Optimizing Planning Domains by Automatic Action Schema Splitting

Carlos Areces and **Facundo Bustos** and **Martín Dominguez**
Universidad Nacional de Córdoba
Córdoba, Argentina
{carlos.areces, facundojosebustos}@gmail.com, mdoming@famaf.unc.edu.ar

Jörg Hoffmann
Saarland University
Saarbrücken, Germany
hoffmann@cs.uni-saarland.de

Abstract

As modeling details can have a large impact on planner performance, domain transformation has been a traditional subject of interest in the planning community not only between languages, but also within languages. Herein, we automate an intra-language transformation method that has as yet been applied only manually, and that has never been formally described: *action schema splitting*, which transforms an action schema with a big interface (many parameters) into several schemas with smaller interfaces, exponentially reducing the number of ground actions. We spell out this method, characterizing exactly the choice of splits preserving equivalence to the original schema. Making that choice involves a trade-off between interface size and plan length, which we explore by designing automatic domain optimization methods. Our experiments show that these methods can substantially improve performance on domains with large interfaces.

Introduction

Automatic domain transformation has been a topic of interest for a long time. There is a large number of works (e. g. (Gazen and Knoblock 1997; Nebel 2000; Palacios and Geffner 2009)) whose objective is to transform the planning task at hand into a simpler language easier to deal with. It is also often beneficial to transform a planning task *within* a given language, the objective being to improve planner performance through a more suitable model. For example, it has been considered to remove redundant actions (Haslum and Jonsson 2000) in order to reduce the branching factor, to (inversely) add additional redundant macro-actions in order to reduce distance-to-goal (e. g. (Botea et al. 2005; Newton et al. 2007)), and to simplify the task in ways proved to be sound using causal graphs and other kinds of analyses (e. g. (Knoblock 1994; Haslum 2007)). In this work, we focus on an intra-language transformation that has not, as yet, been systematically investigated: *action schema splitting*.

Given an action schema $a[X]$, i. e., a PDDL-like action with parameters (variables) X ranging over objects, take the *interface size* of $a[X]$ to be its number of parameters, $|X|$. The splitting operation creates several schemas $a_1[X_1], \dots, a_k[X_k]$ whose combination corresponds exactly to $a[X]$ in any valid plan, yet each of which has a smaller interface. The key advantage of such a split is the smaller number of ground actions. For example, if each $x \in X$ can be instantiated with 100 objects, $|X| = 3$, and $|X_i| = 1$, then we reduce that number from 1000000 to 300.

Action schema splitting has been used as an engineering method to make domains accessible to standard planners that ground out the actions, in the creation of the IPC Pipesworld and Cybersecurity domains (Hoffmann et al. 2006; Boddy et al. 2005), and in the formulation of genome edit distance as planning (Haslum 2011). This was done manually. The splitting method has never been formally described, and no attempt has been made to automate it. Our contribution is to fill these gaps.

We spell out formally what a valid action schema split is, devising a general translation method. Specifically, we show that, given a schema $a[X]$, one can choose any split $a_1[X_1], \dots, a_k[X_k]$ that preserves the intended order among potentially identical preconditions/adds/deletes in the original schema (e. g., preconditions need to be checked before corresponding deletes are applied, or else the split schema may not be applicable even though the original schema is). Choosing $a_1[X_1], \dots, a_k[X_k]$ constitutes a trade-off between minimizing interface size $\max_i |X_i|$ and thus the number of ground actions, vs. minimizing *split size* k and, therewith, plan length. We design automatic domain optimization techniques addressing that trade-off. We evaluate our methods on (a) standard IPC benchmarks, as well as (b) "un-split" versions of the Pipesworld and genome edit distance. Our techniques (a) are typically not beneficial on the former domains, as IPC domains are already engineered to challenge search not pre-processes; however, our techniques (b) are beneficial, and sometimes even more so than the manually split domain versions, for the latter domains.

We next provide the background on our planning framework. We then introduce the action schema splitting operation. We devise automatic domain optimization methods, and evaluate these experimentally. We close the paper with a brief discussion of conclusions and future work.

Background

We focus on STRIPS-like planning domains. Action schema splitting takes place on the first-order (*lifted*) level, where actions are parameterized by object variables as in PDDL. We distinguish that level from the *propositional* level, as used in most current planner implementations, and where the planning semantics is defined. We will denote variables with x, y, z, and sets of variables with X, Y, Z.

Definition 1. *An* action schema $a[X]$ *is a 3-tuple* (P, A, D) *where* P *(the* precondition*),* A *(the* add list*), and* D *(the* delete list*) are finite sets of first-order atoms such that* X *is the set of variables that appear in* $P \cup A \cup D$. X *is called the* interface *of* $a[X]$*, and the variables themselves are often called the* parameters *of the action schema. We denote* $pre(a[X]) = P$*,* $add(a[X]) = A$*, and* $del(a[X]) = D$*. We denote* $At(a[X]) = P \cup A \cup D$.

An action*, or* ground action*,* a *is a 3-tuple* (P, A, D) *where* P*,* A*,* D *are finite sets of propositional symbols. We denote the* $pre(\cdot)$*,* $add(\cdot)$*,* $del(\cdot)$*, and* $At(\cdot)$ *functions exactly as for action schemas. If* A *is a set of action schemas or of actions, then* $At(A)$ *is defined in the obvious way.*

We always distinguish first-order constructs by notating them along with their variables, e. g. writing $l[X]$ for a first-order atom with variables X; like for action schemas, we refer to X as the *interface*. Instead of $a[\{x, y, z\}]$, we often write $a(x, y, z)$.

Action schemas can be *instantiated* by assigning values – *objects* – to their parameters, yielding actions. In that manner, action schemas represent sets of actions. Given a finite set of objects O and an action schema $a[X]$, the *instantiation* of $a[X]$ with O is the ground action a defined in the usual manner, substituting the variables of the first-order atoms in $a[X]$ with the objects assigned to the schema's parameters.

Example 1. *As an illustrating running example, we will consider the action schema moving block* x *from block* y *to block* z*. We can write this in STRIPS notation as:*

$$\begin{array}{l} \textbf{\textit{Move}}(x, y, z) \\ \quad pre: \{on(x, y), clear(x), clear(z)\} \\ \quad add: \{on(x, z), clear(y)\} \\ \quad del: \{on(x, y), clear(z)\} \end{array}$$

In our notation, this schema is represented as the triple $\textbf{\textit{Move}}(x, y, z) = (\{on(x, y), clear(x), clear(z)\}, \{on(x, z), clear(y)\}, \{on(x, y), clear(z)\})$*. Instantiating the variables* x*,* y *and* z *with objects* A, B, C *respectively, the first-order atoms are grounded, and we can consider them as propositional symbols. We write the corresponding (instantiated) action simply by replacing the parameters with objects, like "*$\textbf{\textit{Move}}(A, B, C)$*". In the present case,* $\textbf{\textit{Move}}(A, B, C) = (\{on(A, B), clear(A), clear(C)\}, \{on(A, C), clear(B)\}, \{on(A, B), clear(C)\})$*.*

In PDDL, planning domains are represented using a set of action schemas common to a set of planning instances giving the (finite) object set, initial state, and goal. Apart from the instantiation of an action schema as defined above, we don't require an explicit notation for this. We base our formalization simply on grounded STRIPS as used in state-of-the-art planners like, e. g., Fast Downward (Helmert 2006) and LAMA (Richter and Westphal 2010).

Definition 2. *A* planning task Π *is a 4-tuple* $\Pi = (P, A, I, G)$ *where* P *is a finite set of propositional symbols,* A *is a finite set of actions where* $At(A) \subseteq P$*,* $I \subseteq P$ *is the* initial state*, and* $G \subseteq P$ *is the* goal*.*

A state s *in* Π *is any set* $s \subseteq P$*. Action* a *is* applicable *to* s *if* $pre(a) \subseteq s$*. In that case, the outcome state* s' *of applying* a *to* s *is* $s' = (s \setminus del(a)) \cup add(a)$*, and we write* $s \xrightarrow{a} s'$*. For an actipon sequence* $\overline{a}$*, we write* $s \xrightarrow{\overline{a}} t$ *if the actions in* $\overline{a}$ *can be iteratively applied to* s*, resulting in* t*. A* plan *for* Π *is a sequence* $\overline{a}$ *such that* $I \xrightarrow{\overline{a}} s_G$ *where* $G \subseteq s_G$*. The plan is* optimal *if its length is minimal among all plans for* Π*.*

Note that we give adds a preference over deletes, i. e., if the same proposition p appears in both $del(a)$ and $add(a)$, then p is true after applying a. This complies with the official semantics of PDDL (Fox and Long 2003); our technology can trivially be adapted to deal with the opposite semantics if so desired. We do not consider action costs here, restricting ourselves to the uniform-cost case for simplicity, and as the most efficient planning systems (in terms of runtime) tend to use uniform costs.

Action Schema Splitting

We define *action schema splitting* as a syntactic transformation on action schemas. The transformation is designed such that the plans in a transformed planning task are in one-to-one correspondence with those in the original task. To illustrate the issues that must be tackled in achieving this, consider our example:

Example 2. *For the action schema* $\textbf{\textit{Move}}(x, y, z)$ *from Example 1, a tentative split into* sub-schemas *could be:*

$$\begin{array}{ll} \textbf{\textit{Move}}_1(x, y) & \textbf{\textit{Move}}_2(x, z) \\ \quad pre: \{on(x, y), clear(x)\} & \quad pre: \{clear(z)\} \\ \quad add: \{clear(y)\} & \quad add: \{on(x, z)\} \\ \quad del: \{on(x, y)\} & \quad del: \{clear(z)\} \end{array}$$

The correspondence of this split schema to the original one appears obvious, and one may be tempted to conclude that action schema splitting is trivial. However, note that the split shown is not actually valid:

(1) Nothing ensures that the two sub-schemas are instantiated consistently*, i. e., assign the same object to the shared parameter* x *on both sides.*

(2) Nothing ensures that the two sub-schemas are executed en block*, i. e., both together and without any other actions inserted in between. E.g., applying just* $\textbf{\textit{Move}}_2(x, z)$ *allows us to move any block* x *onto* z*, regardless of the current status of* x*.*

(3) Nothing ensures the intended order *among the unifiable add* $clear(y)$ *and precondition* $clear(z)$*: If* y *and* z *are instantiated with the same object, then, in any reachable state* s*, the original schema will not be applicable because we cannot have* $on(x, y)$ *and* $clear(y)$ *at the same time. In the split schema, however, the add of* $\textbf{\textit{Move}}_1(x, y)$ *will establish that atom, rendering* $\textbf{\textit{Move}}_2(x, z)$ *(and therewith the overall split schema) applicable.*

Issues (1) and (2) are easy to fix, for arbitrary splits, by decorating the sub-schemas with new atoms ensuring consistent instantiation and en-block execution. Issue (3) is more subtle, and is the only one restricting the set of splits we can choose from. We will now focus on issue (3), desining our splitting framework. Issues (1) and (2) will be handled below by augmenting that framework with the mentioned decorations.

Annotated Atoms and Sound Sequentializations

It is convenient to formulate our framework relative to atoms annotated with the part of the schema they belong to:

Definition 3. *Let $a[X]$ be an action schema, and let $l[Y] \in At(a[X])$ be an atom in $a[X]$. Then the corresponding* annotated atom *is the pair $(l[Y], f(\cdot))$ where $f(\cdot)$ is $pre(\cdot)$ in case $l[Y] \in pre(a[X])$, $add(\cdot)$ in case $l[Y] \in add(a[X])$, and $del(\cdot)$ in case $l[Y] \in del(a[X])$. The set of all annotated atoms of $a[X]$ is denoted $AnnAt(a[X])$.*

To avoid clutter, we will write $l_{pre}[Y]$ for annotated atoms from the precondition, and similar for adds/deletes. Annotated atoms are convenient because sub-schemas – schemas containing a part of the original schema – correspond to subsets of $AnnAt(a[X])$. In the remainder of the paper, we will typically identify schemas (and sub-schemas) with their set of annotated atoms. In particular, this is done in our definition of what a split is:[1]

Definition 4. *Let $A[Z]$ be a set of action schemas, and let $\mathcal{A}[Z]$ be the set of all possible action schemas over $At(A[Z])$. A* split function *for $A[Z]$ is a function $\sigma : A[Z] \mapsto \mathcal{P}(\mathcal{A}[Z])$, such that, whenever $\sigma(a[X]) = \{a_1[X_1], \dots, a_k[X_k]\}$, then $a_1[X_1], \dots, a_k[X_k]$ is a partition of $a[X]$, i. e., $a_i[X_i] \cap a_i[X_i] = \emptyset$ for $i \neq j$ and $\bigcup_i a_i[X_i] = a[X]$.*

To address issue (3), we need to define a partial order over $AnnAt(a[X])$. Valid splits are then ones that comply with that order. Namely, we need to make sure that, when the split schema is executed, preconditions are evaluated before adds so the latter cannot establish the former; preconditions are evaluated before deletes so the latter cannot disvalidate the former; and deletes are executed before adds so the latter get the desired preference over the former. Of course, we have to do all this only in case the two atoms in question might actually be identical in a ground action. In other words:

Definition 5. *Let $a[X]$ be an action schema, and let $l[Y], l[Y'] \in At(a[X])$ be atoms in $a[X]$ that share the same predicate l. We order $Ann(l[Y])$ before $Ann(l[Y'])$, written $Ann(l[Y]) \to Ann(l[Y'])$, if either: $l[Y] \in pre(a[X])$ and $l[Y'] \in add(a[X])$; or $l[Y] \in pre(a[X])$ and $l[Y'] \in del(a[X])$; or $l[Y] \in del(a[X])$ and $l[Y'] \in add(a[X])$.*

If $\sigma(a[X])$ is a split of $a[X]$, then a sequentialization *of the split is any linear ordering $a_1[X_1], \dots, a_k[X_k]$ of $\sigma(a[X])$. The sequentialization is* sound *if it complies with the ordering relation $\to$, i. e., whenever $Ann(l[Y]) \to Ann(l[Y'])$ and $Ann(l[Y]) \in a_i[X_i]$ and $Ann(l[Y]) \in a_j[X_j]$, then $i \leq j$.*

Note first that $\to$ is acyclic, and hence in principle always allows sound sequentialization:

[1]While each schema is split separately, we define the splitting operation for *sets* of schemas as the decorations addressing issues (1) and (2) will be shared across all schemas in a domain.

Lemma 1. *Let $a[X]$ be an action schema. Then the directed graph with vertices $AnnAt(a[X])$ and arcs $\to$ is acyclic.*

Proof. Direct from construction: The worst that can happen is for the same predicate to appear in precondition, adds, and deletes, in which case its precondition occurences are ordered before those in each of the adds and deletes, and its deletes occurences are ordered before those in the adds. $\square$

To understand these concepts, consider first the simple example where $a[X]$ has an empty precondition, adds $\{l(x)\}$, and deletes $\{l(y)\}$. Say the split separates the add atom from the delete atom. The original schema will always result in $l(x)$ being true. However, if we instantiate x and y with the same object and allow the deleting sub-schema to be applied last, then the split schema will result in $l(x)$ being false. In a sound sequentialization, this cannot happen, tackling issue (3) in controlling the splitting transformation and thus establishing its correctness relative to the original domain.[2]

Quotient Graphs and Valid Splits

We now establish a characterization of the set of sound sequentializations, which also immediately leads us to methods for actually finding them. For illustration:

Example 3. *For the action schema **Move**(x, y, z) from Example 1, the graph of partially ordered annotated atoms is:*

$$\begin{tikzcd}
clear_{pre}(z) \arrow[dr] \arrow[ddr] & & clear_{pre}(x) \arrow[dl] \arrow[ddl] & & on_{pre}(x,y) \arrow[dl] \arrow[dd] \\
& clear_{del}(z) \arrow[d] & & on_{del}(x,y) \arrow[dr] & \\
& clear_{add}(y) & & & on_{add}(x,z)
\end{tikzcd}$$

Intuitively, the tentative split in Example 2 does not comply with the ordering over the $clear(\cdot)$ predicate atoms, as the delete $clear(z)$ is ordered in between the precondition $clear(x)$ and the add $clear(y)$, but is ordered behind both in the split. We now make this intuition precise.

Definition 6. *Let $a[X]$ be an action schema, and let $\sigma(a[X])$ be a split of $a[X]$. Then the* quotient graph *is the directed graph whose vertices are the sub-schemas $a_i[X_i] \in \sigma(a[X])$, and that has an arc from $a_i[X_i]$ to $a_j[X_j]$ if there exist annotated atoms $Ann(l[Y]) \in a_i[X_i]$ and $Ann(l[Y']) \in a_j[X_j]$ such that $Ann(l[Y]) \to Ann(l[Y'])$. We say that $\sigma(a[X])$ is* valid *if the quotient graph is acyclic.*

Quotient graphs as defined here are equivalent to imposing an "abstraction" – aggregating vertices into "block vertices" – over the acyclic graph of annotated atoms (identified in Lemma 1). As we shall see, this simple concept achieves our aim of characterizing the set of sound sequentializations. For illustration, consider again our example:

[2]A subtle point here is that some planning approaches/tools disallow non-empty intersections between adds and deletes. Our assumption is that these, when given as input a ground action a not complying with their restriction, transform a so that it does comply (removing the duplicate atom from the delete list). Our split schemas produce ground actions equivalent to a (and thus also to the transformed action). Same if non-empty intersections between preconditions and adds are disallowed, and if both are disallowed.

Example 4. *The tentative split in Example 2 is not valid. Its quotient graph is: (we omit self-loops for simplicity)*

$$\begin{tikzcd}
\{on_{pre}(x,y), clear_{pre}(x), clear_{add}(y), on_{del}(x,y)\} \arrow[d, shift left] \\
\{clear_{pre}(z), on_{add}(x,z), clear_{del}(z)\} \arrow[u, shift left]
\end{tikzcd}$$

The downwards arc here results (amongst others) from the ordering $clear_{pre}(x) \rightarrow clear_{del}(z)$ (we need to make sure to not delete our own precondition), whereas the upwards arc results from the orderings $clear_{pre}(z) \rightarrow clear_{add}(y)$ and $clear_{del}(z) \rightarrow clear_{add}(y)$ (we need to make sure to not add our own precondition, and to not delete our add). A minimal way to get rid of the upwards arc, and thus of the cycle, is to separate out $clear_{add}(y)$:

$$\begin{tikzcd}
\{on_{pre}(x,y), clear_{pre}(x), on_{del}(x,y)\} \arrow[dr] \arrow[ddr] & \\
& \{clear_{pre}(z), on_{add}(x,z), clear_{del}(z)\} \arrow[dl] \\
\{clear_{add}(y)\} &
\end{tikzcd}$$

Note how this gets rid of issue (3) as observed in Example 2: If we instantiate y and z with the same object now, then the application of the split schema fails at the second sub-schema when the precondition $clear(z)$ comes up (because that atom is no longer added by the previous sub-schema).

As advertized, quotient graphs characterize exactly the sound sequentializations:

Lemma 2. *Let $a[X]$ be an action schema, let $\sigma(a[X])$ be a split, and let $a_1[X_1], \dots, a_k[X_k]$ be a sequentialization of $\sigma(a[X])$. Then $a_1[X_1], \dots, a_k[X_k]$ is sound if and only if it complies with the quotient graph, i. e., whenever that graph contains an arc from $a_i[X_i]$ to $a_j[X_j]$, then $i \leq j$.*

Proof. Direct from construction: As the block vertices in the quotient graph are the sub-schemas in the split, the ordering constraints imposed by soundness correspond exactly to the arcs in the quotient graph. □

As a split is valid iff its quotient graph is acyclic, which obviously is the case iff it is possible for a sequentialization to comply with that graph, we get:

Corollary 1. *Let $a[X]$ be an action schema, and let $\sigma(a[X])$ be a split. Then $\sigma(a[X])$ is valid if and only if it has at least one sound sequentialization.*

Finding Valid Splits

We have now identified exactly which splits can be chosen (namely, the valid ones). Remains the question, how to actually find such splits? Does there even always exist a non-trivial split, with more than a single sub-schema? Both questions are easily answered; we start with the latter one:

Definition 7. *Let $a[X]$ be an action schema. Then its* trivial split*, denoted $\mathit{TrivialSplit}(a[X])$, is the partition $\{a[X]\}$ that assigns every annotated atom to the same single sub-schema. Its* atom split*, denoted $\mathit{AtomSplit}(a[X])$, is the partition $\{\{\mathit{Ann}(l[Y])\} \mid \mathit{Ann}(l[Y]) \in a[X]\}$ that assigns every annotated atom to a separate sub-schema.*

Corollary 2. *Let $a[X]$ be an action schema. Then $\mathit{TrivialSplit}(a[X])$ and $\mathit{AtomSplit}(a[X])$ are both valid.*

This corollary is trivial for $\mathit{TrivialSplit}(a[X])$, and direct from Lemma 1 for $\mathit{AtomSplit}(a[X])$. As soon as the action schema contains more than a single atom, $\mathit{AtomSplit}(a[X])$ is non-trivial (contains more than one sub-schema). For illustration, reconsider Example 3: The atom split creates a separate sub-schema for each of the seven annotated atoms, and its quotient graph is exactly the one shown. In our simple example from above where $a[X]$ adds $\{l(x)\}$ and deletes $\{l(y)\}$, the atom split separates the add atom from the delete atom, and imposes that the delete is applied first, so that the only compliant sequentialization, like the original schema, always results in $l(x)$ being true.

Towards answering the question how to find more general valid splits, note first that splits naturally form a hierarchy (a partial order): We say that $\sigma(a[X])$ is *coarser than* $\sigma'(a[X])$ if $\sigma(a[X]) \neq \sigma'(a[X])$ and, for every $a_i[X_i] \in \sigma'(a[X])$, there exists $a_j[X_j] \in \sigma(a[X])$ such that $a_i[X_i] \subseteq a_j[X_j]$. The unique coarsest split is the trivial split, and the unique *finest* split (i. e., the least coarse one) is the atom split. We can travel between these two extremes by iteratively merging sub-schemas:

Definition 8. *Let $a[X]$ be an action schema, and let $\sigma(a[X]) = \{a_1[X_1], \dots, a_k[X_k]\}$ be a split. Denote by $\rightarrow^*$ the transitive closure over the arcs in the quotient graph. Then sub-schemas $a_i[X_i]$ and $a_j[X_j]$ are* mergeable *if there exists no $l \in \{1, \dots, k\} \setminus \{i, j\}$ where $a_l[X_l]$ is ordered between $a_i[X_i]$ and $a_j[X_j]$, i. e., where either $a_i[X_i] \rightarrow^* a_l[X_l] \rightarrow^* a_j[X_j]$ or $a_j[X_j] \rightarrow^* a_l[X_l] \rightarrow^* a_i[X_i]$. In that case, the* merged split *is the one that results from merging $a_i[X_i]$ and $a_j[X_j]$, i. e., replacing them with $a_i[X_i] \cup a_j[X_j]$ in $\sigma(a[X])$.*

Iterating such merging steps, starting from the atom split, underlies our search methods for domain optimization, described in the next section. This is suitable because:

Theorem 1. *Let $a[X]$ be an action schema. Then any split constructed by starting with $\mathit{AtomSplit}(a[X])$, and iteratively merging mergeable sub-schemas, is valid. Vice versa, any valid split can be constructed in this way.*

Proof. The first half of the claim follows because $\mathit{AtomSplit}(a[X])$ is valid (Corollary 2), and because if a split is valid then the merged split is valid as well. To show the latter, assume to the contrary that merging $a_i[X_i]$ and $a_j[X_j]$ introduces a cycle. Say $a_l[X_l]$ is any node on that cycle, different from $a_i[X_i] \cup a_j[X_j]$. Then there is a path from $a_i[X_i] \cup a_j[X_j]$ to $a_l[X_l]$, and a path from $a_l[X_l]$ to $a_i[X_i] \cup a_j[X_j]$. The "from" and "to" paths cannot both contact $a_i[X_i] \cup a_j[X_j]$ in $a_i[X_i]$ or else the previous split would have contained a cycle already; same for $a_j[X_j]$. Thus, in the original quotient graph, $a_l[X_l]$ must have been ordered between $a_i[X_i]$ and $a_j[X_j]$, in contradiction.

To see the second half of the claim, let $\sigma(a[X])$ be any valid split, and let $a_i[X_i] \in \sigma(a[X])$. Let the "basis" of $a_i[X_i]$ be those annotated atoms that are not transitively ordered by $\rightarrow$ behind any member of $a_i[X_i]$. Such atoms exist as $\rightarrow$ over the atoms is acyclic. The basis atoms are not transitively ordered relative to each other, so can be merged. All other atoms are reached via a $\rightarrow$ path $\vec{p}$ from at least one basis atom. As $\sigma(a[X])$ is valid, all atoms in between on $\vec{p}$ must be contained in $a_i[X_i]$ as well (else there would be a cycle), so we can iteratively merge-in all atoms on $\vec{p}$. □

Example 5. *The valid split in Example 4 (bottom figure) can be obtained from the atom split in Example 3 by iteratively merging $clear_{pre}(x)$ with $on_{pre}(x,y)$ (basis) with $on_{del}(x,y)$; and merging $clear_{pre}(z)$ with $on_{add}(x,z)$ (basis) with $clear_{del}(z)$.*

Decorating Splits, and Correctness

We have now clarified how to tackle issue (3) raised above in Example 2, but we have not yet done anything about issue (1), ensuring consistent parameter instantiation across the split schema, nor about issue (2), ensuring en-block execution across all split schemas in the domain. As advertized, both issues are easy to address by introducing artificial (new) atoms. We formalize this in terms of modifying the split function to decorate each sub-schema with these new atoms; the construction also fixes a sequentialization of the split:

Definition 9. *Let $A[Z]$ be a set of action schemas, and let σ be a split function for $A[Z]$ such that, for every $a[X] \in A[Z]$, $\sigma(a[X])$ is valid. We define the* decorated split function *$\tilde{\sigma}$ that, for each $a[X]$, selects an arbitrary sound sequentialization $a_1[X_1], \dots, a_k[X_k]$ of $\sigma(a[X])$, and decorates that sequentialization as follows:*

(i) ***Token procnone.*** *If $\sigma(a[X])$ has more than one sub-schema, then define $\tilde{\sigma}(a[X])$ adding the atom **procnone()** to $pre(a_1[X_1])$, $del(a_1[X_1])$ and $add(a_k[X_k])$. If $\sigma(a[X])$ has only one sub-schema, add **procnone()** to $pre(a_1[X_1])$.*

(ii) ***Token do.*** *Assuming a bijective function $id : A[Z] \mapsto \{1, \dots, |A[Z]|\}$, if $\sigma(a[X])$ has more than one sub-schema, then define $\tilde{\sigma}(a[X])$ adding the atoms $\boldsymbol{do}_2^{id(a[X])}$ to $add(a_1[X_1])$; $\boldsymbol{do}_j^{id(a[X])}$ to $pre(a_j[X_j])$ and $del(a_j[X_j])$ and $\boldsymbol{do}_{j+1}^{id(a[X])}$ to $add(a_j[X_j])$, for $1 < j < k$; $\boldsymbol{do}_k^{id(a[X])}$ to $pre(a_k[X_k])$ and $del(a_k[X_k])$. If $\sigma(a[X])$ has only one sub-schema, **do** is not used.*

(iii) ***Token par.*** *Assuming a bijective function $id : X \mapsto \{1, \dots, |X|\}$, if $x \in X_i \cap X_j$ for $i \neq j$, then define $\tilde{\sigma}(a[X])$ adding the literal $\boldsymbol{par}^{id(x)}(x)$ to $add(a_{jmin}[X_{jmin}])$ where $jmin$ is the smallest j such that $x \in X_j$; to $pre(a_j[X_j])$ for every $j > jmin$ such that $x \in X_j$; to $del(a_{jmax}[X_{jmax}])$ where $jmax$ is the largest j such that $x \in X_j$.*

By $\tilde{\sigma}(A[Z]) = \bigcup_{a[X] \in A[Z]} \tilde{\sigma}(a[X])$ we denote the set of sub-schemas obtained by applying $\tilde{\sigma}$ to all action schemas in $A[Z]$. We refer to $\tilde{\sigma}(A[Z])$ as the split domain *obtained from $A[Z]$ via σ.*

Tokens **procnone** and **do** together ensure that the sub-schemas of an original schema $a[X]$ can only be executed en block, i. e., grouped together. Thanks to **procnone**, no other block can be active when we start with $a_1[X_1]$, and we only release the block when we end with $a_k[X_k]$. As the **do** token is ID'ed, no sub-actions from any other action schema can be executed in between. Token **do** furthermore enforces the chosen sequentialization within the block, ensuring soundness, ensuring that every sub-schema is applied exactly once, and ensuring that the temporality underlying clauses (i) and (iii) is adhered to. Token **par** forces the planner to instantiate the sub-schemas consistently, by fixing the instantiation of every shared parameter x in the first sub-schema using x. The token is ID'ed with the variable in question, as otherwise the roles of two shared variables could be exchanged (if x and y are instantiated to o_1 respectively o_2 up front, then the roles of non-ID'ed instantiated tokens **par**(o_1) and **par**(o_2) could be changed later on, e. g. using o_2 for x and o_1 for y).

Example 6. *Consider once more the action schema **Move**(x,y,z). Using the valid split in Example 4 (bottom figure), and using $id(\boldsymbol{Move}(x,y,z)) = 1$ as well as $id(x) = 1$ and $id(y) = 2$, we obtain the following decorated split $\tilde{\sigma}(\boldsymbol{Move}(x,y,z))$:*

$\boldsymbol{Move}_1(x,y)$
 pre : $\{on(x,y), clear(x), \boldsymbol{procnone}\}$
 add : $\{\boldsymbol{do}_2^1, \boldsymbol{par}^1(x), \boldsymbol{par}^2(y)\}$
 del : $\{on(x,y), \boldsymbol{procnone}\}$
$\boldsymbol{Move}_2(x,z)$
 pre : $\{clear(z), \boldsymbol{do}_2^1, \boldsymbol{par}^1(x)\}$
 add : $\{on(x,z), \boldsymbol{do}_3^1\}$
 del : $\{clear(z), \boldsymbol{do}_2^1, \boldsymbol{par}^1(x)\}$
$\boldsymbol{Move}_3(y)$
 pre : $\{\boldsymbol{do}_3^1, \boldsymbol{par}^2(y)\}$
 add : $\{clear(y), \boldsymbol{procnone}\}$
 del : $\{\boldsymbol{do}_3^1, \boldsymbol{par}^2(y)\}$

*These sub-schemas can only be executed in the given order. They consume the block-token **procnone** at the start and release it at the end. Parameters x and y have to be instantiated consistently as **Move**$_2(x,z)$ has to get **par**$^1(x)$ from **Move**$_1(x,y)$, and **Move**$_3(y)$ has to get **par**$^2(y)$ from **Move**$_1(x,y)$. The ordering constraints from the split's quotient graph are respected. Thus all three issues (1–3) from Example 2 are solved.*

In general, the split domain preserves plans exactly; all we have to do is include the new artificial atoms, adding **procnone** into the initial state and goal:

Theorem 2. *Let $A[Z]$ be a set of action schemas, and let σ be a split function for $A[Z]$ such that, for every $a[X] \in A[Z]$, $\sigma(a[X])$ is valid. Let O be a finite set of objects, and let A respectively A^σ be the sets of ground actions obtained by instantiating every schema in $A[Z]$, respectively every schema in the split domain $\tilde{\sigma}(A[Z])$, with O. Denote $P = At(A)$ and let $I, G \subseteq P$ be any subsets of P. Then the plans for the task $\Pi = (P, A, I, G)$ are in one-to-one correspondence with those for the task $\Pi^\sigma = (At(A^\sigma), A^\sigma, I \cup \{\boldsymbol{procnone}\}, G \cup \{\boldsymbol{procnone}\})$.*

Proof. Any plan for Π can be transformed into a corresponding plan for Π^σ in the obvious manner, inflating every action grounding a schema $a[X]$ into the corresponding sequences of grounded sub-schemas $a_1[X_1], \dots, a_k[X_k]$. Vice versa, any plan for Π^σ can be transformed into a corresponding plan for Π in the inverse manner, thanks to the en block execution, consistent instantiation, and sound sequentialization as discussed above. □

Note that our transformation does *not* preserve optimality. An optimal plan for Π does not necessarily correspond to an optimal plan for Π^{σ}, nor vice versa, because action schemas with a larger number of sub-schemas get "punished". This could be avoided with the help of general action costs, simply by giving each sub-schema cost $1/|\sigma(a[X])|$. For now, we stick to uniform costs to keep things simple.

Domain Optimization

With the machinery to split action schemas at hand, we still need to design methods for applying that machinery automatically: *How to find good splits? And what are "good splits" anyhow?* Towards answering these questions, recall the hierarchy of splits between the atom split (all annotated atoms separated) and the trivial split (equal to the original schema, no splitting done). As we move up and down in that hierarchy for an action schema $a[X]$, coarser coverings have less sub-schemas and therefore tend to result in shorter plans using the split domain; and finer coverings have smaller interfaces and therefore tend to result in less ground actions. We capture this in terms of the split's *size*, $\mathit{SplitSize}(\sigma(a[X])) = |\sigma(a[X])|$, and *interface size*, $\mathit{IntSize}(\sigma(a[X])) = \max_{a_i[X_i] \in \sigma(a[X])} |X_i|$. Plan length increases linearly in $\mathit{SplitSize}(\sigma(a[X]))$ (if the underlying action indeed participates in the plan), and the number of ground actions decreases exponentially in $|X| - \mathit{IntSize}(\sigma(a[X]))$ (disregarding pruning methods such as static predicates as used in most implementations).[3] The trivial split is optimal in split size, the atom fit is optimal in interface size. In practice, we need to find a good trade-off between these two extremes. Unsurprisingly, doing so optimally is hard:

Theorem 3. *Let* split optimization *be the problem of deciding, given an action schema* $a[X]$ *as well as natural numbers* K *and* N*, whether there exists a valid split* $\sigma(a[X])$ *such that* $\mathit{SplitSize}(\sigma(a[X])) \leq K$ *and* $\mathit{IntSize}(\sigma(a[X])) \leq N$*. Then split optimization is* **NP***-complete.*

Proof. Membership is trivial: Guess a split $\sigma(a[X])$ and test whether it has the desired properties. Hardness can be proved via a polynomial reduction from Bin Packing. Each "item" of size n is simulated by an add atom with n parameters, with no overlaps between atoms. The interfaces between sub-schemas are then disjoint, simulating the "bins". Bin size corresponds to $\mathit{IntSize}(\sigma(a[X]))$, and the number of bins corresponds to $\mathit{SplitSize}(\sigma(a[X]))$; there are no ordering constraints so validity trivializes. □

Given this, for the time being we experimented with a family of greedy approximate optimization methods (exploring optimal splits is a topic for future work). Within these methods, we capture the trade-off in terms of a weighted sum, normalizing each criterion to the interval $[0, 1]$ to enhance comparability. From now on, for simplicity we assume an action schema $a[X]$ and, abusing notation, denote its split $\sigma(a[X])$ simply by σ. *Normalized split size* is

$$\mathit{NSplitSize}(\sigma) = \frac{\mathit{SplitSize}(\sigma)}{\mathit{SplitSize}(\mathit{AtomSplit}(a[X]))}$$

and *normalized interface size* is

$$\mathit{NIntSize}(\sigma) = \frac{\mathit{IntSize}(\sigma)}{\mathit{IntSize}(\mathit{TrivialSplit}(a[X]))}$$

Our optimization problem then is to find a valid split σ minimizing

$$\mathit{TradeOff}(\sigma) = \gamma \mathit{NSplitSize}(\sigma) + (1 - \gamma)\mathit{NIntSize}(\sigma)$$

where the parameter $\gamma \in [0, 1]$ controls the trade-off.

We approximate that optimization problem through either of *hill-climbing* or *beam search* in the split hierarchy as per Theorem 1, starting at the finest split and moving to coarser ones. In detail, we instantiate hill-climbing as follows:

- **Start node:** $\sigma_0 = \mathit{AtomSplit}(a[X])$.
- **Successor function:** $\mathit{SuccFn}(\sigma) = \{\sigma' \mid \sigma'$ is a merged split of σ as per Definition 8$\}$.
- **Evaluation function:** $f(\sigma) = \mathit{TradeOff}(\sigma)$.
- **Termination condition:** σ is a local minimum, i. e., $\forall \sigma' \in \mathit{SuccFn}(\sigma) \,.\, f(\sigma') > f(\sigma)$.

Hill-climbing thus iteratively generates all splits obtained by merging a mergeable pair of sub-schemas, selecting one with the best trade-off. Note that this is guaranteed to eventually end up in a local minimum, at the latest when we reach $\mathit{TrivialSplit}(a[X])$ which has no successors and thus is a local minimum by our definition.

Beam search is parameterized by *beam width* B. It is like breadth-first search except that, at each breadth-first level t, which we denote by Level_t, only B nodes with best f-value are kept. We instantiate beam search exactly like hill-climbing, except for the termination condition:

- **Beam search termination condition:** $\min_{\sigma' \in \mathit{Level}_{t+1}} f(\sigma') > \mathit{min}_{\sigma \in \mathit{Level}_t} f(\sigma)$, where t is the index of the level that is currently being expanded.

Intuitively, this termination condition can be understood as saying that "viewed as a whole, the current level is a local minimum". Note that, in this setup, hill-climbing is exactly beam search with $B = 1$.

In each of hill-climbing and beam search, ties are broken using the expression $\mathit{Overlap}(a_i[X_i], a_j[X_j]) = \frac{|X_i \cap X_j|}{|X_i \cup X_j|}$. That is, successor nodes σ' are ordered lexicographically by f-value first, and by $\mathit{Overlap}(a_i[X_i], a_j[X_j])$ second, where $a_i[X_i]$ and $a_j[X_j]$ are the sub-schemas merging which lead from the current node σ to σ'. Intuitively, $\mathit{Overlap}(a_i[X_i], a_j[X_j])$ gives a preference to merging sub-schemas that share a lot of variables, so that combining them appears favorable regarding interface size in future search steps. If ties remain within this enhanced ordering, these are broken arbitrarily.

[3]Ideally, one would be interested in the actual increase in plan length, respectively decrease in the number of ground actions, which at least for the latter parameter might even be feasible. We did not explore this for now, considering only the criteria (split size and interface size) that can be read directly off the split.

A few words are in order regarding the extreme cases $\gamma = 1$ (all weight on split size) and $\gamma = 0$ (all weight on interface size). With $\gamma = 1$, $f(\sigma') < f(\sigma)$ for all successor nodes σ' in every search step, so both searches will end up returning $\mathit{TrivialSplit}(a[X])$ (implying that it makes no sense to run them with $\gamma = 1$). With $\gamma = 0$, in contrast, the searches become very conservative, exploring only nodes with optimal interface size equalling that of $\mathit{AtomSplit}(a[X])$. These searches thus attempt to find smaller splits with optimal interface size. As all successors in each search step will have the same f-value (uniquely identified by their size), that search is guided only by $\mathit{Overlap}(a_i[X_i], a_j[X_j])$.

We use HC, respectively BS, to denote hill-climbing, respectively beam search. Although the former is a special case of the latter, we find this notation easier to look at.

Once HC or BS returned a valid split σ, we select a sound sequentialization of σ. Among sub-schemas not ordered with respect to each other, we prefer ones with more preconditions. This way, during a forward search, inapplicable instantiations of the split will be detected earlier on.

Evaluation

Our techniques are implemented as a stand-alone tool (not starting from existing PDDL parser or planner implementations) in Java. The source code is available at http://liis.famaf.unc.edu.ar/resources, together with the original and split domains used in the following evaluation.

A major question in the evaluation is *which domains to run*. We, of course, did run the IPC domains. However, it is a well known fact that these domains are engineered to challenge search, not pre-processes. This is particularly true of the aforementioned IPC Pipesworld and Cybersecurity domains (Hoffmann et al. 2006; Boddy et al. 2005) where action schemas were split manually to make the domains amenable to standard pre-processes. More generally, most benchmarks were created having in mind to test search capabilities; we are aware of only a single benchmark ("Pigeonhole" in (Ridder, Fox, and Long 2014)) that was created specifically to test pre-processing capabilities. A final detail is that domains with complex encodings (like Cybersecurity, DiningPhilosophers, and OpticalTelegraph) often come in ADL and are translated to STRIPS using a grounding compilation, to the effect that the versions we can handle (STRIPS) have no (lifted) schemas. As a result, almost all action schemas in IPC STRIPS domains have small interfaces, and there is not much to gain by schema splitting.

Our *focus domains* thus are (non-IPC) ones whose action schemas have large interfaces. We run the two versions of Pipesworld with the original *un-split* domain prior to the manual splitting operation, on the IPC test instances. The interesting question then is whether our automatic methods result in equal (or better) performance as the manually split domains used in the IPC. Similarly, we run Haslum's STRIPS genome edit distance problems (Haslum 2011) with the un-split original domain ("ged3-itt.pddl") as well as the manually split domain ("ged2-itt.pddl"), on the two test suites ("ds1" and "ds2nd") provided.[4]

We ran BS with $B \in \{2, 4, 8, \dots, 128\}$, and we ran both HC and BS with $\gamma \in \{0, 0.1, 0.2, \dots, 0.9\}$. Table 1 shows statistics about the splitting process. We do not report runtimes as they were always negligible, seeing as the splitting has to be done only once per domain (typically HC takes up to 1 second, and BS takes a few seconds, up to 1 minute). We report data only for HC as, mostly, BS did not find splits that were not also found by HC. With one exception, we report only $\gamma \in \{0, 0.7, 0.8\}$ as, mostly, other values of γ did not result in different splits.

	TrivialSplit			ManualSplit			HC 0.8			HC 0.7			HC 0.0			AtomSplit		
	A	avg	mx	A	avg	mx	A	avg	mx	A	avg	mx	A	avg	mx	A	avg	mx
Genome (1 and 2)	14	2.4	3	21	1.8	2	24	2.0	3	26	1.9	2	26	1.9	2	163	1.1	2
PipesworldNoT	4	8.0	9	6	6.3	7	8	5.0	7	10	4.6	5	24	2.7	3	59	2.0	3
PipesworldT	4	10.5	12	6	6.3	7	14	5.1	6	22	3.6	6	32	3.0	3	93	1.8	3
Freecell	10	4.9	7				19	2.7	7	24	2.2	5	35	1.9	2	117	1.3	2
Transport	3	4.3	5				TrivialSplit			TrivialSplit			15	2.0	2	20	2.0	2

Table 1: Split Statistics. "Trivial Split": un-split domain for our focus benchmarks, original IPC domain for the two IPC examples. "A": number of action schemas; "avg": average interface size; "mx": maximal interface size.

Genome1 and Genome2 use the same domain (they differ in the instance set only) and they are shown in a single row in Table 1. We include Freecell and Transport to exemplify the behavior in standard IPC benchmarks. Clearly, as HC gets more tailored towards small interface size for small values of γ, interface size goes down while the number of action schemas goes up. This is true of our focus domains just like most IPC benchmarks, to varying extents. AtomSplit forms an extreme case with extremely low average interface size, but at the cost of extremely many action schemas and (as dictated by theory) no gain in maximal interface size over HC with $\gamma = 0$. Compared to the manually split domains, our automatic splits always get down to the same maximal interface size or much less, but at the cost of a larger number of action schemas.

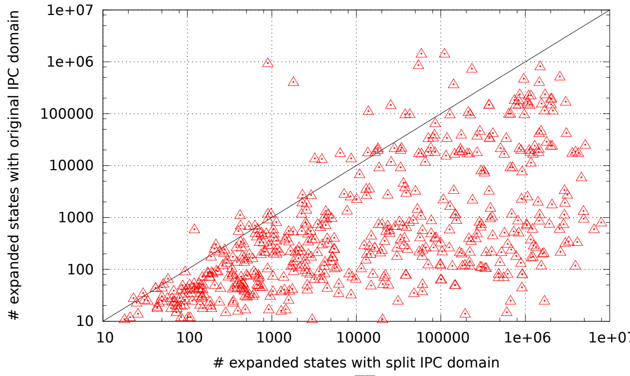

Figure 1: Expanded states with h^{FF} in IPC domains (all HC versions, no AtomSplit).

Regarding performance, as a canonical planner to consider, we ran Fast Downward (FD) (Helmert 2006) using

[4]The original cyber security domain, as well as natural language generation (Koller and Hoffmann 2010), challenge pre-processes as well, but are formulated in ADL so our current tool cannot handle them. Extending our techniques to ADL is an ongoing topic. Pigeonhole from (Ridder, Fox, and Long 2014) has a single unsplittable action schema with interface size 2, and the challenge constructed arises only from an enormously large set of objects.

	TrivialSplit	ManualSplit			HC 0.8			HC 0.7 (Transport: 0.6)			HC 0.0			AtomSplit		
	Cov	Cov	Time	Grd A	Cov	Time	Grd A	Cov	Time	Grd A	Cov	Time	Grd A	Cov	Time	Grd A
	h^{FF} (no preferred operators)															
Genome1	15	*52*	*8380*	*22.7m*	*21*	*5361*	*18.8m*	**66**	**8508**	**21.9m**	**66**	**8508**	**21.9m**	6	-438	18.6m
Genome2	154	*156*	*14566*	*4.3m*	*156*	*15358*	*3.5m*	**156**	**17717**	**4.1m**	**156**	**17717**	**4.1m**	24	-1662	2.7m
PipesworldNoT	24	**28**	**1230**	**6.7m**	**28**	**1220**	**6.7m**	17	625	6.7m	10	-464	6.7m	13	-1096	6.7m
PipesworldT	12	**22**	**365**	**2.9m**	8	-925	2.6m	11	-2585	2.7m	*13*	*-1855*	*2.9m*	11	-492	2.9m
Freecell	**59**				19	-167	0.7m	24	-394	0.8m	30	-2210	0.8m	39	-3467	-13167
Transport	30					TrivialSplit		**30**	**714**	**85410**	25	-151	86188	26	-396	79463
	LAMA (first iteration)															
Genome1	73	**110**	**64176**	**22.7m**	34	-4716	18.8m	*76*	*-1188*	*21.9m*	*76*	*-1188*	*21.9m*	6	-22	18.6m
Genome2	156	**156**	**5392**	**4.3m**	156	-7277	3.5m	156	-4410	4.1m	156	-4410	4.1m	69	-16985	2.7m
PipesworldNoT	38	**44**	**8618**	**6.7m**	**44**	**8625**	**6.7m**	23	794	6.7m	23	-183	6.7m	17	-1014	6.7m
PipesworldT	16	**40**	**3226**	**2.9m**	13	-2900	2.6m	13	-600	2.7m	*19*	*727*	*2.9m*	15	215	2.9m
Freecell	**59**				36	-3728	0.7m	38	-2432	0.8m	35	-2392	0.8m	31	-7167	-13167
Transport	**30**					TrivialSplit		30	-471	85410	30	-1761	86188	28	-1100	79463

Table 2: Performance overview. "Cov": coverage; "Time": sum of the total-runtime advantage over TrivialSplit, across those instances commonly solved using both domain versions involved; "Grd A": sum of the number-of-ground-actions advantage over TrivialSplit ("m": million), across those instances where that set was successfully computed using both domain versions involved (which are all instances except for PipesworldNoTankage where TrivialSplit completed only 38 cases and Pipesworld-Tankage where TrivialSplit completed only 20 cases). Best-performing domain version(s) shown in boldface, split domains better than the original domain shown in italic ("better" here means higher coverage, or equal coverage and better runtime).

h^{FF} in lazy greedy best-first search without preferred operators. As a representation of the state of the art in runtime, we ran (the FD implementation of) the first search iteration of LAMA (Richter and Westphal 2010).

There are 27 IPC STRIPS benchmarks that our parser can handle. On these, (a) maximal interface size and the number of ground actions tends to go down, but (b) overall performance suffers because grounding is not the bottleneck and the split domains tend to result in larger search spaces. To illustrate (a), while the summed-up number of ground actions is about 4 million with the original domains, it is about 2.5 million with the domains split by HC with $\gamma = 0$. To illustrate (b), see Figure 1.

Table 2 summarizes performance data for our focus domains (plus the two exemplary IPC benchmarks from Table 1). A quick glance at the table immediately conveys two major messages: *schema splitting dramatically reduces the number of ground actions in domains with large interfaces, often yielding substantial performance improvements; our automatically split domains are often better than the original ones, and are sometimes as good as, or even strictly better than, the manually split ones.* The second observation is especially true for the canonical h^{FF} planner, where we substantially beat the manually split domain in both genome edit distance test suites, are equally good in PipesworldNoTankage, and produce some automatic split better than the original domain in all focus domains (Transport is one of the very few cases where performance gets better in the IPC benchmarks). The picture is not as positive for LAMA, but still there are good results in Genome1 and PipesworldNoTankage. We remark that, in PipesworldNoTankage, the number of ground actions for HC with $\gamma = 0.8$ is *exactly* the same as that with the manually split domain. In that sense, our automatic splitting methods re-construct the manual split here (although our domain has more action schemas, cf. Table 1).

Figure 2 shows expanded states for our focus domains. The split domains still tend to result in larger search spaces,

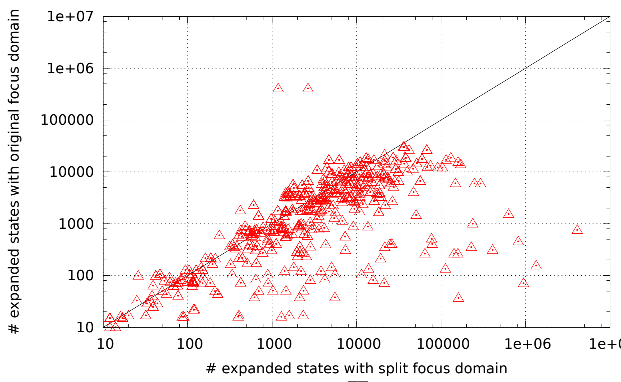

Figure 2: Expanded states with h^{FF} in our focus domains (all HC versions, no AtomSplit).

though to a lesser extent than on the IPC domains (compare Figure 1). Hence the performance improvements in Table 2 are mainly due to the savings in pre-processing time.

Conclusion

We have systematized and automated prior works on action schema splitting, as a pre-process to standard planners that ground out the actions. The method shows promise on domains with large interfaces that were previously split by hand, indicating that it could be a useful tool for, especially, applicationers without planning expertise who wish to apply planning technology but are not intimately familiar with it.

The most pressing line of work, already ongoing, is to extend our techniques to ADL so that we can handle more complex domain descriptions, in particular those of the original cyber security domain as well as natural language generation. An interesting open question regards better domain optimization methods, that measure more directly the impact on the planner, rather than syntactical properties of the split domain. A promising radical variant could be to implement the splitting process as a kind of domain-specific learning, where one would fix a set of small training instances and optimize relative to the actuall performance of a planner solving these instances with different domain versions.

Acknowledgments. This work is partially supported by the EU FP7 Programme under grant agreement no. 295261 (MEALS).

References

Boddy, M.; Gohde, J.; Haigh, T.; and Harp, S. 2005. Course of action generation for cyber security using classical planning. In Biundo, S.; Myers, K.; and Rajan, K., eds., *Proceedings of the 15th International Conference on Automated Planning and Scheduling (ICAPS-05)*, 12–21. Monterey, CA, USA: Morgan Kaufmann.

Botea, A.; Enzenberger, M.; Müller, M.; and Schaeffer, J. 2005. Macro-FF: Improving AI planning with automatically learned macro-operators. *Journal of Artificial Intelligence Research* 24:581–621.

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

Gazen, B. C., and Knoblock, C. 1997. Combining the expressiveness of UCPOP with the efficiency of Graphplan. In Steel, S., and Alami, R., eds., *Recent Advances in AI Planning. 4th European Conference on Planning (ECP'97)*, volume 1348 of *Lecture Notes in Artificial Intelligence*, 221–233. Toulouse, France: Springer-Verlag.

Haslum, P., and Jonsson, P. 2000. Planning with reduced operator sets. In Chien, S.; Kambhampati, R.; and Knoblock, C., eds., *Proceedings of the 5th International Conference on Artificial Intelligence Planning Systems (AIPS-00)*, 150–158. Breckenridge, CO: AAAI Press, Menlo Park.

Haslum, P. 2007. Reducing accidental complexity in planning problems. In Veloso, M., ed., *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI-07)*, 1898–1903. Hyderabad, India: Morgan Kaufmann.

Haslum, P. 2011. Computing genome edit distances using domain-independent planning. In *Proceedings of the 5th International Scheduling and Planning Applications woRKshop (SPARK)*.

Helmert, M. 2006. The Fast Downward planning system. *Journal of Artificial Intelligence Research* 26:191–246.

Hoffmann, J.; Edelkamp, S.; Thíebaux, S.; Englert, R.; Liporace, F.; and Trüg, S. 2006. Engineering benchmarks for planning: the domains used in the deterministic part of IPC-4. *Journal of Artificial Intelligence Research* 26:453–541.

Knoblock, C. 1994. Automatically generating abstractions for planning. *Artificial Intelligence* 68(2):243–302.

Koller, A., and Hoffmann, J. 2010. Waking up a sleeping rabbit: On natural-language sentence generation with ff. In Brafman, R. I.; Geffner, H.; Hoffmann, J.; and Kautz, H. A., eds., *Proceedings of the 20th International Conference on Automated Planning and Scheduling (ICAPS'10)*. AAAI Press.

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

Newton, M. A. H.; Levine, J.; Fox, M.; and Long, D. 2007. Learning macro-actions for arbitrary planners and domains. In Boddy, M.; Fox, M.; and Thiebaux, S., eds., *Proceedings of the 17th International Conference on Automated Planning and Scheduling (ICAPS'07)*, 256–263. Providence, Rhode Island, USA: Morgan Kaufmann.

Palacios, H., and Geffner, H. 2009. Compiling uncertainty away in conformant planning problems with bounded width. *Journal of Artificial Intelligence Research* 35:623–675.

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

Ridder, B.; Fox, M.; and Long, D. 2014. Heuristic evaluation based on lifted relaxed planning graphs. In Chien, S.; Do, M.; Fern, A.; and Ruml, W., eds., *Proceedings of the 24th International Conference on Automated Planning and Scheduling (ICAPS'14)*. AAAI Press.