Planning with Complex Data Types in PDDL

Mojtaba Elahi, Jussi Rintanen

Aalto University
mojtaba.elahi@aalto.fi, jussi.rintanen@aalto.fi

Abstract

Practically all of the planning research is limited to states represented in terms of Boolean and numeric state variables. Many practical problems, for example, planning inside complex software systems, require far more complex data types, and even real-world planning in many cases requires concepts such as sets of objects, which are not convenient to express in modeling languages with scalar types only.

In this work, we investigate a modeling language for complex software systems, which supports complex data types such as sets, arrays, records, and unions. We give a reduction of a broad range of complex data types and their operations to Boolean logic, and then map this representation further to PDDL to be used with domain-independent PDDL planners. We evaluate the practicality of this approach, and provide solutions to some of the issues that arise in the PDDL translation.

Introduction

PDDL is the leading specification language used by the AI planning community in expressing (classical) planning problems and solving them by domain-independent planners. Most of the planners for the classical planning problem (with a unique initial state and actions that are deterministic) implement Boolean state variables only. Some additionally support numerical state variables, integers or reals. While this is theoretically sufficient to express a broad class of complex planning problems, the practical limits of PDDL are sometimes encountered. This is particularly evident when PDDL specification are not written by hand, and need to be generated by programs written in conventional programming languages. There are numerous examples of this both in standard PDDL benchmark problem sets as well as in the planning literature. PDDL is in these cases used an intermediate language to interface with domain-dependent planners.

The idea of more powerful specification language, allowing the easier expression of complex planning problems, and to reduce the need for ad hoc problem generators, is not new. A prominent example is *Functional STRIPS* (Geffner 2000). If PDDL's predicates $P(t_1, \ldots, t_n)$ are viewed as *arrays* as in programming languages, PDDL limits to arrays indexed by object names or by actions' parameter variables only, and only Boolean values as elements. Functional STRIPS goes further, allowing more complex array indexing, including with object-valued expressions and nesting of array indexing. Specifications in this kind of language can often outperform PDDL in elegance and succinctness.

The goal of this work is, similarly to Functional STRIPS, to increase the expressivity of specification languages. We believe that a planning specification language should support a broad collection of different data types, including scalars like Booleans, numeric types, and enumerated types, as well as compound types such as records, unions, arrays, sets and lists. This has been motivated by our work on using existing planning technology in the creation of intelligent software systems, which handle complex structured and relational data, as found in almost any software application.

In this work, we will first present an expressive language that supports several complex data types, and then we will give reductions of that language to an intermediate Boolean representation. This representation can be used as a basis of implementations in different types of intermediate and lower level modeling languages.

As an obvious language to reduce the Booleanized representation to is PDDL (Ghallab et al. 1998), due to the existence of dozens of implementations of PDDL in scalable, robust planners. The front-end of our planning system uses the extended specification language that supports complex datatypes. The expressions in this language, and the actions based on them, are reduced to a purely Boolean representation. The final step is the generation of PDDL, so that existing planners can be used for finding plans.

Intuitively it is clear that reducing complex data types to PDDL is possible, but it is not clear how practical those reductions are, and how efficiently existing planners for PDDL can solve the resulting PDDL specifications. Many existing planners support a large fragment of PDDL, but may translate some parts of PDDL representations to specific normal forms, and these normal form translations may increase the size of the representations. For instance, planner backends may require actions' preconditions to be conjunctions of (positive) literals. For this reason, to support many state-of-the-art planners better, we develop techniques to further process the PDDL to forms that are better digestible by existing planners.

In the experimental part of the work, we try out the resulting planner front-end with different state-of-the-art PDDL

Notation	Data type
$bool$	Boolean
$n..m$	bounded integer
$\{c_1, \ldots, c_n\}$	enumerated type with items $c_1, \ldots, c_n$
$\{t\}$	set of elements of type t
$t_v\{t_k\}$	associative array with index type of t_k and value type of t_v
$\langle t_1, \ldots, t_n \rangle$	n-tuple with elements of types $t_1, \ldots, t_n$
$\{f_1 : t_1, \ldots, f_n : t_n\}$	record with n fields
$[u_1 : t_1, \ldots, u_n : t_n]$	union with n components

Table 1: Data types

Data type	Operations
Boolean	$\neg, \wedge, \vee$
bounded integer	$+, -, \times, \div, <, \leqslant, =, \neq, \geqslant, >$
enum	$=$
set	$\in, \subseteq, \cup, \cap, \setminus$
array	index access
tuple	element access
record	field access
union	component access

Table 2: Supported operations for each data type

Data type	Representation
bool	Boolean variable
$n..m$	vector of $m - n + 1$ Boolean variables
$\{c_1, \ldots, c_n\}$	vector of n Boolean variables
$\langle t_1, \ldots, t_n \rangle$	n-tuple of representations of $t_1, \ldots, t_n$
$\{t\}$	vector of $\mathrm{dom}(t)$ Boolean variables
$t_v\{t_k\}$	vector of $\mathrm{dom}(t_k)$ representations of t_v

Table 3: Representation of values

planners as back-ends. We demonstrate their ability to solve complex problems that would be tedious and unintuitive to express in PDDL.

The experiments help identify bottle-necks in existing planners, which could aid in developing existing planning technology to better handle more complex problem specifications.

Target Planning Model

We define a planning problem by a quadruple of $P = \langle \mathcal{V}, s_0, g, \mathcal{A} \rangle$, where $\mathcal{V}$ is the set of state variables, s_0 defines the initial state by specifying the values of all state variables in $\mathcal{V}$, g is a Boolean formula describing the set of goal states (i.e., a state s is a goal state iff g holds in s), and $\mathcal{A}$ denotes the set of actions. An action $a \in \mathcal{A}$ is defined by a triple of $a = \langle args_a, pre_a, eff_a \rangle$. $args_a$ specifies the parameters of the action. A parameter is a variable defined in the scope of the action. pre_a is the precondition of the action, which is a Boolean formula specifying the applicability of the action in a state. eff_a is a set of conditional assignments defining how a state will be modified after applying the action. A conditional assignment is a pair $\langle cond, r := e \rangle$, where $cond$ is a Boolean formula defining the condition and $r := e$ is the assignment that assign the value of e to the reference variable r after executing action a if the condition $cond$ holds.

Theoretically, variables of $\mathcal{V}$ could be of any type, but in this work we limit to Boolean state variables, which is the most commonly used type in planners that support PDDL.

Extended Language

The goal of our extended language is to support more complex data types and standard operations on them. The data types we are covering are described in Table 1.

Using standard operations over the variables of these data types makes it possible to compactly formulate arbitrary nested expressions to define complex actions and goal condition. The operations we support in the extended language are listed in Table 2.

The intermediate representations of values of all supported data types are listed in Table 3 (the $\mathrm{dom}(t)$ function used in this table is defined in Table 7). Expressions that represent values of these data types have the same structure, except that what is the atomic part of the representation will be replaced by general propositional formula. For example, the Booleanized representation of numeric expressions of type $n..m$ is as $m - n + 1$ element vectors of propositional formulas.

Reduction to Boolean Representation

Although versions of PDDL support numeric and object fluents (Kovacs 2011) (which are similar to integer and enumerated data types), the lack of widespread support of these features by planners made us choose the version of PDDL as our target language that only supports Boolean state variables. Consequently, having only Boolean state variables means we need to represent state variables of different data types with Boolean variables and translate all of their expressions into Boolean formulas.

Scalar Data Types

Representation of Boolean state variables as individual Boolean variables is trivial. Other variables need to be represented by collections of Booleans. Here we only consider a *unary* representation for bounded integers and Enums. In other words, we represent a bounded integer $n..m$ as a vector of $m-n+1$ Boolean variables; also, an Enum type variable with n possible values can be represented by n Boolean variables. In either of these two examples, exactly one Boolean variable in the collection should be true at a time. A more compact representation would encode an integer value with the more standard representation with only a logarithmic number of bits.[1]

[1]This latter representation would, in some cases, be preferable. However, experience with constraint-solving shows that the unary representation can be more efficient when the value range is narrow.

Notation	Explanation	Example
c	constant value	3
v	variable	
e	expression	$(v_1$ add $2)$ eq v_2
ϕ	Boolean formula	$(\top \wedge v_1) \vee v_2$
$[\![e]\!]^t$	semantic of e, and e is value of type t	$[\![v_1$ add $2]\!]^{0..7}$
$\langle e_i \rangle_{i=1}^n$	$\langle e_1, \ldots, e_n \rangle$	
$\langle e_1, \ldots, e_n \rangle .i$	picking the i-th element (i.e., e_i)	
$\lvert \langle e_1, \ldots, e_n \rangle \rvert$	size of the tuple (i.e., n)	

Table 4: Notational conventions

More generally, we translate a *value* of a scalar type with a collection of *Boolean formulas*. For example, we translate a value of a bounded integer $n..m$ to $m-n+1$ Boolean formulas; each one of those formulas represents the truth value of the corresponding integer value. The translation of scalar expressions is shown in Table 5. We have adopted the notational conventions described in Table 4 to define the translations.

To prevent overflow in the arithmetic operations, we always cast the type of the result value to a bounded integer with the minimum range that covers all possible values. For example, if we have a value v_1 of type $n_1..m_1$ and another value v_2 of type $n_2..m_2$, then the type of expression v_1 add $v2$ will be $n..m$, such that $n = n_1 + n_2$ and $m = m_1 + m_2$.

Records and Unions

Tuples are similar to records in that they consist of multiple fields of possibly different types. Tuples are ordered, and the fields are numbered as 1,2,..., where record fields have alphanumeric names and are not a priori ordered. Records can be trivially reduced to tuples by ordering the fields, e.g., lexicographically, and viewing n-field record as an n-tuple.

A program variable of a union type can have alternative values of different types. For example, the values of a variable of type *mailing address* could be alternatively an email address (of type string) or a physical address consisting of multiple fields such as a street name, city, zip code, and country. A union type can similarly be represented as a tuple by representing the *tag* (e.g., `emailAddress` and `physicalAddress` in our example) in the first component of the tuple, and then the remaining components representing the alternative values for each tag. For example, the second component could be the string for the email address, and the third component could be a tuple for all of the physical address fields. Often the alternative types share components of the same type. For example, both the email address and the street name are strings, so one field could be used to store values belonging to different tags. This helps reduce the number of bits needed to store the alternative values.

From now on, we assume that record and union types have been reduced to tuples. We do this for the simplicity of pre-

Expression	= Translation
$[\![v]\!]^{bool}$	$= v$
$[\![c]\!]^{n..m}$	$= \langle \phi_i \rangle_{i=n}^m, s.t. \quad \phi_i = \begin{cases} \bot & \text{if } i \neq c \\ \top & \text{if } i = c \end{cases}$
$[\![v]\!]^{n..m}$	$= \left\langle [\![v_i]\!]^{bool} \right\rangle_{i=n}^m$
$[\![c_i]\!]^{\{c_1,\ldots,c_n\}}$	$= \langle \phi_j \rangle_{j=1}^n, s.t. \quad \phi_j = \begin{cases} \bot & \text{if } j \neq i \\ \top & \text{if } j = i \end{cases}$
$[\![v]\!]^{\{c_1,\ldots,c_n\}}$	$= \left\langle [\![v_{c_i}]\!]^{bool} \right\rangle_{i=1}^n$
$[\![\text{not } e]\!]^{bool}$	$= \neg [\![e]\!]^{bool}$
$[\![e_1 \text{ and } e_2]\!]^{bool}$	$= [\![e_1]\!]^{bool} \wedge [\![e_2]\!]^{bool}$
$[\![e_1 \text{ or } e_2]\!]^{bool}$	$= [\![e_1]\!]^{bool} \vee [\![e_2]\!]^{bool}$
$[\![e_1 \text{ add } e_2]\!]^{n..m}$	$= \left\langle \bigvee_{\substack{n_1 \leqslant j \leqslant m_1, \\ n_2 \leqslant i-j \leqslant m_2}} \left(\begin{smallmatrix} [\![e_1]\!]^{n_1..m_1}.j \wedge \\ [\![e_2]\!]^{n_2..m_2}.(i-j) \end{smallmatrix} \right) \right\rangle_{i=n}^m$
$[\![e_1 \text{ sub } e_2]\!]^{n..m}$	$= \left\langle \bigvee_{\substack{n_1 \leqslant j \leqslant m_1, \\ n_2 \leqslant j-i \leqslant m_2}} \left(\begin{smallmatrix} [\![e_1]\!]^{n_1..m_1}.j \wedge \\ [\![e_2]\!]^{n_2..m_2}.(j-i) \end{smallmatrix} \right) \right\rangle_{i=n}^m$
$[\![e_1 \text{ mul } e_2]\!]^{n..m}$	$= \left\langle \bigvee_{\substack{n_1 \leqslant j \leqslant m_1, \\ n_2 \leqslant \frac{i}{j} \leqslant m_2}} \left(\begin{smallmatrix} [\![e_1]\!]^{n_1..m_1}.j \wedge \\ [\![e_2]\!]^{n_2..m_2}.(\frac{i}{j}) \end{smallmatrix} \right) \right\rangle_{i=n}^m$
$[\![e_1 \text{ div } e_2]\!]^{n..m}$	$= \left\langle \bigvee_{\substack{n_1 \leqslant j \leqslant m_1, \\ n_2 \leqslant \frac{j}{i} \leqslant m_2}} \left(\begin{smallmatrix} [\![e_1]\!]^{n_1..m_1}.j \wedge \\ [\![e_2]\!]^{n_2..m_2}.(\frac{j}{i}) \end{smallmatrix} \right) \right\rangle_{i=n}^m$
$[\![e_1 \text{ eq } e_2]\!]^{bool}$	$= \bigvee_{j=max(n_1,n_2)}^{min(m_1,m_2)} \left(\begin{smallmatrix} [\![e_1]\!]^{n_1..m_1}.j \wedge \\ [\![e_2]\!]^{n_2..m_2}.j \end{smallmatrix} \right)$
$[\![e_1 \text{ lt } e_2]\!]^{bool}$	$= \bigvee_{\substack{n_1 \leqslant j_1 \leqslant m_1, \\ j_1 < j_2 \leqslant m_2}} \left(\begin{smallmatrix} [\![e_1]\!]^{n_1..m_1}.j_1 \wedge \\ [\![e_2]\!]^{n_2..m_2}.j_2 \end{smallmatrix} \right)$
$[\![e_1 \text{ leq } e_2]\!]^{bool}$	$= [\![e_1 \text{ lt } e_2]\!]^{bool} \vee [\![e_1 \text{ eq } e_2]\!]^{bool}$
$[\![e_1 \text{ gt } e_2]\!]^{bool}$	$= [\![e_2 \text{ lt } e_1]\!]^{bool}$
$[\![e_1 \text{ geq } e_2]\!]^{bool}$	$= [\![e_2 \text{ leq } e_1]\!]^{bool}$

Table 5: Translation of scalar expressions

sentation, but this would also be a good strategy for implementing these types.

Notice that *recursive data types*, with a field of a record or a union type pointing to another value of the same type, could not be handled by this reduction without rather strict additional restrictions, due to there being no size bounds of values of recursive data types. Typical data structures represented as recursive data types are different types of trees. For lists, another data structure involving recursion, could be given a natural Booleanized representation if limited to bounded size lists.

Arrays, Sets, and Tuples

Complex data types such as arrays, sets, and tuples are containers to store a collection of elements. Each of those containers has special properties; arrays map the elements of one set to another. Sets support set-theoretic operations over its elements. Tuples stores elements of different types.

Above, we represented scalar values with a collection of Boolean formulas. We can extend this definition to recursively represent the values of our complex data types. In other words, we can represent arrays, sets, and tuples by a collection of elements, such that each one of those elements is a collection of other elements (Table 6). By this definition, we can represent complex expressions with tree-like structures in which the leaves are Boolean formulas.

Example 1 *The representation of a variable v of the type $\langle 0..1, \{0..2\}\rangle\{0..1\}$ is:*

$$
\begin{aligned}
&[\![v]\!]^{\langle 0..1,\{0..2\}\rangle\{0..1\}}\\
&= \left\langle [\![v_0]\!]^{\langle 0..1,\{0..2\}\rangle}, [\![v_1]\!]^{\langle 0..1,\{0..2\}\rangle} \right\rangle\\
&= \left\langle \begin{array}{l} \left\langle [\![v_{0_1}]\!]^{0..1}, [\![v_{0_2}]\!]^{bool\{0..2\}} \right\rangle,\\ \left\langle [\![v_{1_1}]\!]^{0..1}, [\![v_{1_2}]\!]^{bool\{0..2\}} \right\rangle \end{array} \right\rangle\\
&= \left\langle \begin{array}{l} \left\langle \begin{array}{l} \left\langle v_{0_{1_0}}, v_{0_{1_1}} \right\rangle, \left\langle \begin{array}{l} [\![v_{0_{2_0}}]\!]^{bool},\\ [\![v_{0_{2_1}}]\!]^{bool},\\ [\![v_{0_{2_2}}]\!]^{bool} \end{array} \right\rangle \end{array} \right\rangle,\\ \left\langle \begin{array}{l} \left\langle v_{1_{1_0}}, v_{1_{1_1}} \right\rangle, \left\langle \begin{array}{l} [\![v_{1_{2_0}}]\!]^{bool},\\ [\![v_{1_{2_1}}]\!]^{bool},\\ [\![v_{1_{2_2}}]\!]^{bool} \end{array} \right\rangle \end{array} \right\rangle \end{array} \right\rangle\\
&= \left\langle \begin{array}{l} \left\langle \left\langle v_{0_{1_0}}, v_{0_{1_1}} \right\rangle, \left\langle v_{0_{2_0}}, v_{0_{2_1}}, v_{0_{2_2}} \right\rangle \right\rangle,\\ \left\langle \left\langle v_{1_{1_0}}, v_{1_{1_1}} \right\rangle, \left\langle v_{1_{2_0}}, v_{1_{2_1}}, v_{1_{2_2}} \right\rangle \right\rangle \end{array} \right\rangle
\end{aligned}
$$

The most complicated part of the translation in Table 6 is array indexing, in which we select an element of the array that is associated with the index. To translate this expression, we construct a result value with the same form as the array elements; to fill the content of each component of this value, we iterate over the same component of the array elements to find the value of the matching index. To implement this idea, we define two helper functions in Table 7.

Expression		Translation
$[\![v]\!]^{\langle t_1,\ldots,t_n\rangle}$	$=$	$\left\langle [\![v_i]\!]^{t_i} \right\rangle_{i=1}^{n}$
$[\![v]\!]^{t_v\{t_k\}}$	$=$	$\left\langle [\![v_i]\!]^{t_v} \right\rangle_{i=1}^{\lvert\mathrm{dom}(t_k)\rvert}$
$[\![v]\!]^{\{t\}}$	$=$	$[\![v]\!]^{bool\{t\}}$
$[\![e.i]\!]^{t_i}$	$=$	$[\![e]\!]^{t_1\times\cdots\times t_n}.i$
$[\![e_1[e_2]]\!]^{t_v}$	$=$	$\mathrm{fix}\left(\begin{array}{l}[\![e_1]\!]^{t_v\{t_k\}},\\ \left\langle [\![e_2 \text{ eq } c_i)]\!]^{bool} \right\rangle_{i=1}^{n}\end{array}\right),$ $where \quad \mathrm{dom}(t_k) = \langle c_i\rangle_{i=1}^{n}$
$[\![e_1 \in e_2]\!]^{bool}$	$=$	$[\![e_2[e_1]]\!]^{bool}$
$[\![e_1 \subseteq e_2]\!]^{bool}$	$=$	$\bigwedge_{i=1}^{\lvert[\![e_1]\!]^{\{t\}}\rvert}\left(\begin{array}{r}\neg\,[\![e_1]\!]^{\{t\}}.i \vee\\ [\![e_2]\!]^{\{t\}}.i\end{array}\right)$
$[\![e_1 \cup e_2]\!]^{\{t\}}$	$=$	$\left\langle \begin{array}{l}[\![e_1]\!]^{\{t\}}.i \vee\\ [\![e_2]\!]^{\{t\}}.i\end{array} \right\rangle_{i=1}^{\lvert[\![e_1]\!]^{\{t\}}\rvert}$
$[\![e_1 \cap e_2]\!]^{\{t\}}$	$=$	$\left\langle \begin{array}{l}[\![e_1]\!]^{\{t\}}.i \wedge\\ [\![e_2]\!]^{\{t\}}.i\end{array} \right\rangle_{i=1}^{\lvert[\![e_1]\!]^{\{t\}}\rvert}$
$[\![e_1 \backslash e_2]\!]^{\{t\}}$	$=$	$\left\langle \begin{array}{r}[\![e_1]\!]^{\{t\}}.i \wedge\\ \neg\,[\![e_2]\!]^{\{t\}}.i\end{array} \right\rangle_{i=1}^{\lvert[\![e_1]\!]^{\{t\}}\rvert}$
$[\![e_1 \text{ eq } e_2]\!]^{bool}$	$=$	$\begin{cases}[\![e_1]\!]^{bool} \leftrightarrow [\![e_2]\!]^{bool}, & \text{if } e_1 \text{ and } e_2 \text{ are of type } bool\\ \bigwedge_{i=1}^{\lvert[\![e_1]\!]^{t}\rvert}\left([\![e_1]\!]^{t}.i \text{ eq } [\![e_2]\!]^{t}.i\right), & \text{if } t \text{ is not a scalar type}\end{cases}$

Table 6: Translation of complex data type expressions

Function		Definition
$\mathrm{dom}(bool)$	$=$	$\langle \bot, \top \rangle$
$\mathrm{dom}(n..m)$	$=$	$\langle i \rangle_{i=n}^{m}$
$\mathrm{dom}(\{e_1, \ldots, e_n\})$	$=$	$\langle e_i \rangle_{i=1}^{n}$
$\mathrm{dom}(\langle t_1, \ldots, t_n \rangle)$	$=$	$\langle \langle c_i \rangle_{i=1}^{n} \rangle_{\substack{c_1 \in \mathrm{dom}(t_1), \\ \ldots, \\ c_n \in \mathrm{dom}(t_n)}}$
$\mathrm{dom}(t_v\{t_k\})$	$=$	$\mathrm{dom}\left(\langle t_v \rangle_{i=1}^{\lvert\mathrm{dom}(t_k)\rvert}\right)$
$\mathrm{dom}(\{t\})$	$=$	$\mathrm{dom}(bool\{t\})$
$\mathrm{fix}\left(\begin{matrix}\langle [\![e_i]\!]^{bool} \rangle_{i=1}^{n}, \\ \langle \phi_i \rangle_{i=1}^{n}\end{matrix}\right)$	$=$	$\bigvee_{i=1}^{n}\left([\![e_i]\!]^{bool} \wedge \phi_i\right)$
$\mathrm{fix}\left(\begin{matrix}\langle [\![e_i]\!]^{t} \rangle_{i=1}^{n}, \\ \langle \phi_i \rangle_{i=1}^{n}\end{matrix}\right)$	$=$	$\left\langle \mathrm{fix}\left(\begin{matrix}\langle [\![e_i]\!]^{t}.j \rangle_{i=1}^{n}, \\ \langle \phi_i \rangle_{i=1}^{n}\end{matrix}\right)\right\rangle_{j=1}^{\lvert[\![w]\!]^{t}\rvert}$

Table 7: Helper function definitions

Example 2 *Suppose v is the variable of Example 1 (a variable of type $\langle 0..1, \{0..2\}\rangle\{0..1\}$), and v_k is a variable of type $0..1$. the translation of $v[v_k]$ is:*

$$
\begin{aligned}
&[\![v[v_k]]\!]^{\langle 0..1,\{0..2\}\rangle} \\
&= \mathit{fix}\left(\begin{matrix}[\![v]\!]^{\langle 0..1,\{0..2\}\rangle\{0..1\}}, \\ \left\langle [\![v_k \ \mathit{eq}\ 0]\!]^{bool}, [\![v_k \ \mathit{eq}\ 1]\!]^{bool} \right\rangle\end{matrix}\right) \\
&= \mathit{fix}\left(\begin{matrix}[\![v]\!]^{\langle 0..1,\{0..2\}\rangle\{0..1\}}, \\ \langle v_{k_0}, v_{k_1} \rangle\end{matrix}\right) \\
&= \left\langle \mathit{fix}\left(\begin{matrix}[\![v]\!]^{\langle 0..1,\{0..2\}\rangle\{0..1\}}.i, \\ \langle v_{k_0}, v_{k_1} \rangle\end{matrix}\right)\right\rangle_{i=1}^{2} \\
&= \left\langle \left\langle \mathit{fix}\left(\begin{matrix}[\![v]\!]^{\langle 0..1,\{0..2\}\rangle\{0..1\}}.i.j, \\ \langle v_{k_0}, v_{k_1} \rangle\end{matrix}\right)\right\rangle_{j=1}^{2}\right\rangle_{i=1}^{2} \\
&= \left\langle \left\langle \left\langle \mathit{fix}\left(\begin{matrix}[\![v]\!]^{\langle 0..1,\{0..2\}\rangle\{0..1\}}.i.j.k, \\ \langle v_{k_0}, v_{k_1} \rangle\end{matrix}\right)\right\rangle_{k=1}^{\lvert[\![w]\!]^{t_j}\rvert}\right\rangle_{j=1}^{2}\right\rangle_{i=1}^{2} \\
&= \left\langle \left\langle (v_{0_{j_k}} \wedge v_{k_0}) \vee (v_{1_{j_k}} \wedge v_{k_1}) \right\rangle_{k=1}^{\lvert[\![w]\!]^{t_j}\rvert}\right\rangle_{j=1}^{2}
\end{aligned}
$$

Where $t_1 = 0..1$ and $t_2 = \{0..2\}$.

Reduction to PDDL

So far, we have described an abstract modeling language with complex data types; then, we devised a concrete modeling language based on this abstract syntax to be used as a front-end of a planning system.

A natural way to implement the back-end of the planner is to map the abstract syntax further to an existing intermediate-level modeling language such as PDDL, that already has several scalable and robust implementations. This is what we do next.

Using the bottom-up fashion, we first explain how we construct the fundamental components of the PDDL: the parameters of actions, conditions, and actions' effects. Since their integration into top-level components (actions, initial state, and the goal condition) is trivial, we skip explaining it; but we discuss the challenges we face, and we will provide a solution for them.

Action Parameters

As PDDL only supports parameterization of actions with object-valued parameters, action parameters for the extended language need to be reduced to something simpler. We use the same approach described above to represent action parameters with collections of Boolean variables.

Example 3 *Suppose an action in the extended language has a parameter v of type $\langle 0..1, \{0..2\}\rangle\{0..1\}$, which is represented by the following structure of Boolean variables (Example 1):*

$$
\left\langle \begin{matrix}\left\langle v_{0_{1_0}}, v_{0_{1_1}} \right\rangle, \left\langle v_{0_{2_0}}, v_{0_{2_1}}, v_{0_{2_2}} \right\rangle\right\rangle, \\ \left\langle v_{1_{1_0}}, v_{1_{1_1}} \right\rangle, \left\langle v_{1_{2_0}}, v_{1_{2_1}}, v_{1_{2_2}} \right\rangle\end{matrix} \right\rangle
$$

We decompose this structure to the following set of Boolean variables to determine the parameters of corresponding PDDL action.

$$
\left\{v_{0_{1_0}}, v_{0_{1_1}}, v_{0_{2_0}}, v_{0_{2_1}}, v_{0_{2_2}}, v_{1_{1_0}}, v_{1_{1_1}}, v_{1_{2_0}}, v_{1_{2_1}}, v_{1_{2_2}}\right\}
$$

Conditions and Handling Disjunctions

Like PDDL, the conditions (actions' preconditions, effects' conditions, and goal conditions) in our planning model are expressed by Boolean formulas. However, Most planners do not intrinsically support disjunctions. They compile away disjunctions by converting Boolean formulas into their disjunctive normal form (DNF) and splitting their surrounding structures (i.e., actions or effects) into multiple instances based on conjunctive terms of the DNF (Helmert 2009). However, due to the exponential growth of the formula size in DNF conversion, this approach is often not feasible for the Boolean formulas generated by our translation.

Nebel (2000) has shown that general Boolean formulas in conditions cannot be reduced to conjunctions of literals without adding auxiliary actions, and the number of auxiliary actions in the plans necessarily increases superlinearly. Nebel, however, sketches reductions that increase plan length only polynomially. We use this type of reduction in our planner front-end.

More specifically, for each action a, we eliminate its disjunctions in multiple rounds; in round i, we replace each subformula of the form $\phi = \psi_1 \vee \cdots \vee \psi_n$ by w^i_ϕ, such that $\psi_j, 1 \leqslant j \leqslant n$, contains no disjunction. After m rounds that all disjunctions have been eliminated, we create m auxiliary actions $b^a_1, \ldots, b^a_m$ to maintain the values of our auxiliary variables. For each w^i_ϕ, we add the set of $\{\langle \psi_j, w^i_\phi := \top \rangle \mid 1 \leqslant j \leqslant n\}$ to the effects of the action

b_i^a; moreover, we add $\langle \top, w_\phi^i := \bot \rangle$ to the effects of action a, and initialize all auxiliary variables with the value of $\bot$ in the initial state.

It is also possible that $\psi_j, 1 \leqslant j \leqslant n$, be a parameter of the action a. In this case, we move the parameter from action a to the auxiliary action b_i^a, and add a state variable v^{ψ_j} to the problem. Then, we replace ψ_j by v^{ψ_j} in $b_{i+1}, \ldots, b_m$, and a, and to make everything consistent, we add $\{\langle \psi_j, v^{\psi_j} := \top \rangle, \langle \neg\psi_j, v^{\psi_j} := \bot \rangle\}$ to the effects of b_i^a.

Finally, we enforce the sequence of $b_1^a, \ldots, b_n^a$ to be executed before the execution of action a, by introducing variables of $p_0, p_i^a, 1 \leqslant i \leqslant n$, and:

- adding p_0 and $\{\langle \top, p_0 := \bot \rangle, \langle \top, p_1^a := \top \rangle\}$ to the precondition and effects of b_1^a, respectively,
- adding p_{i-1}^a and $\{\langle \top, p_{i-1}^a := \bot \rangle, \langle \top, p_i^a := \top \rangle\}$ to the precondition and effects of $b_i^a, 2 \leqslant i \leqslant n$, respectively,
- adding p_n^a and $\{\langle \top, p_n^a := \bot \rangle, \langle \top, p_0 := \top \rangle\}$ to the precondition and effects of action a, respectively, and
- initializing p_0 with $\top$ and $p_i^a, 1 \leqslant i \leqslant n$, with $\bot$ in the initial state.

Assignments

In our planning model, the actions' effects are described by a set of conditional assignments, which are pairs of the form $\langle cond, r := e \rangle$. Since the structures of both $[\![r]\!]^t$ and $[\![e]\!]^t$ are the same, to reduce these assignments to Boolean assignments suitable for PDDL, we can just find the Boolean variables in $[\![r]\!]^t$ and their corresponding Boolean formula in $[\![e]\!]^t$, and create conditional Boolean assignments based on them. This procedure is straightforward, except when array indexing is used to specify the reference variables. Using the array indexing feature, we can refer to different elements of an array; the element is determined based on the values of other variables specified by the index expression. Therefore, different Boolean variables may be selected when we have different values for the variables.

To translate the conditional assignment $\langle cond, r := e \rangle$, first, we find the set of pairs of $\langle [\![v]\!]^t, \phi \rangle$ for the reference expression r, such that $[\![v]\!]^t$ is a reference variable with the same structure as $[\![e]\!]^t$, and ϕ is a Boolean formula shows the condition under which r indicates v (Table 8). Then, we can perform the procedure described before. More precisely, for each Boolean variable v_b in $[\![v]\!]^t$, we can find the corresponding element e_b in $[\![e]\!]^t$, and create a conditional Boolean assignment $\langle cond \wedge \phi, v_b := e_b \rangle$. Moreover, since PDDL only supports $\top$ and $\bot$ as the assignment values (add effects and delete effects), we further reduce it to $\langle cond \wedge \phi \wedge e_b, v_b := \top \rangle$ and $\langle cond \wedge \phi \wedge \neg e_b, v_b := \bot \rangle$.

Action Splitting

Most PDDL planners are based on *grounding*: going through all possible combinations of parameter values and creating one non-parametric action for each combination. With complex data type parameters, grounding becomes quickly infeasible due to the high number of parameter value combinations. For example, an action with a set-valued parameter, with values from a set with only 20 elements, would have over one million ground instances. With 30 elements, this would be one billion.

Expression		Definition
${}^r[\![v]\!]^t$	$=$	$\left\{\langle [\![v]\!]^t, \top \rangle\right\}$
${}^r[\![r.i]\!]^{t_i}$	$=$	$\left\{\langle [\![v]\!]^{t_i}.i, \phi \rangle \,\middle|\, \langle [\![v]\!]^t, \phi \rangle \in {}^r[\![r]\!]^t\right\}$, $where,\ t = \langle t_1, \ldots, t_n \rangle$
${}^r[\![r[e]]\!]^{t_v}$	$=$	$\left\{\langle [\![v]\!]^{t_v}.i, \psi \rangle \,\middle|\, \begin{array}{l} \langle [\![v]\!]^t, \phi \rangle \in {}^r[\![r]\!]^t, \\ 1 \leqslant i \leqslant \lvert\mathrm{dom}(t_k)\rvert, \\ c_i = \mathrm{dom}(t_k).i, \\ \omega = [\![c_i \text{ eq } e]\!]^{bool}, \\ \psi = \phi \wedge \omega \end{array}\right\}$, $where,\ t = t_v\{t_k\}$

Table 8: Finding reference variables from reference expressions

	#	LAMA	FDSS	MpC	FF
Rubik	20	8	17	11	10
Buckets	30	3	3	10	30
Scrabble	11	8	8	3	1
$\text{Sudoku}_{\text{scalar}}$	46	3	8	19	4
$\text{Sudoku}_{\text{array}}$		**19**	**25**	**46**	**16**
$\text{Trucks}_{\text{scalar}}$	64	34	35	**45**	**48**
$\text{Trucks}_{\text{set}}$		**42**	**38**	16	38

Table 9: Results of experiments

In the disjunction elimination part, we described the idea of moving one parameter to one of its auxiliary actions, which might mitigate the issue described here by partitioning the parameters to the auxiliary action's parameters. Still, since there is no guarantee that it solves our issue entirely, or the issue might transfer from the action to its auxiliary actions, we need to solve this another way.

To approach this issue, for each action a, we partition its parameters into k sets with at most m elements. Then, we create sub-actions $c_1^a, \ldots, c_k^a$ and move the partitioned parameters to their corresponding sub-action, precisely the same way as we described in disjunction elimination. Furthermore, we can transfer the precondition of action a to c_1^a, to improve the search by preventing choosing parameter values for inapplicable actions. It is worth mentioning that because we have already eliminated disjunctions, we can eliminate all parameters existing in the precondition from action a by replacing them with $\top$.

This is similar to the action splitting done by (Areces et al. 2014) in order to be able to ground actions with a very high number of parameter combinations.

Experiments

We pursued two goals in our experiments. The first goal was to evaluate the practicality of the proposed method to translate problems with complex data types into the most common version of PDDL that supports only Boolean variables. Moreover, our second and more specific goal was to evaluate the effectiveness and improvement of using complex data types compared to the cases that we can also describe our problems with scalar type values without too much difficulty.

For the first goal, we designed some new domains that are difficult to express with only scalar type values. These domains are: the Rubik's cube, the Buckets problems, and the Scrabble game. To describe the Rubik's cube, we need to define a three-dimensional (3D) array; its actions transform this 3D array. The Buckets problem is a famous numeric problem in which we have two buckets; we can fill up, empty, or pour water from one bucket to the other bucket as much as possible. Our goal is to reach a state that a bucket has a certain amount of water. Since our translation support bounded-range integers, we consider this problem to evaluate that feature. Finally, the Scrabble game, which is the most complicated problem, is a game to fill a board (which is a two-dimensional array) with some tiles of alphabets. In each action, a subsequent of alphabets should be chosen such that by putting them on the board, it forms a word in the dictionary. A word is an array of alphabets, and the dictionary is a set of arrays of alphabets.

The experiments are on a number of older and newer planners, including FF (Hoffmann and Nebel 2001), LAMA (Richter and Westphal 2010), MpC (Rintanen 2012), and FDSS (Seipp et al. 2015). We ran the planners with a 30 minute time limit, and report the number of solved instances for each planner in Table 9.

Our experimental results showed that the two domains of the Rubik's cube and the Buckets problem could be solved reasonably well by well-known PDDL planners. However, planners have some difficulty in solving the challenging domain of Scrabble, which means there exist plenty of potential research opportunities in this area.

To better evaluate the effects of using complex data types, we conducted other experiments to compare the performance of solving identical problems in two cases: in one case, we use complex data types, and in another case, we use only scalar data types to describe the same problem. In most existing domains, manipulating complex data types is not reasonable; scalar values provide a more simple and straightforward problem definition. On the other hand, complex data types add only complications to the problem because their operations affect a large number of elements, which makes the reasoning more challenging. However, if the intrinsic nature of a problem requires complex data types, using them to describe the problem might improve the performance.

Here, we examined two domains: Sudoku and the Trucks. In the Sudoku problem, the goal is to fill up a 9×9 board such that each digit between 1 and 9 exists in exactly one cell, in every row, column, and the 3×3 subgrids. We compared two versions of this problem; in the first version, all the board is filled at once by using array data type. In the second version, we fill the board cell-by-cell.

The second domain is the slightly modified version of the Trucks domain, used in IPC 2006. In this domain, a truck should deliver some packages to some location by loading them from other locations and transferring them to their destinations. However, there are also some time constraints that specify the latest arrival time of the packages. In our version, the truck can load a set of packages at once, so we no longer have the spatial constraints of the original domain. However, we enforce the capacity constraints in another way, such that the truck could not load all packages at once.

The results show that using the array data type in the Sudoku domain significantly improves the performance compared to the version with only scalar values. This is mainly because in the former version the action specifies all the required constraints at the begining, compared to the latter version that dead-end nodes will be determined after performing some actions.

Although in the Trucks domain, some planners performed slightly better in the version with the set data type, the results show that Madagascar's performance has been considerably worsen in this version. This is because Madagascar often critically relies on the possibility of performing multiple actions in parallel, and this is not allowed by the way our auxiliary actions for parameterization and elimination of disjunctions are constructed.

Conclusion

We have proposed a very expressive modeling language for planning, with a rich collection of data types, and devised a translation of this language first to Boolean logic, and then further to the Planning Domain Description Language PDDL. We have formalized in our modeling language a number of planning problems which would be clumsy to write in PDDL directly, and shown that they can be solved with off-the-shelf domain-independent planners for PDDL.

We also demonstrated, very surprisingly, that in one case a handcrafted PDDL formalization is solved by PDDL planners *less* efficiently than the generated PDDL formalization produced automatically from our higher-level formalization.

Although adapting some well-known planning techniques to more expressive modeling languages is often a challenge, the potential of the compact problem representations has been recognized and it has already led to successes, for example in the case of Functional STRIPS (Frances and Geffner 2016). Important part of future work is investigating the powerful search methods that directly work on the more expressive language, rather than going through a less powerful intermediate language such as PDDL, as we have done in this work.

References

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

Frances, G.; and Geffner, H. 2016. Effective planning with more expressive languages. In *IJCAI 2016, Proceedings of the 25th International Joint Conference on Artificial Intelligence*, 4155–4159. IJCAI / AAAI Press.

Geffner, H. 2000. Functional STRIPS: a more flexible language for planning and problem solving. In Minker, J., ed., *Logic-based Artificial Intelligence*, volume 597 of *The Springer International Series in Engineering and Computer Science*, 187–209. Springer-Verlag.

Ghallab, M.; Howe, A.; Knoblock, C.; McDermott, D.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. The Planning Domain Definition Language. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, Yale University.

Helmert, M. 2009. Concise finite-domain representations for PDDL planning tasks. *Artificial Intelligence*, 173(5): 503–535.

Hoffmann, J.; and Nebel, B. 2001. The FF planning system: fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14: 253–302.

Kovacs, D. L. 2011. BNF definition of PDDL 3.1. *Unpublished manuscript from the IPC-2011 website*, 15.

Nebel, B. 2000. On the compilability and expressive power of propositional planning formalisms. *Journal of Artificial Intelligence Research*, 12: 271–315.

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

Rintanen, J. 2012. Engineering efficient planners with SAT. In *ECAI 2012. Proceedings of the 20th European Conference on Artificial Intelligence*, 684–689. IOS Press.

Seipp, J.; Sievers, S.; Helmert, M.; and Hutter, F. 2015. Automatic configuration of sequential planning portfolios. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 3364–3370. AAAI Press.