Optimal Infinite-State Planning with Presburger Automata

Björn Borowsky
Stefan Edelkamp

Forschungsbericht Nr. 815
April 2007

Optimal Infinite-State Planning with Presburger Automata

Björn Borowsky and Stefan Edelkamp
Computer Science Department
Otto-Hahn-Str. 14
University of Dortmund

April 24, 2007

Abstract

The paper proposes a new approach to infinite-state action planning with automata theory. State sets and actions are encoded as Presburger formulae and represented using minimized automata. The symbolic exploration that contributes to the planning via model checking paradigm repeatedly applies partitioned images on such automata to compute the automata for the successors of the current state set. The implementation adapts a library for automata manipulation.

1 Introduction

With recent developments of the problem domain description language PDDL [9], modern action planning features arithmetic expressions. Despite that numerical planning is undecidable in general [13], in recent planning competitions, explicit-state planners have reported considerable success in solving benchmark domains with numbers. As these planners all rely on heuristic or local search, there is no guarantee on the plan quality.

For propositional planning, several optimal planning approaches are known. Planners like Satplan [18] and Graphplan [3] optimize the parallel plan length, while other planners optimize the number of steps [21]. For propositional planning problems with preferences [10], where the degree of satisfaction is computed in a (linear) cost metric, optimal plans have been generated [7].

As resources in time and space are limited, compact symbolic representations like BDDs [5] have been exploited to explore the planning state space more efficiently. They represent sets of states and actions uniquely and compactly. State sets can either be generated due to the nondeterministic structure of the problem, in which case the set represents the current belief [2]. In deterministic planning, our focus, state sets are generated during the exploration, like the set of states in a certain breadth-first layer.

One drawback of BDDs is that they refer to a fixed-sized binary state encoding. For propositional domains, a minimized multi-variate variable encoding of a planning problem can be inferred [14], but due to their structural limitations BDDs cannot handle unbounded numbers.

This work proposes a novel metric planning approach for symbolic exploration based on automata theory, which generates step- or cost-optimal plans in domains that contain numerical expressions. The language expressiveness includes linear expressions in the preconditions and effects. Each state set is represented as a set of linear constraints. Therefore, the approach can cope with infinite sets of states.

The paper is structured as follows. First, we introduce to Presburger arithmetics and present an equivalent finite state automata representation, which together with the basic operations allows to prove theorems on first-order expression. Next we consider how to represent state sets and actions in the formalism, making explicit, which fragment of PDDL the approach is designed for. Three different encoding strategies for propositions are discussed. We then formalize the automated translation process of a planning problem into Presburger formulae and automata. Next we propose exploration algorithms that are capable of finding shortest and cost-optimal plans even in infinite state spaces. Finally, we provide an implementation of the planner in *Java* and draw conclusion.

2 Presburger Arithmetics

Presburger arithmetic is the first-order theory of addition and ordering over the integers[1]. Terms in Presburger arithmetic consist of constants 0 and 1 and sums of terms. For example, $x+x+1+1+1=2x+3$ is a term. Equations and inequations over terms are atomic formulae. The first-order theory over the natural numbers with addition are the set of all sentences, which are true in first-order predicate logic over atomic formulae. Given a successor function the Presburger arithmetic expressions can be formalized by a set of axioms and a schema for induction.

The automata encoding for Presburger formulae exploits the fact that integer numbers can be expressed by words over the alphabet $\{0,1\}$, using the 2-complement representation. Here, the binary representation is written from left to right, such that the most significant bit is located left. The representation is not unique as the bit string can be prefixed with the an arbitrary number of zeros without changing its interpretation. In general, vectors of integer numbers can be represented as terms over $\{0,1\}^n$. For example, the pair $(4,13)$ generates the vectors

$$\binom{00100}{01101} = \binom{000100}{001101} = \binom{0000100}{0001101} = \dots$$

The automata representation accepts all solutions to a Presburger equation or inequation. For example, an automaton for $x-3y=1$ has 5 states (see Figure 1): The initial state a whose outgoing transitions interpret the sign bits correctly, b for $x-3y=0$, c for $x-3y=2$, d for $x-3y=1$ (accepting) and e, which the automaton takes if the word already read is not a prefix

[1] Usually the Presburger arithmetic is defined on natural numbers, but negative numbers can be realized using the 2-complement.

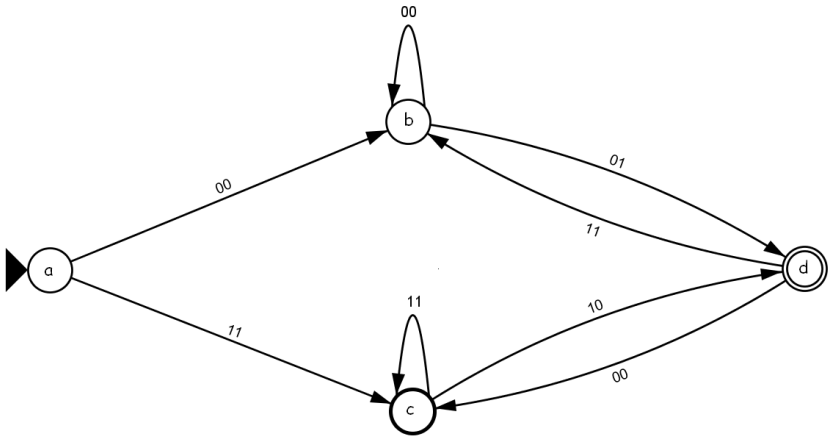

Figure 1: Sample automaton for $x - 3y = 1$ (e omitted).

of some solution. The transition from a to b is labeled by $\binom{0}{0}$. For $\binom{0}{1}$ we go to e, since for $x = 0$ and $y = -1$, $x - 3y$ has a value of 3 which cannot become smaller when reading further bitvectors, as x cannot become smaller and y cannot become larger. For a similar reason we go to e when reading $\binom{1}{0}$. For $\binom{1}{1}$ we go to c.

As the automaton for a Presburger formula is not unique, minimization is needed. One of the main advantages of Presburger formulae compared to most other representation formalisms in planning is that they can concisely represent infinite sets of states. For example the set of all odd numbers is represented by the equation $\exists k : x = 2k + 1$ and leads to an automaton with two states.

The conjunction of two atomic formulae is realized via intersecting the two automata. The disjunction of two atomic formulae might yield a non-deterministic automaton (that can be determinized with a possible exponential blow up). Negation requires to represent the complement of the language accepted by the automata, which (in case it is deterministic) simply toggles the acceptance condition. The projection of one variable is a linear-time operation, that also may lead to a non-deterministic automaton.

To prove the correctness of a theorem in Presburger arithmetic, one simply has to construct the automaton for it, and check, if its language is not empty. A lower bound for such decisions is triple exponential in the size of the formula [8], and an matching upper bound with automata has been given by [19].

3 Representing State Sets and Actions

This section describes how the semantics of a problem and its domain can be represented as a set of minimized finite automata. Henceforth, we assume that both the problem and the domain have been fully-instantiated. Our approach supports many of the elements of PDDL2.2 [16]. We impose the following restrictions (**R**):

1. Functions have integer values only.
2. Numerical expressions are linear.
3. Linear expressions used in preconditions or effects do not contain fractional numbers or divisions.
4. Scalars used in `scale-up` or in `scale-down` assignments are constant expressions.
5. There are no temporal actions.
6. The domain does not contain conditional effects.
7. The problem does not specify timed initial literals.

In the simplest case, a problem is given by one initial state and a set of goal states. An action can be modeled as a (finite or infinite) set of transitions within the state space, where a transition t is a pair (s_1, s_2) of a predecessor state s_1 and a successor state s_2.

3.1 Encoding the State Space

The state space induced by the domain is encoded using integer variables such that sets of states as well as sets of transitions can be characterized by Presburger formulae.

3.1.1 Numerical components

A planning domain defines a set $\mathcal{F} = \{f_1, \ldots, f_n\}$ of zero-arity functions and a set $\mathcal{P} = \{p_1, \ldots, p_m\}$ of zero-arity predicates, with $n, m \in \mathbb{N}_0$. We map each function f_i, $i = 1, \ldots, n$, onto a variable x_i, in the following referred to as a *current state variable*. The variables x_i in a Presburger formula describe properties of the numerical components of those states that are members of the state set to be characterized by the formula. Since x_i can be assigned to arbitrary integer, we are able to encode infinite sets of states. To be capable of characterizing transition sets, too, for each current state variable x_i we create a *successor state variable* x_i', which denotes the value of x_i in the successor state.

3.1.2 Propositional components

Predicates can be represented in a similar straightforward way, but it may be rewarding to choose a more advanced encoding. In our work we have considered the following three encodings.

Definition 1. *(Binary Encoding) A* binary encoding *introduces a current state variable y_j and a successor state variable y_j' for $j = 1, \ldots, m$. If predicate p_j is currently* true*, then y_j is assigned to -1. The fact that p_j is currently* false *is expressed through $y_j = 0$. The current state variable and the successor state variable belonging to a predicate p are denoted by* $\mathrm{curr}(p)$ *and* $\mathrm{succ}(p)$*, respectively.*

Definition 2. *(Prime Number Encoding) Let $G_1, \ldots, G_l$ be the blocks of a partition of $\mathcal{P}$. For each $p_j \in \mathcal{P}$, let* group(p_j) *be the index $k \in \{1, \ldots, l\}$ for which $p_j \in G_k$ holds. A* prime number encoding *introduces a current state variable g_k and a successor state variable g'_k for each group G_k of predicates. Each $p_j \in \mathcal{P}$ is assigned to a prime number* prime(p_j)*, which is unique within $G_{\text{group}(p_j)}$. The domain of g_k and g'_k is the set of all products of prime numbers assigned to the members of G_k, where the prime factors of each product must be pairwise different. Predicate p_j is* true *if, and only if,* prime(p_j) *is a prime factor of $g_{\text{group}(p_j)}$. The current state variable and the successor state variable belonging to a group G are denoted by* curr(G) *and* succ(G)*, respectively.*

Definition 3. *(Mutex Group Encoding) Let $G_1, \ldots, G_l$ be the blocks of a partition of $\mathcal{P}$, where all predicates being the member of the same group G_k are mutual exclusive, i.e., at most one predicate in G_k can be* true *at any point in time. Again, for each $p_j \in \mathcal{P}$, let* group(p_j) *be the index $k \in \{1, \ldots, l\}$ for which $p_j \in G_k$ holds. A* mutex group encoding *introduces a current state variable g_k and a successor state variable g'_k for each predicate group G_k. All members of a group G_k are numbered uniquely from 1 to $|G_k|$. Let* num(p_j) *be the number of p_j within $G_{\text{group}(p_j)}$. The domain of g_k and g'_k is $\{0, \ldots, |G_k|\}$. Predicate p_j is* true *if, and only if, $g_{\text{group}(p_j)} =$* num(p_j)*. The fact that no predicate in G_k is currently true is expressed through $g_k = 0$. The current state variable and the successor state variable belonging to a group G are also denoted by* curr(G) *and* succ(G)*, respectively.*

The binary encoding straightforward, but it doesn't make any attempts to reduce the dimensionality of the state space. For prime number encodings, moderate group sizes should be selected, as products grow very fast for an increasing number of factors. The larger the amount of places at which a predicate p_j occurs, the lower *prime*(p_j) should be. Groups should be sized as similar as possible. For many domains it is possible to automatically identify groups of mutual exclusive predicates [14]. If such a partitioning is known, a corresponding mutex group encoding should be chosen.

3.2 Linear Expressions and Numerical Effects

Linear expressions may occur in effects as the right side (rvalue) of assignments (like `increase`), and in preconditions as operands of binary comparisons (like <=). Each linear expression is simplified such that the result is of the form $a_0 + a_1 z_1 + \cdots + a_d z_d$, where $a_0, \ldots, a_d$ are non-zero integers and $z_1, \ldots, z_d$ are pairwise different current state variables. We can do that because we require that linear expressions occurring in effects or preconditions do not make use of non-integer numbers or divisions. For a linear PDDL2.2 expression e let *lin*(e) be this simplified representation.

Let *trans*(P) be a Presburger formula describing a PDDL2.2 segment P. For a function f_i, linear expressions e, e_1 and e_2, a constant expression c and a relation $\prec \in \{=, <=, <, >=, >\}$, we define

- *trans*(`(assign` f_i e`)`) := $(x'_i =$ *lin*$(e))$

- *trans*((increase f_i e)) := $(x_i' = x_i + \mathit{lin}(e))$
- *trans*((decrease f_i e)) := $(x_i' = x_i - \mathit{lin}(e))$
- *trans*((scale-up f_i c)) := $(x_i' = \mathit{lin}(c) \cdot x_i)$
- *trans*(($\prec$ e_1 e_2)) := $(\mathit{lin}(e_1) \prec \mathit{lin}(e_2))$

Obviously, for an assignment a the formula *trans*(a) describes exactly the set of transitions which correctly update the lvalue of a in dependence on the old content of the lvalue and the rvalue, evaluated before applying a. Functions other than the one updated by a can be changed arbitrarily. The translation of binary comparisons should be self-explanatory.

Due to the restriction on integers, our semantics of `scale-down` differs from the PDDL2.2 semantics:

- *trans*((scale-down f_i c)) :=
  $((x_i \geq 0 \wedge \exists r : (x_i = \mathit{lin}(c) \cdot x_i' + r \wedge 0 \leq r < |\mathit{lin}(c)|)) \vee$
  $(x_i < 0 \wedge \exists r : (x_i = \mathit{lin}(c) \cdot x_i' - r \wedge 0 \leq r < |\mathit{lin}(c)|)))$

The result of a division is the non-fractional part of the result a precise division would yield. The formula is a distinction of two cases. For one state exactly one disjunct is satisfiable. If x_i is not negative, then x_i' will be equal to $\frac{\mathit{lin}(c)}{|\mathit{lin}(c)|} \frac{x_i - (x_i \, mod \, |\mathit{lin}(c)|)}{|\mathit{lin}(c)|}$, which is equivalent to $x_i = \mathit{lin}(c) \cdot x_i' + (x_i \, mod \, |\mathit{lin}(c)|)$. For $x_i < 0$, we have $x_i' = -\frac{\mathit{lin}(c)}{|\mathit{lin}(c)|} \frac{-x_i - ((-x_i) \, mod \, |\mathit{lin}(c)|)}{|\mathit{lin}(c)|}$, which is equivalent to $x_i = \mathit{lin}(c) \cdot x_i' - (x_i \, mod \, |\mathit{lin}(c)|)$.

3.3 Propositional Effects and Access to Predicates

Propositional preconditions access predicates in a reading, whereas propositional effects access predicates in a writing manner. As the translation of these accesses is also determined by the encoding of the logical state space, we will discuss special aspects of the translation process for each encoding introduced above.

For $p \in \mathcal{P}$, let *read*(p) be a Presburger formula which is *true* in a state s, if, and only if, p is *true* in s according to the state encoding.

If one pair of Presburger variables describes a group of predicates instead of a single predicate, then the computation of the value for the successor state variable may require complete knowledge about all updates to some member of the group performed by an effect. For this reason, the translation of an effect is more complex than the translation of a read access. For $D \subseteq \mathcal{P}$ and $A \subseteq \mathcal{P}$, *write*(D, A) is a Presburger formula, which is *true* for a transition $t = (s_1, s_2)$, if, and only if,

- all predicates $p \in D$ are *false* in the successor state s_2,
- all predicates $p \in A$ are *true* in s_2, and

- all predicates $p \in \mathcal{P} \backslash (D \cup A)$ are *true* in s_2 (if p is *true* in s_1).

For $D \cap A = \emptyset$, *write*(D, A) is undefined. The formula *write*(D, A) can be used to encode the logical part of an effect by choosing for A the set of all predicates that occur in a positive literal and choosing for D the set of all predicates that occur in a negative literal, but not in a positive literal (PDDL2.2 first executes the deletions, and then executes the addings). Note that $(D, A) = (\emptyset, \emptyset)$ is a valid choice.

The initial state could be described by a goal description, which is translated into a Presburger formula that reads the predicates, but as we will see later, negations may cause the acceptance of invalid states. On the one hand, the invalid states could be removed by building the conjunct of the formula with another formula that characterizes the whole state space. On the other hand, it is possible to describe an initial state by a much more elegant formula, because due to the fact that all predicate valuations are known, the formula needs not to read the current state variables. This motivates the definition of *init*. For a set $T \subseteq \mathcal{P}$, *init*(T) is a Presburger formula, which accepts only one logical state, namely the state in which, according to the state encoding, $p \in \mathcal{P}$ is interpreted as *true* if, and only if, $p \in T$.

In the following, we provide definitions of *read*, *write* and *init* for all three encodings.

3.3.1 Binary Encoding

Per construction, a predicate $p \in \mathcal{P}$ is to be interpreted as being *true*, if, and only if, *curr*(p) is equal to -1. Obviously, for the following definitions of $\mathit{read}_{\mathit{bin}}$, $\mathit{write}_{\mathit{bin}}$ and $\mathit{init}_{\mathit{bin}}$, the axioms of *read*, *write* and init hold with respect to the semantics of the binary encoding:

$$\mathit{read}_{\mathit{bin}}(p) := (\mathit{curr}(p) = -1)$$
$$\mathit{write}_{\mathit{bin}}(D, A) := (\textstyle\bigwedge_{d \in D}(\mathrm{succ}(d) = 0) \wedge \bigwedge_{a \in A}(\mathit{succ}(a) = -1) \wedge \bigwedge_{p \in \mathcal{P} \backslash (D \cup A)}(\mathit{succ}(d) = \mathit{curr}(d)))$$
$$\mathit{init}_{\mathit{bin}}(T) := (\bigwedge_{t \in T}(\mathit{curr}(t) = -1) \wedge (\bigwedge_{p \in \mathcal{P} \backslash T}(\mathit{curr}(p) = 0))$$

3.3.2 Prime Number Encoding

To read a predicate $p \in \mathcal{P}$ in prime number encoding, it is sufficient to check, if *prime*(p) is a prime factor of $\mathit{curr}(G_{\mathit{group}(p)})$. Thus,

$$\mathit{read}_{\mathit{prime}}(p) := (\exists k : \mathit{curr}(G_{\mathit{group}(p)}) = \mathit{prime}(p) \cdot k)$$

is a sound definition of $read$ with respect to the prime number encoding.

A formula describing the propositional part of an effect must take into account two aspects. First, if p is added by the effect, then the value of $\mathit{curr}(G_{\mathit{group}(p)})$ is to be multiplied with *prime*(p_j) only if prime(p_j) is not already a prime factor of $\mathit{curr}(G_{\mathit{group}(p)})$. Similarly, if p is deleted by the effect, then the value of $\mathit{curr}(G_{\mathit{group}(p)})$ is to be divided by $prime(p_j)$ only if *prime*(p_j) is a prime factor of $\mathit{curr}(G_{\mathit{group}(p)})$. Second, it must update each group variable

according to all updates that affect some member of the group. If the formula simply was a conjunction of subformulae, where each subformula describes the value of $\mathit{succ}(G_{\mathit{group}(p)})$ for exactly one $p \in \mathcal{P}$ that is updated by the effect, then the formula might be contradictious when a group is affected by more than one update.

We first build formulae, which characterize updates to single predicates, ignoring what happens to all group variables other than the one affected by the update. For a predicate $p \in \mathcal{P}$ and two variables h_0 and h_1, we define

$\mathit{add}(p, h_0, h_1) := (((\exists k : \mathit{curr}(G_{\mathit{group}(p)}) = \mathit{prime}(p) \cdot k) \wedge h_1 = h_0) \vee (\neg(\exists k : \mathit{curr}(G_{\mathit{group}(p)}) = \mathit{prime}(p) \cdot k) \wedge h_1 = \mathit{prime}(p) \cdot h_0))$

$\mathit{del}(p, h_0, h_1) := (((\exists k : \mathit{curr}(G_{group(p)}) = \mathit{prime}(p) \cdot k) \wedge \mathit{prime}(p) \cdot h_1 = h_0) \vee (\neg(\exists k : \mathit{curr}(G_{\mathit{group}(p)}) = \mathit{prime}(p) \cdot k) \wedge h_1 = h_0))$.

The formula $\mathit{add}(p, \mathit{curr}(G_{\mathit{group}(p)}), \mathit{succ}(G_{\mathit{group}(p)}))$ is *true* for a transition $t = (s_1, s_2)$ if either p is *true* in s_1 and the variable representing the group p belongs to remains unchanged so that p is also *true* in s_2, or p is $false$ in s_1 and the variable representing the group p belongs to is multiplied with $\mathit{prime}(p)$, such that p is *true* in s_2. Thus, $\mathit{add}(p, \mathit{curr}(G_{\mathit{group}(p)}), \mathit{succ}(G_{\mathit{group}(p)}))$ correctly characterizes the adding of p ignoring all other group variables. An analogous argumentation shows that $\mathit{del}(p, \mathit{curr}(G_{\mathit{group}(p)}), \mathit{succ}(G_{\mathit{group}(p)}))$ correctly characterizes the deletion of p ignoring all other group variables.

The definitions of *add* and *del* cover the first aspect. The parameters h_0 and h_1 enable a formula that represents the whole effect to cover the second aspect, too. In order to avoid contradictions, an update to p does not write its result to $\mathit{succ}(G_{\mathit{group}(p)})$, but into an own auxiliary variable, which becomes the input for the next update. The first auxiliary variable is identical to $\mathit{curr}(G_{\mathit{group}(p)})$ and becomes the input for the first update, the last auxiliary variable is the output of the last update and is identical to $\mathit{succ}(G_{\mathit{group}(p)})$. By existentially quantifying the auxiliary variables away, we obtain the formula we were looking for. For $i = 1, \ldots, l$, be $G_i \cap D = \{d_{i,1}, \ldots, d_{i,l_i^D}\}$ and $G_i \cap A = \{a_{i,1}, \ldots, a_{i,l_i^A}\}$. Now we define

$$\begin{aligned}\mathit{write}_{\mathit{prime}}(D, A) := (&\textstyle\bigwedge_{1 \le k \le l} \exists h_0 : \ldots \exists h_{|G_k|} : \\ &(h_0 = \mathit{curr}(G_k) \wedge \mathit{succ}(G_k) = h_{l_k^D + l_k^A} \\ &\wedge \textstyle\bigwedge_{1 \le j \le l_k^D} \mathit{del}(d_{k,j}, h_{j-1}, h_j) \\ &\wedge \textstyle\bigwedge_{1 \le j \le l_k^A} \mathit{add}(a_{k,j}, h_{l_k^D + j - 1}, h_{l_k^D + j})))\end{aligned}$$

$\mathit{init}_{\mathit{prime}}(T) := (\bigwedge_{1 \le k \le l} \mathit{curr}(G_i) = \prod_{t \in T \cap G_k} \mathit{prime}(t))$.

3.3.3 Mutex Group Encoding

To check, if a predicate p is *true* in a mutex group encoding it suffices to compare the appropriate group variable to $\mathit{num}(p)$:

$\mathit{read}_{\mathit{mutex}}(p) := (\mathit{curr}(G_{\mathit{group}(p)}) = \mathit{num}(p))$.

If an effect adds $p \in \mathcal{P}$, $\mathit{num}(p)$ will be the correct new value for $\mathit{curr}(G_{\mathit{group}(p)})$. Now we assume that no member of G_i is affected by an add. If $p \in G_i$ is deleted by the effect while p is currently *true*, then we must choose $\mathit{succ}(G_i) = 0$, otherwise $\mathit{succ}(G_i) = \mathit{curr}(G_i)$ is appropriate. The formula $\mathit{delset}(G, D)$ correctly updates the variable representing a group G, where no $p \in G$ is affected by an add, and D is the set of all deletes affecting some member of G. Now this subsection can be concluded with the definition of $\mathit{write}_{\mathit{mutex}}$ and $\mathit{init}_{\mathit{mutex}}$.

$\mathit{delset}(G, D) := (((\bigvee_{d \in D} \mathit{curr}(G) = \mathit{num}(d)) \wedge \mathit{succ}(G) = 0) \vee ((\bigwedge_{d \in D} \mathit{curr}(G) \neq \mathit{num}(d)) \wedge \mathit{succ}(G) = \mathit{curr}(G)))$

$\mathit{write}_{\mathit{mutex}}(D, A) := ((\bigwedge_{G \in \{G_1,\ldots,G_l\}: A \cap G = \emptyset} \mathit{delset}(G, G \cap D)) \wedge (\bigwedge_{G \in \{G_1,\ldots,G_l\}: A \cap G = \{p\}} \mathit{curr}(G) = \mathit{num}(p)))$

$\mathit{init}_{\mathit{mutex}}(T) := ((\bigwedge_{G \in \{G_1,\ldots,G_l\}: T \cap G = \{p\}} \mathit{curr}(G) = \mathit{num}(p)) \wedge (\bigwedge_{G \in \{G_1,\ldots,G_l\}: T \cap G = \emptyset} \mathit{curr}(G) = 0))$.

3.4 Derived Predicates

A grounded derived predicate is of the form `(:derived (`p_j`)` Φ`)`, where p_j is a predicate and Φ is a goal description. Φ may contain other predicates, which may be derived themselves. For two derived predicates p and q be $p \prec q$ if, and only if, q depends on p, that is, there is a sequence $z_0, \ldots, z_l$ of derived predicates, such that $z_0 = p$, for each z_k, $1 \leq k \leq l$ there is a `:derived` definition, which derives z_k from a goal description containing z_{k-1}, and $z_l = q$. $\prec$ is reflexive and transitive. As the set of all `:derived` definitions of a domain must not contain cyclic definitions, $\prec$ is also antisymmetric. Thus, $\prec$ is a partial order.

Now it is clear that a `:derived` definition can be seen as a macro for its goal description with the predicate it derives as its name. Our aim is to remove all derived predicates from the domain by substituting each occurence of a derived predicate q by a goal description, which is equivalent to q. In the domain, we first replace all `:derived` definitions $d_1, \ldots, d_z$ deriving the same predicate q by one new `:derived` definition deriving q from the disjunction of the goal descriptions of $d_1, \ldots, d_z$. Afterwards we topologically sort all derived predicates according to $\prec$ and obtain a linear order $q_1 \prec \ldots \prec q_c$. Predicate q_1 is one of the most independent derived predicates, i.e., q_1 is independent from all other derived predicates. For $k = 1, \ldots, c$, all other derived predicates predicate q_k depends on lay within $\{q_1, \ldots, q_{k-1}\}$. For $k = 1, \ldots, c$, in the goal description of the `:derived` definition belonging to q_k we substitute each occurrence of a derived predicate q by the goal description of the `:derived` definition belonging to q (which is already free of derived predicates). When this has been done, then each occurrence of a derived predicate q in a precondition of an action or in the goal specification of the problem is substituted by the goal description of the `:derived` definition belonging to q.

3.5 Goal Descriptions, Effects and Initializations

We extend our translation function *trans* to goal descriptions. For a predicate $p \in \mathcal{P}$ and goal descriptions $d_1, \ldots, d_z$ the following definitions are given:

- $\textit{trans}(p) := \textit{read}(p)$
- $\textit{trans}(\texttt{(not } d_1\texttt{)}) := (\neg \textit{trans}(d_1))$
- $\textit{trans}(\texttt{(and } d_1 \ldots d_z\texttt{)}) := (\bigwedge_{1 \leq k \leq z} \textit{trans}(d_k))$
- $\textit{trans}(\texttt{(or } d_1 \ldots d_z\texttt{)}) := (\bigvee_{1 \leq k \leq z} \textit{trans}(d_k))$
- $\textit{trans}(\texttt{(imply } d_1\, d_2\texttt{)}) := (\neg \textit{trans}(d_1) \vee \textit{trans}(d_2))$

Goal descriptions only read current state variables.

A PDDL2.2 effect is a conjunction of numerical effects, where literals are either positive or negative. For an arbitrary effect e, let $\textit{pos}(e)$, $\textit{neg}(e)$ and $\textit{fluent}(e)$ denote the set of predicates occurring in a positive literal, the set of predicates occurring in a negative literal and the set of numerical effects contained in e, respectively. Let $\textit{pres}(e)$ be the set of all functions $f \in \mathcal{F}$ that are not affected by some $a \in \textit{fluent}(e)$. The translation of e is

- $\textit{trans}(e) := (\textit{write}(\textit{neg}(e) \backslash \textit{pos}(e), \textit{pos}(e)) \wedge (\bigwedge_{a \in \textit{fluent}(e)} \textit{trans}(a)) \wedge (\bigwedge_{f \in \textit{pres}(e)} \textit{succ}(f) = \textit{curr}(f)))$

An initialization specified by a problem can be seen is a conjunction of literals and equations. Literals not mentioned in the `:init` section are assumed to be *false* (closed world assumption). Here, functions that are not explicitly initialized are initialized with 0. For an arbitrary `:init` section *start*, let $\textit{pos}(\textit{start})$, $\textit{equat}(\textit{start})$ and $\textit{uinit}(\textit{start})$ denote the set of predicates occurring in a positive literal, the set of equations and the set of not explicitly initialized functions $f \in \mathcal{F}$, respectively. Then $\textit{trans}(\textit{start})$ can be defined as

- $\textit{trans}(\textit{start}) := (\textit{init}(\textit{pos}(\textit{start})) \wedge (\bigwedge_{u \in \textit{equat}(\textit{start})} \textit{trans}(u)) \wedge (\bigwedge_{v \in \textit{uinit}(\textit{start})} \textit{curr}(v) = 0))$

3.6 Domains and Problems

For an action *act* with a precondition *pre* and an effect *post*, the translation is defined through

- $\textit{trans}(\textit{act}) := (\textit{trans}(\textit{pre}) \wedge \textit{trans}(\textit{post}))$

The transition relation defined by $\textit{trans}(\textit{act})$ contains a transition $t = (s_1, s_2)$ if, and only if, the predecessor state s_1 fulfills the precondition, and the successor state s_2 is the result of applying *post* to s_1.

The translation of a problem and its domain consists of the following steps:

1. Substitution of derived predicates in all conditions, preconditions and in the goal specification of the problem
2. Selection of a logical state encoding
3. Translation of the actions
4. Translation of the `:init` section

How a `:metric` specification of a problem is translated, will be explained in the planning section.

3.7 Building Automata for the Formulae

The next step is to create a minimized deterministic finite state automaton (DFA)

$$A_\varphi = (Q_\varphi, \Sigma_{\textit{trans}}, \delta_\varphi, \textit{init}_\varphi, F_\varphi)$$

for each formula φ, using the algorithm described in [20], with Q_φ being the state set, $\Sigma_{\textit{trans}}$ being the alphabet, $\delta_\varphi : Q \times \Sigma_{\textit{trans}} \rightarrow Q$ being the transition relation, $\textit{init}_\varphi \in Q_\varphi$ being the initial state and $F_\varphi \subseteq Q$ being the set of accepting states. A_φ recognizes the set of all solutions to φ. The numerical state space is represented by n current state variables and n successor state variables. Let l be the number of pairs consisting of a current state variable and a successor state variable, which represent the logical state space. The bitvectors of the alphabet $\Sigma_{\textit{trans}} = \{0,1\}^{2(n+l)}$ have one component for each variable. We assign all current state variables to the components with the indices $1, 3, \ldots, 2(n+l)-1$, whereas all successor state variables are assigned to the components with the indices $2, 4, \ldots, 2(n+l)$. For the initial state and the goal state set, automata are constructed, too. Since these automata recognize sets of planning states rather than sets of transitions from one planning state to another, they use the alphabet $\Sigma_{\textit{state}} := \{0,1\}^{n+l}$.

The entire representation of the domain and the problem is denoted by $\mathcal{R}$, with $\mathcal{R} = (\mathcal{A}, \mathcal{I}, \mathcal{G})$, where $\mathcal{A}$ is the set of all automata constructed for the actions, $\mathcal{I}$ is the automaton constructed for the initial state, and $\mathcal{G}$ is the automaton constructed for the goal state set.

4 Planning

For deterministic finite automata A_1 and A_2, $A_1 \cap A_2$ ($A_1 \cup A_2$) denote the minimized deterministic finite automata recognizing the intersection (union) of the languages $L(A_1)$ and $L(A_2)$ recognized by A_1 and A_2; $A_1 \backslash A_2$ be the minimal DFA recognizing $L(A_1) \backslash L(A_2)$. $A_1 = A_2$ hold if, and only if, $L(A_1) = L(A_2)$. For a DFA $A = (Q, \Sigma_{\textit{trans}}, \delta, \textit{init}, F)$ the *projection* $\textit{proj}_{\textit{curr}}(A)$ is defined as the result of determinizing and minimizing

$$A' := (Q, \Sigma_{\textit{state}}, \delta', \textit{init}, F)$$

with $\delta'(q,(a_1,\ldots,a_{n+l})) := \{r \in Q \mid \exists b_1,\ldots,b_{n+l} : \delta(q,(a_1,b_1,\ldots,a_n,b_n)) = r\}$, whereas the *projection* $proj_{succ}(A)$ is defined as the result of determinizing and minimizing

$$A' := (Q, \Sigma_{state}, \delta', init, F)$$

with $\delta'(q,(b_1,\ldots,b_{n+l})) := \{r \in Q \mid \exists a_1,\ldots,a_{n+l} : \delta(q,(a_1,b_1,\ldots,a_n,b_n)) = r\}$. In both cases, before determinization and minimization is done, A' is modified by another algorithm to ensure that all representations of a solution are accepted (see ...). The alphabet can be restored using the *expansions* exp_{curr} and exp_{succ}. The set $exp_{curr}(A)$ is obtained by replacing the transition relation $\delta'' : Q \times \Sigma_{state} \to Q$ of A by $\delta''' : Q \times \Sigma_{trans} \to Q$, defined through $\delta'''(q,(a_1,b_1,\ldots,a_{n+l},b_{n+l})) := \delta''(q,(a_1,\ldots,a_{n+l}))$. The set exp_{succ} is defined analogously with $\delta'''(q,(a_1,b_1,\ldots,a_{n+l},b_{n+l})) := \delta''(q,(b_1,\ldots,b_{n+l}))$. The minimal DFA A with $L(A) = \emptyset$ using the alphabet Σ_{state} is denoted by $\bot$ (the complement of $\bot$ is $\top$).

4.1 Breadth First Search

Given a representation $\mathcal{R} = (\mathcal{A}, \mathcal{I}, \mathcal{G})$, we are looking for a shortest sequential plan, which transforms the initial planning state recognized by $\mathcal{I}$ into a goal state $G \in L(\mathcal{G})$. The planning graph induced by $\mathcal{R}$ is infinite, directed and unweighted, but the outdegree of each node is finite; it cannot exceed $|\mathcal{A}|$. A shortest sequential plan corresponds to a shortest path from $L(\mathcal{I})$ to a $G \in L(\mathcal{G})$. In the following, we identify automata A with their languages $L(A)$.

Algorithm 1 Breadth-First-Search

Input: Representation $\mathcal{R} = (\mathcal{A}, \mathcal{I}, \mathcal{G})$
Output: Shortest sequential plan or 'no plan'

```
stack := ∅; layer := reached := I; stack->push(I)
while layer ∩ G = ⊥ do
    layer' := ⊥
    for all A ∈ A do
        trans := exp_curr(layer) ∩ A
        layer' := layer' ∪ proj_succ(trans)
    layer := layer'\reached
    if layer = ⊥ then
        return 'no plan'
    else
        reached := reached ∪ layer
        stack->push(layer)
stack->pop()
stack->push(layer ∩ G)
return Extract-Plan(R, stack)
```

Obviously, a breadth first search that starts from $\mathcal{I}$ is capable of finding a shortest path (cf. Algorithm 1). For each action whose precondition is *true* in the initial state, we compute the

successor state that results from applying the action to the initial state. The set of all these successor states, without the initial state, forms the next layer. Because DFAs are closed under union, it is possible to represent the layer by one single DFA. Be $L_0 := \mathcal{I}$. When the exploration has completed layer $L_i (i \geq 0)$, layer L_{i+1} is obtained by doing the following for all $A \in \mathcal{A}$. First, $B := \mathit{exp}_{\mathit{curr}}(L_i) \cap A$ is computed. L_i is not compatible to A since L_i is a set of planning states, whereas A is a set of transitions. $exp_{curr}(L_i)$ converts L_i to the set of transitions from any planning state in L_i to any planning state. B covers exactly those transitions (s_1, s_2) with s_1 in L_i, s_1 fulfilling the precondition of the action the automaton A corresponds to, and s_2 being the successor state that results from executing the action in s_1. The projection $C := \mathit{proj}_{\mathit{succ}}(B)$ only recognizes the successor states. L_{i+1} becomes the union of all C computed for the $A \in \mathcal{A}$ minus the set of all states reached by the exploration so far. By intersecting $L_i (i = 0, \ldots)$ with $\mathcal{G}$ before computing L_{i+1}, we check whether or not we have already found a goal state. All L_i are stored, as they are needed for the reconstruction of a plan. If $L_{i+1} = \bot$, then that part of the state space, which is reachable from the initial state has been completely explored without finding a goal state. In this situation, we have proved that there is no solution. Exploration does not terminate if infinite many planning states are reachable from the initial state while there is no solution.

4.2 Reconstruction of a Solution

In situations where algorithm 1 reaches a nonempty subset $G \subseteq \mathcal{G}$, we will mainly be interested in a plan, which transforms the initial state into one of the members of G. If the length z of a shortest plan is at least 1, then algorithm 1 creates a stack with L_0 on its bottom and $G = L_z \cap \mathcal{G}$ on its top (otherwise the stack will only contain $L_0 \cap \mathcal{G}$, and plan extraction becomes trivial). The stack may induce various plans since it results from executing in each $s \in L_i$ all actions, which are executable in s. Plan extraction cannot be done in forward direction like breadth first search, because the stack does not provide any information about, which action is selected next in order to reach a goal. For this reason, plan extraction works in backward direction, starting from G. The algorithm searches for an action *act* which transforms a subset $S \subseteq L_{z-1}$ into a subset of G. Action *act* will be the last action executed by our plan. Afterwards, if $z \geq 2$, then an action to be executed just before *act* is determined by finding an action that transforms a subset $S' \subseteq L_{z-2}$ into a subset of S. More precisely, plan extraction considers pairs (L_{i-1}, L_i) of layers and pairs (S_{i-1}, S_i) with $S_{i-1} \subseteq L_{i-1}$ and $S_i \subseteq L_i$ $(i = z, \ldots, 1)$. We choose $S_z := G$. S_{i-1} is obtained from S_i by first finding an automaton $A_i \in \mathcal{A}$ with $\emptyset \neq B_{i-1} := \mathit{exp}_{\mathit{curr}}(L_{i-1}) \cap \mathit{exp}_{\mathit{succ}}(S_i) \cap A_i$. $\mathit{exp}_{\mathit{curr}}(L_{i-1})$ is the set of all transitions $t = (s_1, s_2)$ with $s_1 \in L_{i-1}$ and s_2 being an arbitrary planning state. $\mathit{exp}_{\mathit{succ}}(S_i)$ is the set of all transitions $t = (s_1, s_2)$ with s_1 being an arbitrary planning state and $s_2 \in S_i$. Thus, B_{i-1} is the set of all transitions $t = (s_1, s_2)$ with $s_1 \in L_{i-1}$, $s_2 \in S_i$ and $t \in A_i$. In $S_{i-1} := \mathit{proj}_{\mathit{curr}}(B_{i-1})$ only the current states remain. S_{i-1} is the set of all predecessors of S_i under $\mathit{action}(A_i)$, the action that corresponds to A_i. $\mathit{action}(A_i)$ will be the i-th action executed by the plan. Algorithm 2 summarizes the ideas described. The validity of the plan extracted by the algorithm results from $S_0 = \mathcal{I}$, $S_z = \mathcal{G}$ and the fact that, for $i = 1, \ldots, z$, $\mathit{action}(A_i)$ transforms a state $s_{i-1} \in S_{i-1}$ into a state $s_i \in S_i$.

Algorithm 2 Extract-Plan

Input: Representation $\mathcal{R} = (\mathcal{A}, \mathcal{I}, \mathcal{G})$, a stack *stack*
Output: Sequential plan of length $|stack| - 1$

```
plan := []; post := stack->pop()
while stack ≠ ∅ do
    pre := stack->pop(); post' := ⊥
    while post' = ⊥ do
        a := nextA ∈ A
        post' := exp_curr(pre) ∩ exp_succ(post) ∩ A
        post' := proj_curr(post')
    post := post'
    plan := [action(a)].plan
return plan
```

4.3 Solving Metric Problems

An extension of algorithm 1 is also able to solve metric problems (cf. Algorithm 3). For metric problems, the representation $\mathcal{R}$ includes an additional component $\mathcal{M}$ which represents the metric, i.e., $\mathcal{R}$ now is a 4-tuple $(\mathcal{A}, \mathcal{I}, \mathcal{G}, \mathcal{M})$. $\mathcal{M}$ is a linear expression $a_1 z_1 + \cdots + a_d z_d$ with pairwise different variables $z_i \in \{x_1, \ldots, x_n\}$, $a_1, \ldots, a_d \in \mathbb{Z} \backslash \{0\}$ and $d \geq 1$. Since we permit the use of rational numbers and divisions in metrics, the original metric must be scaled by a suitable factor. If the original metric is to be maximized, we turn the maximization problem into a minimization problem by multiplying the metric with -1. If a summand $c \cdot$`total-time` occurs in the simplified scaled metric, then this summand will be removed, and we keep c in memory for later use (setting $c := 0$ if no such summand occurs). Minimizing $\mathcal{M}$ is equivalent to optimizing the original metric, but $\mathcal{M}$ is compatible to Presburger arithmetic. Given a value $\mathcal{M}$ takes, it is easy to calculate the corresponding value of the original metric. For *upperValue* $\in \mathbb{Z}$ let *boundAut*$(\mathcal{M}, \textit{upperValue})$ be the minimal DFA for $\mathcal{M} \leq$ *upperValue*, using the alphabet Σ_{state}. Define *boundAut*$(\mathcal{M}, \infty) := \top$.

Again, the planning graph is explored using breadth first search. The `while` loop now uses $true$ as its guard because hitting the goal set does not exclude that further hits in following layers would lead to a smaller value for the metric. The algorithm maintains the value *bound* the metric $\mathcal{M}$ takes for the best goal state found so far, with initially *bound* $:= \infty$.

As long as the exploration has not found a goal state yet, each layer L_i is intersected with $\mathcal{G}$. If the intersection B is not empty, then B may contain various goal states, where each goal state makes the metric take a certain value. We are interested in the smallest value the metric takes for some $s \in B$ (a smallest value exists since B is finite). For a finite set $S \neq \emptyset$ of planning states and a metric $\mathcal{M}$, let *minimalValue*$(\mathcal{M}, S)$ be the smallest value $\mathcal{M}$ takes for some state in S. To compute *minimalValue*$(\mathcal{M}, S)$, first an upper bound u is found by evaluating the metric for an arbitrary $s \in S$ and then intersecting S with *boundAut*$(\mathcal{M}, u - e \cdot 2^i)$ for $i = 0, 1, 2, 3, \ldots$ and some integer $e \geq 1$, until the intersection is empty. The last $u - e \cdot 2^i$ is a lower bound for *minimalValue*$(\mathcal{M}, S)$, and the exact value can be determined using a binary search approach. We

update $bound$ with *minimalValue*$(\mathcal{M}, B)$ and copy the stack, since solution reconstruction will need the stack as it is now, if exploration terminates without finding a better goal state.

After the goal state has been hit at least once, whenever a layer L_i is completed, the algorithm computes $B := L_i \cap \mathcal{G} \cap$ *boundAut*$(\mathcal{M}, bound - timeterm - 1)$. We subtract 1 because we would like to know if L_i contains a goal state which reaches a metric value that is smaller than the one we are already able to realize. But this alone would not be correct, as the metric actually contains the summand $c \cdot$ `total-time`, which has increased by $c \cdot (i - j)$ since the last update to *bound*, if the last update was done when examining layer L_j. The algorithm guarantees that, at this point of computation, *timeterm* is equal to $c \cdot (i - j)$. Thus, *bound* $-$ *timeterm* is the value $\mathcal{M}$ must take now in order to let the original metric take the same value as before. If B is not empty, the algorithm proceeds as described above. Exploration terminates if, and only if, $L_i = \emptyset$ for some i. At any time, it is possible to stop the algorithm and extract a plan from the stack saved when the latest update to *bound* was performed.

Algorithm 3 Metric-Breadth-First-Search

Input: $\mathcal{R} = (\mathcal{A}, \mathcal{I}, \mathcal{G}, \mathcal{M})$, coefficient c of `total-time`
Output: Shortest seq. plan minimizing $\mathcal{M}$ or 'no plan'

```
stack := stack' := ∅; layer := reached := I
bound := ∞; timeterm := 0; stack->push(I)
while true do
  inter := layer ∩ G ∩ boundAut(M, bound − timeterm − 1)
  if inter ≠ ⊥ then
    bound := minimalValue(M, inter); timeterm := 0
    stack' := stack; stack'->pop()
    stack'->push(inter ∩ boundAut(M, bound))
  timeterm := timeterm + c
  for all A ∈ A do
    trans := exp_curr(layer) ∩ A
    layer' := layer' ∪ proj_succ(trans)
  layer := layer'\reached
  if layer = ⊥ then
    if bound = ∞ then
      return 'no plan'
    else
      return Extract-Plan((A, I, G), stack')
  else
    reached := reached ∪ layer'
    stack->push(layer)
```

5 The Triple-A Planning System (TTAPS)

We have implemented our approach in *Java*, adapting an already existing automata library. Our tool TTAPS (for *The Triple-A Planning System*) has full PDDL2.2 functionality up to the restrictions $\mathbf{R1} - \mathbf{R7}$, but its performance has not yet left the status of a prototype.

TTAPS comes as an executable *Java* archive, including sample PDDL-files Automata currently use bitvector alphabets as described above. For each pair (q_1, q_2) of automaton states q_1 and q_2 with at least one transition from q_1 to q_2, there is exactly one object that represents all transitions from q_1 to q_2. A BDD characterizes the set of labels that belong to one of those transitions. Our BDDs do not share structures, so there is space for optimization, too. But another, very promising approach is to make $\{0, 1\}$ the alphabet of all automata, and let automata read each vector serially instead of parallelism. This approach can also be combined with another encoding of integers leading to asymptotic smaller automata in certain cases [1].

In the appendix, we give example domains and problems we have designed as test instances the current planner prototype is able to solve.

6 Conclusion

The paper proposes an optimal planner in Presburger arithmetic for metric problems. It contributes to the area of planning via model checking [12]. The expressiveness covers a large part of current PDDL. The memory problems that are inherent to representing large sets of states are addressed by using a symbolic representation based on minimized (and therefore unique) automata. As a feature, the approach can deal with infinite state sets.

The paper is the first report on optimal plan finding in infinite state numerical domains. It implements a planner on top of the existing library AAA[2]. All other related planners are non-optimal and provide no guarantee on optimal plan finding. Previous work on metric action planning is based on representing the states explicitly. For example, Metric-FF [15] is a forward-state heuristic search planner that uses an involved Graphplan inspired heuristics to guide the search process. LPG [11] is a local search planner that gradually improves an initial invalid plan resolving conflicts by inserting and deleting actions, until it eventually becomes sound. SGPlan [6] also uses local search and partitions the overall problem into tightly connected subproblems. Until a plan is found, the planner first resolves local constraint violations and then global constraint violations.

There is recent work on SAT encodings of state-space planning in numeric domains [17]. The approach proposed in this paper first infers a finite state encoding using an approximate fixpoint analysis, and then uses the finite-state variables to infer the domain. The approach is fast, provides optimality guarantees, and applies well to current benchmark domains. However, as it relies on finite domain encodings of numbers, it is less general than the approach presented here.

In future work we will work on performance improvements in the exploration and might try tackling real numbers using the results of [4].

[2] texttthttp://sourceforge.net/projects/triple-a

References

[1] C. Bartzis and T. Bultan. Efficient symbolic representations for arithmetic constraints in verification. *Int. J. Found. Comput. Sci.*, 14(4):605–624, 2003.

[2] P. Bertoli, A. Cimatti, and M. Roveri. Heuristic search symbolic model checking = efficient conformant planning. In *IJCAI*, pages 467–472, 2001.

[3] A. Blum and M. L. Furst. Fast planning through planning graph analysis. In *IJCAI*, pages 1636–1642, 1995.

[4] B. Boigelot, S. Jodogne, and P. Wolper. An effective decision procedure for linear arithmetic over the integers and reals. *ACM Transaction on Computational Logic*, 6:614–633, 2005.

[5] R. E. Bryant. Symbolic manipulation of boolean functions using a graphical representation. In *DAC*, pages 688–694, 1985.

[6] Y. Chen and B. W. Wah. Subgoal partitioning and resolution in planning. In *Proceedings of the International Planning Competition*, 2004.

[7] S. Edelkamp. Cost-optimal symbolic planning with state trajectory and preference constraints. In *ECAI*, 2006.

[8] M. Fischer and M. Rabin. Super-exponential complexity of Presburger arithmetic. In *SIAM-AMS*, 1974.

[9] M. Fox and D. Long. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20:61–124, 2003.

[10] A. Gerevini and D. Long. Plan constraints and preferences in PDDL3. Technical report, Department of Electronics for Automation, University of Brescia, 2005.

[11] A. Gerevini and I. Serina. Fast plan adaptation through planning graphs: Local and systematic search techniques. In *AIPS*, pages 112–121, 2000.

[12] F. Giunchiglia and P. Traverso. Planning as model checking. In *ECP*, pages 1–19, 1999.

[13] M. Helmert. Decidability and undecidability results for planning with numerical state variables. In *AIPS*, pages 303–312, 2002.

[14] M. Helmert. A planning heuristic based on causal graph analysis. In *ICAPS*, pages 161–170, 2004.

[15] J. Hoffmann. The Metric FF planning system: Translating "Ignoring the delete list" to numerical state variables. *Journal of Artificial Intelligence Research*, 20:291–341, 2003.

[16] J. Hoffmann and S. Edelkamp. The deterministic part of IPC-4: An overview. *Journal of Artificial Intelligence Research*, 24:519–579, 2005.

[17] J. Hoffmann, C. P. Gomes, B. Selman, and H. A. Kautz. Sat encodings of state-space reachability problems in numeric domains. In *IJCAI*, pages 1918–1923, 2007.

[18] H. Kautz and B. Selman. Pushing the envelope: Planning propositional logic, and stochastic search. In *ECAI*, pages 1194–1201, 1996.

[19] F. Klaedtke. On the automata size for Presburger arithmetic. In *LICS*, pages 110–119, 2004.

[20] P. Wolper and B. Boigelot. On the construction of automata from linear arithmetic constraints. In *TACAS*, pages 1–19, 2000.

[21] R. Zhou and E. Hansen. Breadth-first heuristic search. In *ICAPS*, pages 92–100, 2004.

7 Appendix

The planner is completely controlled via its Graphical User Interface (GUI).

7.1 Graphical User Interface

User interaction is mainly based on tabs. Initially, Triple-A offers tabs that ask the user for a domain, a problem, information about groups of mutual exclusive predicates or parameters. Results will be presented in additional *tabs*:

- The *Domain* tab is the place where the user can give a PDDL2.2 domain. Domains can be loaded and saved using the appropriate buttons. The domain must be fully-instantiated, and it must consider all restrictions imposed in the paper. Planning can be started by clicking *Solve Problem*.
- In the *Problem* tab the user can give a PDDL2.2 problem. Problems can be loaded and saved using the appropriate buttons. The problem must be fully-instantiated, and it must consider all restrictions imposed in the paper. Planning can be started by clicking "Solve Problem".
- The *Parameters* panel allows the user to choose the values for some parameters which affect the planning process. They are shown in Table 1.
- In the *Groups* tab, the user provides the partitioning for a mutex group encoding. Each group is given by a list of its members, where each member is in its own line, and there are no empty lines between two members. Groups are separated from each other by exactly one empty line. Predicates may be paranthesized, but no parantheses are required. Before the first group and after the last group, an arbitrary amount of whitespace characters (line feeds, spaces) is tolerated.
- The *Progress* tab is created when the planning process is started. It displays a log which explains what the system has done at which time. The log is updated in real time, so the user is always able to see what the system is doing at the moment. Information about the sizes of the automata is included, too. Exploration can be aborted by pressing *a* (but the planner completes the current layer).
- This *Result* tab is created as soon as the planning process terminates. If a plan was found, the plan is displayed. *Result* also allows the user to display automata and formulae representing the input (in separated tabs).

7.2 Propositional Planning Domains

7.2.1 Domain `abcd`

Our first example domain is a purely propositional domain. We directly consider the fully-instantiated version. There are two persons and four places. For each person-place combination

the domain specifies a predicate. Both persons can move freely between PLACEa and PLACEb, between PLACEb and PLACEc as well as between PLACEc and PLACEd, but only both persons together can directly move from PLACEa to PLACEd.

```
(define (domain abcd)
(:requirements :typing :fluents )

(:predicates
(at_PERSON1_PLACEa) (at_PERSON1_PLACEb)
(at_PERSON1_PLACEc) (at_PERSON1_PLACEd)
(at_PERSON2_PLACEa) (at_PERSON2_PLACEb)
(at_PERSON2_PLACEc) (at_PERSON2_PLACEd))

(:action FROM_TO-PLACEa-PLACEd-PERSON1-PERSON2
:parameters()
:precondition (and (at_PERSON1_PLACEa)(at_PERSON2_PLACEa))
:effect (and (not (at_PERSON1_PLACEa)) (not (at_PERSON2_PLACEa))
             (at_PERSON1_PLACEd) (at_PERSON2_PLACEd)))

(:action FROM_TO-PLACEa-PLACEb-PERSON1
:parameters()
:precondition (at_PERSON1_PLACEa)
:effect (and (not (at_PERSON1_PLACEa)) (at_PERSON1_PLACEb)))

(:action FROM_TO-PLACEb-PLACEa-PERSON1
:parameters()
:precondition (at_PERSON1_PLACEb)
:effect (and (not (at_PERSON1_PLACEb)) (at_PERSON1_PLACEa)))

(:action FROM_TO-PLACEb-PLACEc-PERSON1
:parameters()
:precondition (at_PERSON1_PLACEb)
:effect (and (not (at_PERSON1_PLACEb)) (at_PERSON1_PLACEc)))

(:action FROM_TO-PLACEc-PLACEb-PERSON1
:parameters()
:precondition (at_PERSON1_PLACEc)
:effect (and (not (at_PERSON1_PLACEc)) (at_PERSON1_PLACEb)))

(:action FROM_TO-PLACEc-PLACEd-PERSON1
:parameters()
:precondition (at_PERSON1_PLACEc)
:effect (and (not (at_PERSON1_PLACEc)) (at_PERSON1_PLACEd)))
```

```
(:action FROM_TO-PLACEd-PLACEc-PERSON1
:parameters()
:precondition (at_PERSON1_PLACEd)
:effect (and (not (at_PERSON1_PLACEd)) (at_PERSON1_PLACEc)))

(:action FROM_TO-PLACEa-PLACEb-PERSON2
:parameters()
:precondition (at_PERSON2_PLACEa)
:effect (and (not (at_PERSON2_PLACEa)) (at_PERSON2_PLACEb)))

(:action FROM_TO-PLACEb-PLACEa-PERSON2
:parameters()
:precondition (at_PERSON2_PLACEb)
:effect (and (not (at_PERSON2_PLACEb)) (at_PERSON2_PLACEa)))

(:action FROM_TO-PLACEb-PLACEc-PERSON2
:parameters()
:precondition (at_PERSON2_PLACEb)
:effect (and (not (at_PERSON2_PLACEb)) (at_PERSON2_PLACEc)))

(:action FROM_TO-PLACEc-PLACEb-PERSON2
:parameters()
:precondition (at_PERSON2_PLACEc)
:effect (and (not (at_PERSON2_PLACEc)) (at_PERSON2_PLACEb)))

(:action FROM_TO-PLACEc-PLACEd-PERSON2
:parameters()
:precondition
(at_PERSON2_PLACEc)
:effect (and (not (at_PERSON2_PLACEc)) (at_PERSON2_PLACEd)))

(:action FROM_TO-PLACEd-PLACEc-PERSON2
:parameters()
:precondition (at_PERSON2_PLACEd)
:effect (and (not (at_PERSON2_PLACEd)) (at_PERSON2_PLACEc))))
```

7.2.2 Problem `from-a-and-c-to-d`

The problem makes `PLACEa` the initial place for `PERSON1`, and `PLACEc` the initial place for `PERSON2`. The goal is to move both persons to `PLACEd`:

```
(define (problem from-a-and-c-to-d)
```

```
(:domain abcd)
(:init (at_PERSON1_PLACEa) (at_PERSON2_PLACEc))
(:goal (and (at_PERSON1_PLACEd) (at_PERSON2_PLACEd))))
```

7.2.3 Solving `from-a-and-c-to-d` (Binary Encoding)

We leave all parameters on their default values (Trigger type = *OBDD*, Variable ordering = *Interleave current/successor*, Logical state encoding = *Binary Encoding*, Automaton management = *Internal*). Then we click *Solve Problem* in either the domain tab or the problem tab. The *Progress* tab appears, monitoring the progress of the planning process. The *Result* tab provides the following output:

```
Makespan:           3
Optimality:         PROVED

0: (FROM_TO PLACEc PLACEb PERSON2) [1]
1: (FROM_TO PLACEb PLACEa PERSON2) [1]
2: (FROM_TO PLACEa PLACEd PERSON1 PERSON2) [1]
```

It is easy to see that this plan is optimal.

7.2.4 Solving `from-a-and-c-to-d` (Mutex Group Encoding)

Now we perform the same task, but with a mutex group encoding. We choose *Mutex Group Encoding* in the *Parameters* tab and copy the content of the domain's `:predicates` block into the *Groups* tab, and separate the first four predicates (describing the location of `PERSON1`) from the last four predicates (describing the location of `PERSON2`) by exactly one empty line. Afterwards, we start the planning process. The resulting plan is the same as before, but computation time is approximately one fourth of the computation time for the binary encoding. We can review the formulae and automata constructed for the domain and the problem by clicking the appropriate link and selecting the corresponding tab.

8 Derived Predicates

Next we rename `abcd` to `abcd-derived` and insert the following `:derived` definitions:

```
(:derived (OCCUPIED_PLACEb)
    (or (at_PERSON1_PLACEb) (at_PERSON2_PLACEb)))
(:derived (OCCUPIED_PLACEd)
    (or (at_PERSON1_PLACEd) (at_PERSON2_PLACEd)))
(:derived (DOUBLY_OCCUPIED_PLACEd)
    (and (at_PERSON1_PLACEd) (at_PERSON2_PLACEd)))
(:derived (GOAL)
```

```
    (and (OCCUPIED_PLACEb) (OCCUPIED_PLACEd)))
(:derived (GOAL)
    (DOUBLY_OCCUPIED_PLACEd))
```

8.0.5 Problem `travel-derived`

The problem `travel-derived` suits to the domain `abcd-derived`:

```
(define (problem travel-derived)
(:domain abcd-derived)
(:init (at_PERSON1_PLACEa) (at_PERSON2_PLACEc))
(:goal (GOAL)))
```

The shortest plan lets `PERSON1` travel from `PLACEa` to `PLACEb`, and it lets `PERSON2` travel from `PLACEc` to `PLACEd`. If we modify the problem such that both persons are initially at `PLACEa`, then the planner will send both persons to `PLACEd`.

8.1 Metric Planning

8.1.1 Domain `metric-abcd`

Now let us consider a metric variant of `abcd`. `PERSON1` and `PERSON2` are able to do the same movements as before, but now there is a distance between each pair of places. The total distance both persons have left behind, is modeled by the function `totaldistance`. Each action increases `totaldistance` by the distance it makes the person leave behind. The distance does not depend on the person or the direction. The distance between `PLACEa` and `PLACEb` is 10. The distance between `PLACEb` and `PLACEc` is 10, too, whereas the distance between `PLACEc` and `PLACEd` is 19. The journey from `PLACEa` to `PLACEd` has a length of 40.

```
(define (domain metric-abcd)
(:requirements :typing :fluents)

(:predicates
(at_PERSON1_PLACEa) (at_PERSON1_PLACEb)
(at_PERSON1_PLACEc) (at_PERSON1_PLACEd)
(at_PERSON2_PLACEa) (at_PERSON2_PLACEb)
(at_PERSON2_PLACEc) (at_PERSON2_PLACEd))
(:functions (totaldistance))

(:action FROM_TO-PLACEa-PLACEd-PERSON1-PERSON2
:parameters()
:precondition (and (at_PERSON1_PLACEa) (at_PERSON2_PLACEa))
:effect (and (increase totaldistance 40)
             (not (at_PERSON1_PLACEa)) (not (at_PERSON2_PLACEa))
```

```
            (at_PERSON1_PLACEd) (at_PERSON2_PLACEd)))

(:action FROM_TO-PLACEa-PLACEb-PERSON1
:parameters()
:precondition (at_PERSON1_PLACEa)
:effect (and (increase totaldistance 10)
            (not (at_PERSON1_PLACEa)) (at_PERSON1_PLACEb)))

(:action FROM_TO-PLACEb-PLACEa-PERSON1
:parameters()
:precondition (at_PERSON1_PLACEb)
:effect (and (increase totaldistance 10)
            (not (at_PERSON1_PLACEb)) (at_PERSON1_PLACEa)))

(:action FROM_TO-PLACEb-PLACEc-PERSON1
:parameters()
:precondition (at_PERSON1_PLACEb)
:effect (and (increase totaldistance 10)
            (not (at_PERSON1_PLACEb)) (at_PERSON1_PLACEc)))

(:action FROM_TO-PLACEc-PLACEb-PERSON1
:parameters()
:precondition (at_PERSON1_PLACEc)
:effect (and (increase totaldistance 10)
            (not (at_PERSON1_PLACEc)) (at_PERSON1_PLACEb)))

(:action FROM_TO-PLACEc-PLACEd-PERSON1
:parameters()
:precondition (at_PERSON1_PLACEc)
:effect (and (increase totaldistance 19)
            (not (at_PERSON1_PLACEc)) (at_PERSON1_PLACEd)))

(:action FROM_TO-PLACEd-PLACEc-PERSON1
:parameters()
:precondition (at_PERSON1_PLACEd)
:effect (and (increase totaldistance 19)
            (not (at_PERSON1_PLACEd)) (at_PERSON1_PLACEc)))

(:action FROM_TO-PLACEa-PLACEb-PERSON2
:parameters()
:precondition (at_PERSON2_PLACEa)
:effect (and (increase totaldistance 10)
            (not (at_PERSON2_PLACEa)) (at_PERSON2_PLACEb)))
```

```
(:action FROM_TO-PLACEb-PLACEa-PERSON2
:parameters()
:precondition (at_PERSON2_PLACEb)
:effect (and (increase totaldistance 10)
             (not (at_PERSON2_PLACEb)) (at_PERSON2_PLACEa)))

(:action FROM_TO-PLACEb-PLACEc-PERSON2
:parameters()
:precondition (at_PERSON2_PLACEb)
:effect (and (increase totaldistance 10)
             (not (at_PERSON2_PLACEb)) (at_PERSON2_PLACEc)))

(:action FROM_TO-PLACEc-PLACEb-PERSON2
:parameters()
:precondition (at_PERSON2_PLACEc)
:effect (and (increase totaldistance 10)
             (not (at_PERSON2_PLACEc)) (at_PERSON2_PLACEb)))

(:action FROM_TO-PLACEc-PLACEd-PERSON2
:parameters()
:precondition (at_PERSON2_PLACEc)
:effect (and (increase totaldistance 19)
             (not (at_PERSON2_PLACEc)) (at_PERSON2_PLACEd)))

(:action FROM_TO-PLACEd-PLACEc-PERSON2
:parameters()
:precondition (at_PERSON2_PLACEd)
:effect (and (increase totaldistance 19)
             (not (at_PERSON2_PLACEd)) (at_PERSON2_PLACEc))))
```

8.1.2 Problem `metric-from-a-and-c-to-d`

Now we adapt problem `from-a-and-c-to-d` such that `totaldistance` is minimized:

```
(define (problem metric-from-a-and-c-to-d)
(:domain metric-abcd)
(:init (at_PERSON1_PLACEa) (at_PERSON2_PLACEc))
(:goal (and (at_PERSON1_PLACEd) (at_PERSON2_PLACEd)))
(:metric minimize totaldistance))
```

In order to avoid a heap overflow, we select *CompactOBDD* for *Trigger type*. As this is an infinite state problem, the exploration will never terminate itself. Please assign 4 to *Abort BFS*

after level. We ensure that the mutex group encoding we used above is still configured and start the planning process. On a Pentium 4 with 2.66 GHz and 1.25 GB RAM, planning took more than 20 minutes. The exploration first finds the solution which was the output for the non-metric variant. This solution has a metric value of 60. Later, the planner will find an optimal solution with a *makespan* of 4, but a metric value of 58:

```
0: (FROM_TO PLACEa PLACEb PERSON1) [1]
1: (FROM_TO PLACEb PLACEc PERSON1) [1]
2: (FROM_TO PLACEc PLACEd PERSON2) [1]
3: (FROM_TO PLACEc PLACEd PERSON1) [1]
```

8.1.3 Domain `uv`

```
(define (domain uv)
(:functions (u) (v))

(:action SET_u_4
:parameters()
:effect (assign u 4))

(:action SET_u_5
:parameters()
:effect (assign u 5))

(:action SET_v_3
:parameters()
:effect (assign v 3))

(:action SET_v_m7
:parameters()
:effect (assign v -7)))
```

8.1.4 Problem `set-uv`

```
(define (problem set-uv)
(:domain uv)
(:init (= u 0) (= v 0))
(:goal (not (= u 0)))
(:metric minimize (- (* -7 u) (* -3 v) )))
```

8.1.5 Problem `set-uv-total-time`

```
(define (problem set-uv-total-time)
(:domain uv)
```

```
(:init (= u 0) (= v 0))
(:goal (not (= u 0)))
(:metric minimize (+ (- (* -7 u) (* -3 v)) (* 100 total-time))))
```

8.1.6 Problem `set-uv-complex`

```
(define (problem set-uv-complex)
(:domain uv)
(:init (= u 0) (= v 0))
(:goal (not (= u 0)))
(:metric minimize (+ 301 (+ (- (* -10.5 u) (* -4.5 v))
                            (* 150 total-time)))))
```

8.1.7 Domain `lindomain`

```
(define (domain lindomain)
(:functions (x) (y) (z))

(:action calc_z
:parameters ()
:precondition (= z 0)
:effect (assign z (- (* 3 x) (* 7 y)))))
```

8.1.8 Problem `linproblem`

```
(define (problem linproblem)
(:domain lindomain)
(:init (= x 20) (= y 40) (= z 0))
(:goal (and (= z -220) (= y 40) (= x 20))))
```

8.2 `:multi-init` and Multi-Plans

To exploit the fact that Presburger arithmetics can also deal with infinite state sets, we allow the use of an arbitrary goal description in the `:init` section of a problem (the problem must specify the `:multi-init` requirement that extends PDDL). It may be natural to characterize several or infinite many initial states instead of exactly one initial state, if there is uncertainty or no knowledge about what is the initial state. In this case, the goal description only describes all aspects that are certain or known. If a plan is found, then the plan extraction algorithm has also found a subset of the initial state set, with the property that the plan transforms each member into a goal state. The process is restarted, where the subset is excluded from the set of initial states.

8.2.1 Domain `x-plus-17`

```
(define (domain x-plus-17)
```

```
(:functions (x))

(:action INC
:parameters()
:precondition (< x 20)
:effect (increase x 17)))
```

8.2.2 Problem `x-plus-17-problem`

```
(define (problem x-plus-17-problem)
(:domain x-plus-17)
(:requirements :multi-init)
(:init (and (>= x -17) (<= x 3)))
(:goal (and (>= x 0) (<= x 20)))
(:metric minimize x))
```

Table 1: Parameter Description

Type of triggers to be used by the automata	The structure used to represent the set of all bitvectors that are the label of some transition from a certain automaton state q1 to a certain automaton state q2. *OBDD* is the default. *CompactOBDD* is an OBDD implementation which needs much less memory than OBDD. It can prevent an automaton or a set of automata from overtaxing main memory, but it increases computation time because CompactOBDD operations are realized by converting the operands to OBDDs, performing the corresponding OBDD operation and then converting the result back to a CompactOBDD.
Variable ordering to be observed by alphabets	Controls the bijective mapping between bitvector components and variables. The variable ordering is irrelevant to the automaton itself, but it is important for the OBDDs/CompactOBDDs, as they observe the variable ordering.
Logical state encoding	Allows the selection of one of the three predicate encodings. For the prime encoding, the parameters *Preferred group size* and *Consider frequencies of predicates* are relevant, too. If the mutex encoding is chosen, then the *Groups* tab must provide a partitioning of the whole predicate set, where all predicates contained in the same block are mutual exclusive.
Preferred group size	The maximal size groups can have, if a prime encoding is applied. Consider frequencies of predicates Only relevant to prime encodings. If checked, then predicates which occur often will be assigned to lower prime numbers than predicates which occur seldom. Abort BFS after level The maximal number of layers to be explored by breadth first search, i.e. this is an additional termination criterion.
Ignore `:metric` block	If checked, metric problems will be treated as non-metric problems. Character used to separate parameters In fully-instantiated domains, action names should begin with the name of the corresponding original action, followed by the parameters the original action has been instantiated with, separated by a certain character which can be chosen here. The knowledge about the separator is needed for ungrounding, when representing a plan as a string.
Automaton management	Select *Internal* if all automata should be kept in main memory. Select *External* if all automata should be swapped to the disk (does not work in the `.jar` version!). Position for automaton files Directory for the automata swapped to disk, if *External* has been selected for *Automaton management*.