in` functions denote sets representing the packages at each location and in each truck, respectively, and the `connects` function returns the set of locations that can be reached from a given location. The unload action uses this representation to model the effects correctly: the set of packages at the location of the truck is updated to its union with the packages in the truck and there are then no packages left in the truck.

```
(:action load-truck
  :parameters (?p - package ?t - truck)
  :precondition ((member (at (loc-of ?t)) ?p)
                         (< (cardinality (in ?t)) 2))
  :effect
   ((loc-of ?t) := (rem-element (at (loc-of ?t)) ?p)
    (in ?t) := (add-element (in ?t) ?p)))
```

Figure 3: Sample operator using the set module, integer module and object fluents. The truck has unit capacity: assignment to `(in ?t)` directly affects the result returned by `(cardinality (in ?t))` in the next state.

In order to model this problem in standard PDDL, we propose the following alternatives:

1. The ADL fragment of the language can be used, quantifying over the packages in the effects of actions.
2. The propositional fragment of PDDL can be used with the powerset of the universal package enumerated as distinct named PDDL objects.

The ADL approach appears preferable to the propositional one, since the latter requires an exponential construction of sets of packages. However, in practice, most planners capable of handling ADL ground the powerset when confronted with the ADL version, effectively producing the same model as the propositional version.

Within the PMT framework, it is possible to specify goals based on function evaluation. For example, the goal:

```
(= (cardinality (at loc1)) (cardinality (at loc2)))
```

specifies that the numbers of packages at `loc1` and `loc2` must be equal. It is possible to encode this goal in PDDL only by changing the former model to include metric variables that count the numbers of items in the trucks and at locations. The counts must then be updated in a consistent way in load and unload actions (this is only possible because we know implicitly that all package sets are disjoint, so when a truck unloads the size of the set of objects at the location increases by the size of the set of objects in the truck). The goal can then be specified in the following way:

```
(= (cardinality-at loc1) (cardinality-at loc2))
```

where `cardinality-at` is a metric fluent counting packages at each location. Very few planners are currently capable of handling ADL and numbers.

The cardinalities goal can be modelled using the ADL subset of PDDL in combination with explicitly naming the subsets of packages that can occur. This is achieved by adding a predicate that models whether two (named) sets have the same cardinality. When a load or unload occurs, the effect determines which (named) subset is now present in the truck and the location. The cardinality of the sets at two locations can be then be checked using the cardinality testing predicate. This example illustrates how changing the forms of goal expressions can greatly complicate the PDDL model.

A very wide range of goals can be specified in the PMT framework. For example, using the set module, it is possible to reason about combinations of any sets using standard set operations. In the PDDL model, each new set must be managed explicitly throughout the actions. Consider the following PMT goal:

```
(= (cardinality (union (connects (position truck1))
                       (connects (position truck2))) 4)
```

specifying that the union of the sets of locations accessible to the two trucks has four elements. This goal is extremely challenging to write in the PDDL model, since the two sets of accessible locations might overlap. It can be captured by adding a new function to the domain (`accessible`, say) that explicitly defines, in the initial state, the size of the union of the sets of connected locations for each pair of locations. Then the goal can be written:

```
(exists ?l1 ?l2 - location (and (at truck1 ?l1)
      (at truck2 ?l2) (= (accessible ?l1 ?l2) 4)))
```

Adding this function will still not allow us to express goals that involve three or more trucks, or those that require the set of reachable locations to include a specific subset of values, or a host of other variations. In general, it is not possible to create a definitive PDDL representation of a PMT model because there is always a more expressive goal specification than can be captured in the PDDL domain model.

## 5 A PMT Planner

We have implemented a forward state-space planner, PMT-Plan, operating on PMT models directly. One of the challenges in building such a planner is that, since types and functions are user-defined through modules, the planner must handle arbitrary (and possibly infinite) types. This has very significant implications for the heuristic computation, as well as complicating other aspects of the typical planning process, such as grounding. At an implementation level, it is a design goal to make it as easy as possible for a user to implement a new module for use in PMTPlan.

The planner has two parts: the Core and the Modules, supporting the CDDL and MDDL elements of the domain descriptions respectively. The Core drives the search for a plan and accesses each of the Modules. The Modules are custom components which implement the new types and functions. The Core module provides the `boolean` and `object` types and implements assignment. It provides the following functions:

```
  assign :: variable -> expression -> state -> state
  =      :: expression -> expression -> boolean
```

Each of the Core and Module components implements the following interface:

```
  evaluate  :: expression -> state -> constant
  satisfies :: expression -> state -> boolean
```

The function `evaluate` returns the value of a given expression in a given state and `satisfies` determines whether or not a boolean expression is satisfied in a state (realising the first of the components of Definition 2.4). It might appear that `satisfies` is redundant since it can be implemented using `evaluate`, but we will highlight its role when we discuss heuristics. Each module implements `evaluate` and `satisfies` for the expressions it provides. For example, the set module `evaluate` implements the `union` function by calculating the actual union of two sets, using its own internal representation of the sets involved. The Core `evaluate` function implements equality, evaluation of variables in a state, handles `object` values and invokes calls on appropriate modules when evaluating module-defined functions.

The module designer must implement the functions that each module provides. A precondition expression is checked by a call to `Core.satisfies` and an effect is applied by calling `Core.evaluate` on the assigned expression and then setting the modified variable to the expression value in the next state. This is all that is needed to implement state progression, which is sufficient to support a simple breadth-first search through state space to find plans.

### 5.1 Abstractions and Heuristics

One of the most powerful techniques in modern planning is the use of relaxed problems to provide an estimate of plan length, providing a heuristic for A$^*$ or similar searches. A simple heuristic is $h_{max}$, which can be seen as the length of the shortest plan in the relaxed reachable state space (Haslum & Geffner 2000). The standard relaxation used to compute $h_{max}$, in propositional domains, is to ignore delete effects of actions. We now consider a slightly different interpretation of the computation of $h_{max}$ and use it to generalise the computation to arbitrary types.

As Haslum and Geffner (2000) observe, $h_{max}$ is equivalent to $h^1$, constructed by building a relaxed reachability analysis from the state being evaluated, $s$, until the goals are satisfied: the analysis constructs a sequence of states, starting from $s$, generating each successive state by simultaneously applying the add effects of all actions whose preconditions are satisfied in the preceding state, until the goals are satisfied. The length of the sequence is then the heuristic value of $s$. One way to see this process is that the value assigned to a variable by any update effect, regardless of whether it is true or false, is combined with the value of the variable in the preceding state by logical disjunction. The reason for using disjunction is that the relaxation must be monotonic: once a sentence is true in a state it must remain true in all subsequent states in the reachability analysis. In a classical propositional model, goals and preconditions are all conjunctions of positive literals, so monotonicity is achieved by ensuring that variables remain true once they have become true.

If goals or preconditions are arbitrary propositional sentences over the state variables then achieving monotonic behaviour requires a modification of this approach. In the reachability analysis, the propositional variables are considered to take values in a new domain, $\{F, T, TorF\}$. The initial state in a reachability analysis starts with each variable assigned either $T$ or $F$ according to its value (*true* or *false*) in the state being evaluated. Sentences are evaluated in this domain using a modified interpretation of the logical connectives. In particular: $\neg TorF = TorF$, $TorF \wedge T = TorF$, $TorF \wedge F = F$, $TorF \vee F = TorF$ etc. and a sentence is satisfied if its final value is $T$ or $TorF$. $T$, $F$ and $TorF$ are equivalent to $\{\text{true}\}$, $\{\text{false}\}$ and $\{\text{true}, \text{false}\}$, respectively, with the operator used to combine values (corresponding to disjunction in the original relaxation) being set union.

These examples serve to motivate the following definitions, which generalise the ideas. We introduce the notion of an abstraction of each value domain, such as $\{T, F, TorF\}$ discussed above. We also rely on an abstraction of the theory used in a PMT model: for the $\{T, F, TorF\}$ domain the extended definition of the logical connectives is the abstracted theory. Finally, we require an operator, which we call the *folding* operator, to combine prior abstract values with new ones during the updating of variables, which is the role played by disjunction in the delete relaxation and union for the domain $\{\{\text{true}\}, \{\text{false}\}, \{\text{true}, \text{false}\}\}$. The penultimate part of Definition 5.1 makes the abstraction a relaxation, while the last part ensures that the the folding operation provides the necessary monotonicity in the reachability analysis defined in Definition 5.2.

**Definition 5.1 — Domain Abstraction** Give a planning problem modulo $T$, $P = \langle S, A, I, G \rangle$, where $S$ is a set of valuations for variables $\langle v_1, ..., v_n \rangle$, a *domain abstraction of* $P$ is an abstracted state space, $\mathcal{A}(S) = \mathcal{A}(D_{v_1}) \times ... \times \mathcal{A}(D_{v_n})$, where:

- $\mathcal{A}(D_{v_i})$ is a type called the abstraction of $D_{v_i}$ and there is an associated function $\text{abstract} :: D_{v_i} \rightarrow \mathcal{A}(D_{v_i})$
- for $s \in S$, $\text{abstract}(s)$ is defined to be the result of applying abstract to each value assigned by $s$
- the *abstract initial state* is the valuation $\text{abstract}(I)$
- $\mathcal{A}(T)$ is an abstraction of $T$ that defines the behaviours of the functions, predicates and constants from $T$ interpreted over the types $\mathcal{A}(D_{v_1}), ..., \mathcal{A}(D_{v_n})$
- for any sentence, $\mathcal{S}$, and state, $s \in S$, such that $T, s \models \mathcal{S}$, $\mathcal{A}(T), \text{abstract}(s) \models \mathcal{S}$
- each $\mathcal{A}(D_{v_i})$ has an associated *folding* operator, $\oplus$, such that, for any $x, y \in \mathcal{A}(D_{v_i})$, any sentence $\mathcal{S}$ and any state $s \in \mathcal{A}(S)$, if $\mathcal{A}(T), s[v_i = x] \models \mathcal{S}$ or $\mathcal{A}(T), s[v_i = y] \models \mathcal{S}$ then $\mathcal{A}(T), s[v_i = x \oplus y] \models \mathcal{S}$.

**Definition 5.2 — Reachability Analysis** A *Reachability Analysis* for a domain abstraction of a planning problem modulo $T$ is a sequence of abstract states, $s_0, ..., s_k$, where $s_0$ is the abstract initial state and, for each $i = 1, ..., k$, $s_i = \text{apply}(\bigcup_{A s.t. s_{i-1}, \mathcal{A}(T) \models Pre_A} \mathit{Eff}_A, s_{i-1})$, where:

$$\text{apply}(es, s) = \text{doEach}(es, s, s)$$
$$\text{doEach}(\{e\} \cup es, s', s) = \text{doEach}(es, \text{doOne}(e, s', s), s)$$
$$\text{doOne}(v := x, s', s) = s'[v = s'[v] \oplus \text{evaluate}(x, s)]$$

MetricFF (Hoffmann 2003) illustrates how Definition 5.1 can be applied to the reals, $\mathbb{R}$, which are associated, in MetricFF, with the type of real intervals: $\{[a, b] | a, b \in \mathbb{R}, a \leq b\}$. The abstracted theory is used to check sentences that contain inequalities between reals by testing whether *any* values in the intervals being compared satisfy the comparisons. Intervals are folded by taking the smallest enclosing interval. MetricFF uses this abstraction to construct the reachable abstracted states, but then constructs a heuristic by building a relaxed plan, which is a more informed heuristic than $h_{max}$, although inadmissible. LPRPG (Coles *et al.* 2008a) uses a similar abstraction, but modifies the way in which effects are combined in Definition 5.2, using a Linear Program to calculate interval bounds from the constraints in the action preconditions and effects, while preserving the necessary monotonicity condition.

To implement domain abstractions in PMTPlan, each type must be linked to an abstract domain which conforms to the following interface:

```
evaluate :: expression -> state -> constant
satisfies:: expression -> state -> boolean
fold     :: constant -> constant -> constant
```

The link between a type and its abstraction also requires an implementation of the `abstract` function. It is because of the way that abstractions are handled that we separate `satisfies` and `evaluate`, since their implementations can differ in handling particular types.

## 5.2 Domain Abstractions

In PMTPlan we exploit several domain abstractions. The simplest is the *identity abstraction*, which leaves the space unchanged. We explain why this is useful shortly. A second abstraction is the *enumerated abstraction*, in which a type, $T$, is associated with the abstract type $\mathcal{P}(T)$, the powerset of $T$, and the folding operator is set union. The enumerated abstraction was described above for the boolean type, but can also be used for infinite types such as integers, enumerating the (finite) set of values reachable at each abstract state in a reachability analysis. The enumerated abstraction can work well when the goals are reachable after relatively few steps, but the cost of maintaining sets of reachable values in infinite types becomes prohibitive for long reachability analyses.

A third abstraction, which combats the problem of managing sets of reachable values in the enumerated abstraction, is the *finite abstraction* in which a special limit value is added to the powerset of a finite subset of values of the original type. We call the finite subset the *basis*. The limit value is used to represent any set that includes values outside the basis. Thus, natural numbers can be abstracted with the finite abstraction: $\{\{0\}, \{1\}, \{0, 1\}, \mathit{large}\}$, where *large* is used to represent any set containing values other than 0 or 1. One advantage of finite abstractions is that the choice of the basis can be made to compromise between informedness of the reachability analysis and the computational cost of constructing it. Finally, a *bounds abstraction* uses an abstract value space that represents bounds on the range of possible values of the base type: the interval abstraction for numbers used in MetricFF is one example, but it can be generalised to work with sets, multisets and other ordered infinite types.

PMTPlan actually uses two abstractions: one, the *plan-level abstraction*, for the space in which a plan is sought and another, the *heuristic abstraction*, for the reachability analysis. In problems with infinite types, the use of a plan-level finite abstraction can demonstrate unsolvability, aiding identification of dead end states. In general, we use the identity abstraction at the plan-level, only switching to a finite abstraction if the reachability analysis extends beyond a threshold size without yielding a value.

## 5.3 Automatic Domain Abstraction

Although the use of domain abstractions offers a general way to handle arbitrary new types in PMT, it relies on provision of appropriate abstract types (and folding operators) to accompany new types added to the modelling framework. We have devised a way to avoid this by automatically constructing finite abstractions. The approach we use is to sample the values in a type by probing the reachable state space around the initial state, until a set of values is constructed for each of the types that supports the goals. The probing is driven by selecting actions to maximise the number of new values visited. We maximise the log of the size of the new sets of values to avoid quickly growing subsets of infinite types dominating our probes. Once the goals have been reached (in the abstracted theory) we use the subsets of values constructed for each of the types as the basis of a finite abstraction for each of them. The probes need only be constructed once, at the start of planning, and the constructed enumerated abstractions then used throughout.

We also construct a heuristic that is often more informed, though inadmissible, by using the finite set of sampled values found by the probing technique to restrict reachable values assigned to variables in a reachability analysis using an enumerated abstraction. Restricting the reachable values to a set, $R$, changes the value returned by $\mathrm{doOne}$ in Definition 5.2 to be $\mathrm{doOne}(v := x, s', s) = s'[v = (s'[v] \oplus \mathrm{evaluate}(x, s)) \cap R]$. Using the restriction can cause some actions to fail to be applicable that are applicable in the unrestricted abstraction. States that appear to be dead-ends under this restricted abstraction are assigned a very large value, but kept in the search space, to avoid incompleteness. Another source of inadmissibility is that actions can be delayed in application compared with the unrestricted case. The restricted abstraction is often more informed, apparently because it reduces spurious interactions between values that are reachable along quite different trajectories, artificially supporting action applications.

## 6 Experiments

We now provide some analysis of the performance of PMTPlan. We use three problems, illustrating the use of sets and the power of our approach in handling domains with numbers. The problems are Jugs-and-Water (McDermott 2000), which we use to illustrate the power of our heuristic in handling number-based problems, the Dumper-truck domain we outlined in Section 4 and a new problem called the Storytellers domain.

Table 2 shows our performance on Jugs-and-Water problems, compared with MetricFF, as the number of jugs grows, both in terms of nodes evaluated and absolute time to solve the problem; empty entries were not solved within 30 minutes and 4Gb (memory is most often the limiting factor for the unsolved problems). In this problem we are using our automatically constructed enumerated abstractions with the identity as the base abstraction. The automatic enumeration works well in this domain, identifying the useful reachable values in the integer domain and allowing our $h_{max}$ to be more informed than the relaxed plan in the interval abstraction space used in MetricFF.

### 6.1 The Dump-trucks Domain

The Dump-trucks domain uses sets and numbers. The complications in representing different goals for the domain in PDDL were discussed in Section 4. For brevity we consider only a simple collection of problems, with just two locations and two trucks, initially one at each location, together with varying numbers of packages all, initially, at one location. The goals are:

```
   (> (cardinality (union (in T1) (in T2))) 5)
   (< (cardinality (in T1)) (cardinality (in T2)))
```

As can be seen in Table 3, PMTPlan scales better than MetricFF on these problems. It is also clear in these results that MetricFF can evaluate nodes far faster than PMTPlan and that the benefit of using the direct representation of sets depends on the size and structure of the problems being solved.

|  | PMTPlan |  | MetricFF |  |
|---|---|---|---|---|
| Jugs | Nodes | Time | Nodes | Time |
| 02 | 34 | 2.47 | 18 | 0.01 |
| 04 | 24 | 2.44 | 136 | 0.00 |
| 06 | 97 | 6.94 | 582 | 0.03 |
| 08 | 116 | 8.20 | 2516 | 0.19 |
| 10 | 198 | 10.89 | 10564 | 1.17 |
| 12 | 270 | 15.79 | 28740 | 4.79 |
| 14 | 484 | 26.58 | 73558 | 18.00 |
| 16 | 507 | 37.07 | 186206 | 64.51 |
| 18 | 323 | 36.26 |  |  |
| 20 | 529 | 58.70 |  |  |
| 22 | 568 | 91.41 |  |  |
| 24 | 524 | 104.10 |  |  |
| 26 | 1995 | 392.32 |  |  |
| 28 | 395 | 108.66 |  |  |
| 30 | 707 | 201.68 |  |  |

Table 2: Jugs and Water with increasing numbers of jugs.

|  | PMTPlan |  | MetricFF |  |
|---|---|---|---|---|
| Packages | Nodes | Time | Nodes | Time |
| 10 | 1247 | 11.44 | 54658 | 1.07 |
| 12 | 1855 | 16.78 | 84759 | 2.83 |
| 15 | 10933 | 52.60 | 271705 | 47.90 |
| 17 | 20247 | 156.83 | 551430 | 215.14 |
| 20 | 49414 | 1095.77 |  |  |

Table 3: Dump-trucks problems with increasing numbers of packages.

### 6.2 The Storytellers Domain

In the Storytellers domain a set of storytellers tell their stories to a collection of different audiences. The storytellers know different (possibly intersecting) sets of stories. The audiences begin having heard none of the stories. Entertaining an audience leaves it having heard all the stories a storyteller knows. A storyteller might tell stories an audience has already heard, adding nothing to the stories the audience knows. A PMT encoding of this domain is as follows:

```
(define (domain storytellers)
 (:types   storyteller audiences stories)
 (:modules   integer set)
 (:functions  (known ?t - storyteller) - set of stories
              (heard ?a - audience)    - set of stories
              (story-set)              - set of stories)

 (:action entertain
  :parameters (?t - storyteller ?a - audience)
  :precondition (true)
  :effect ((heard ?a) := (union (heard ?a) (known ?t)))))
```

Our goals require set equality, set cardinality and expressions involving the stories heard by the audiences collectively: any equivalent PDDL model must be able to express these goals. The number of stories could be large (at least up to 20). A propositional encoding of all story subsets is ruled out, since $2^{20}$ subset objects are required and $2^{40}$ facts to encode the subset relation. Therefore, some other encoding must be used. Another problem in using PDDL arises from the fact that numbers and quantification need to be used in concert. In order to perform set-union in PDDL, it is natural to use quantification. Reasoning about the cardinalities of sets is difficult using this model, as counters cannot be incremented within a PDDL `forall` structure, so it is impossible to update the size of a set in a single action. With neither a purely propositional model nor the ability to update numbers in iterative structures, dummy actions are required

| | PMTPlan | | MetricFF | |
|---|---|---|---|---|
| Stories | Nodes | Time | Nodes | Time |
| 10 | 20 | 0.67 | 18 | 0.00 |
| 12 | 11 | 0.57 | 26 | 0.01 |
| 14 | 8 | 0.60 | 33 | 0.01 |
| 16 | 14 | 0.68 | 59 | 0.02 |
| 18 | 28 | 0.68 | 32 | 0.00 |
| 20 | 22 | 0.68 | 105 | 0.02 |
| 30 | 26 | 0.81 | 53 | 0.01 |
| 40 | 21 | 0.83 | 2218 | 0.17 |
| 50 | 23 | 0.83 | 46872 | 6.48 |
| 60 | 34 | 0.99 | 701345 | 208.40 |
| 70 | 38 | 0.98 | | |
| 80 | 31 | 0.97 | | |
| 90 | 22 | 0.85 | | |
| 100 | 25 | 0.82 | | |

Table 4: Storytellers Saturate Problems

to emulate the desired behaviour. Thus, we arrive at the following PDDL model:

```
(define (domain storytellers)
  (:requirements :fluents :adl)
  (:types storyteller audience - person story)
  (:predicates  (knows  ?t - storyteller ?s - story)
                (heard  ?a - audience ?s - story)
         (entertaining ?t - storyteller ?a - audience)
         (heard2  ?s - story))
  (:functions  (num-stories ?a - audience))

(:action start
 :parameters (?t - storyteller ?a - audience)
 :precondition
   (not (exists (?aa - audience ?tt - storyteller)
                   (entertaining ?tt ?aa)))
 :effect (entertaining ?t ?a))

(:action switch
 :parameters
  (?ot ?nt - storyteller ?oa ?na - audience)
 :precondition
  (and  (not (exists (?s - story)
                 (and (knows ?ot ?s)
                      (not (heard ?oa ?s)))))
         (entertaining ?ot ?oa))
 :effect (and  (not (entertaining ?ot ?oa))
               (entertaining ?nt ?na)))

(:action entertain
  :parameters (?t - storyteller ?a - audience ?s - story)
  :precondition (and  (not (heard ?a ?s))
                      (knows ?t ?s)
                      (entertaining ?t ?a))
  :effect (and  (heard  ?a ?sa)
                (heard2 ?s)
                (increase (num-stories ?a) 1)))
```

In this model, storytellers tell stories one-by-one, so the cardinalities of the sets of stories heard by audiences can be incremented as each story is told. Once committed, a storyteller must tell all possible stories unknown to the audience. This is achieved by the first precondition of the `switch` action that switches which storyteller is entertaining an audience: the condition specifies that a new storyteller-audience pair can be selected only if no stories known to the current storyteller have not been heard by the audience.

We compare PMTPlan using the PMT model described above and MetricFF using the PDDL model described above, with counting and dummy actions. We consider two different goals with the same basic problems: saturation and equality. Both of these goals require the audiences to hear at least half of the stories, but the saturation problems require all of the stories to have been heard by at least one of the audiences, while the equality problems require that the audiences all hear the same stories. We use problems with two audiences and five storytellers and vary the number of stories. Tables 4 and 5 show the performances of the planners.

| | PMTPlan | | MetricFF | |
|---|---|---|---|---|
| Stories | Nodes | Time | Nodes | Time |
| 5 | 13 | 0.71 | 14 | 0.01 |
| 6 | 12 | 0.64 | 14 | 0.13 |
| 7 | 4 | 0.55 | 33 | 2.56 |
| 8 | 8 | 0.70 | 24 | 48.04 |
| 9 | 4 | 0.62 | | |
| 10 | 4 | 0.57 | | |
| 12 | 2 | 0.61 | | |
| 14 | 11 | 0.72 | | |
| 16 | 4 | 0.66 | | |
| 18 | 4 | 0.61 | | |
| 20 | 4 | 0.59 | | |

Table 5: Storytellers Equality Problems

MetricFF performs better on the saturate instances than it does on the equality instances because it can avoid explicitly dealing with sets. The audiences accumulate stories monotonically, so the problem only really concerns the simple counter to track whether all of the stories have been heard by all the audiences. When reasoning about whether two audiences have heard the same stories it is impossible to avoid explicit reasoning over sets using the model shown above. The data shows that the PDDL approach is infeasible for larger instances, while PMTPlan performs equally well across these problems, both saturate and equality (the plans are very similar for all of these problem instances because the problem encoding deals with the sets of stories directly).

## 7 Conclusions

As planning research becomes more relevant to application, demands on expressive power in the modelling language grow. SAT has responded to the same problem by allowing extension with theories in SMT: PMT can play an analogous role in planning. We have discussed how PMT problems can be encoded in a language similar to PDDL and considered what advantages modelling for PMT offers over modelling in PDDL. We have shown how a planner can be constructed, using interpretation modules for the theories exploited in a PMT model. More significantly, we have also shown how the $h_{max}$ heuristic can be generalised to these theories and explained how abstractions that support the heuristic can be automatically derived using the supporting modules. Our planner, PMTPlan, implements a search guided by this heuristic and we have demonstrated its effectiveness on some benchmark problems that exploit sets.

We recognise the need to develop a formal semantics for PMT: this is future work. However, we note that PMT demands a much stronger type-theory than PDDL and that an important aspect of the semantics will be to differentiate between state-dependent (e.g. connectedness) and state-independent functions (e.g. arithmetic functions).

We are already engaged in developing additional modules: we have built domains based on multisets and geometric reasoning (with path planning) and intend to explore the effectiveness of the automatic heuristic generation in a wider range of problems and domains. Planning Modulo Theories offers new ways to integrate planning and constraint-solving and we intend to explore how no-goods can be learned from subsolvers and fed back to the core, analogously to the links between subsolvers and a central SAT-solver in SMT such as in LP-SAT (Wolfman & Weld 1999).

## References

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

Barrett, C.; Stump, A.; and Tinelli, C. 2010. The SMT-LIB Standard: Version 2.0: http://combination.cs.uiowa.edu/smtlib/.

Cambon, S.; Alami, R.; and Gravot, F. 2009. A Hybrid Approach to Intricate Motion, Manipulation and Task Planning. *I. J. Robotic Res.* 28(1):104–126.

Chien, S. A.; Knight, R.; Stechert, A.; Sherwood, R.; and Rabideau, G. 2000. Using iterative repair to improve the responsiveness of planning and scheduling. In *Proc. Int. Conf. AI Planning and Scheduling (AIPS)*, 300–307.

Coles, A. I.; Fox, M.; Long, D.; and Smith, A. J. 2008a. A Hybrid Relaxed Planning Graph-LP Heuristic for Numeric Planning Domains. In *Proc. 18th Int. Conf. on Automated Planning and Scheduling (ICAPS )*.

Coles, A. I.; Fox, M.; Long, D.; and Smith, A. J. 2008b. Planning with problems requiring temporal coordination. In *Proc. 23rd AAAI Conf. on Artificial Intelligence*.

Coles, A. J.; Coles, A. I.; Fox, M.; and Long, D. 2009. Temporal Planning in Domains with Linear Processes. In *Proc. 21st Int. Joint Conf. on AI (IJCAI)*.

Currie, K., and Tate, A. 1991. O-Plan: The open Planning Architecture. *Artif. Intell.* 52(1):49–86.

de Moura, L. M., and Bjørner, N. 2008. Z3: An Efficient SMT Solver. In *Conf. on Tools and Alg. for the Construction and Analysis of Systems (TACAS)*.

Do, M. B., and Kambhampati, S. 2003. Sapa: A Multi-objective Metric Temporal Planner. *J. Art. Int. Res. (JAIR)* 20:155–194.

Dornhege, C.; Eyerich, P.; Keller, T.; Trüg, S.; Brenner, M.; and Nebel, B. 2009a. Semantic Attachments for Domain-Independent Planning Systems. In *Proc. 19th Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 114–121.

Dornhege, C.; Gissler, M.; Teschner, M.; and Nebel, B. 2009b. Integrating Symbolic and Geometric Planning for Mobile Manipulation. In *IEEE Int. Workshop on Safety, Security and Rescue Robotics (SSRR)*.

Dutertre, B., and de Moura, L. M. 2006. A Fast Linear-Arithmetic Solver for DPLL(T). In *Proc. 18th Int. Conf. Computer Aided Verification (CAV)*, 81–94.

Dvorak, F., and Barták, R. 2010. Integrating Time and Resources into Planning. In *Proc. 22nd Int. Conf. on Tools with Artificial Intelligence (ICTAI)*, 71–78.

Fox, M., and Long, D. 2003. PDDL2.1: An extension of PDDL for expressing temporal planning domains. *J. Art. Int. Res. (JAIR)* 20:61–124.

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

Geffner, H. 2000. Functional Strips: a more flexible language for planning and problem solving. In Minker, J., ed., *Logic-Based Artificial Intelligence*. Kluwer. chapter 9, 187–212.

Golden, K. 1998. Leap Before You Look: Information Gathering in the PUCCINI Planner. In *Proc. Int. Conf. on AI Planning and Scheduling (AIPS)*, 70–77.

Gupta, S. K.; Nau, D. S.; and Regli, W. C. 1998. IMACS: A case study in real-world planning. *IEEE Expert and Intelligent Systems* 13(3):49–60.

Haslum, P., and Geffner, H. 2000. Admissible Heuristics for Optimal Planning. In *Proc. Int. Conf. on AI Planning and Scheduling (AIPS)*, 140–149. AAAI Press.

Hoffmann, J.; Weber, I.; Scicluna, J.; Kaczmarek, T.; and Ankolekar, A. 2008. Combining Scalability and Expressivity in the Automatic Composition of Semantic Web Services. In *Proc. 8th Int. Conf. on Web Engineering (ICWE)*.

Hoffmann, J. 2003. The Metric-FF Planning System: Translating "Ignoring Delete Lists" to Numeric State Variables. *J. Art. Int. Res. (JAIR)* 20:291–341.

Hoffmann, J. 2008. Towards Efficient Belief Update for Planning-Based Web Service Composition. In *Proc. 18th European Conf. on AI (ECAI)*, 558–562.

Maris, F., and Régnier, P. 2008. TLP-GP: New Results on Temporally-Expressive Planning Benchmarks. In *Proc. 20th IEEE Int. Conf. on Tools with AI (ICTAI)*.

McCarthy, J. 1963. Situations, Actions and Causal Laws. Technical report, Stanford University. Reprinted in *Semantic Information Processing* (M. Minksy ed.), MIT Press, 1968, pp 410–417.

McDermott, D. 2000. The 1998 AI Planning Systems Competition. *AI Magazine* 2(2):35–55.

Nau, D.; Cao, Y.; Lotem, A.; and Muñoz-Avila, H. 1999. SHOP: Simple hierarchical orderd planner. In *Proc. Int. Joint Conf. on Artificial Intelligence (IJCAI)*.

Nieuwenhuis, R.; Oliveras, A.; and Tinelli, C. 2006. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(*T*). *J. ACM* 53(6):937–977.

Reddy, S. Y.; Iatauro, M. J.; Kürklü, E.; Boyce, M. E.; Frank, J. D.; and Jónsson, A. K. 2008. Planning and monitoring solar array operations on the ISS. In *Proc. Scheduling and Planning App. Workshop (SPARK), ICAPS*.

Rintanen, J. 2011. Heuristics for Planning with SAT and Expressive Action Definitions. In *Proc. of 21st Int. Conf. on Automated Planning and Scheduling (ICAPS)*.

Shin, J., and Davis, E. 2005. Processes and Continuous Change in a SAT-based Planner. *Art. Int. (AIJ)* 166:194–253.

Thiébaux, S.; Hoffmann, J.; and Nebel, B. 2005. In defense of PDDL axioms. *Art. Int. (AIJ)* 168(1-2):38–69.

Wolfman, S., and Weld, D. 1999. The LPSAT System and its Application to Resource Planning. In *Proc. 16th Int. Joint Conf. on Artificial Intelligence (IJCAI)*.