Fluent Merging for Classical Planning Problems

Jendrik Seipp and **Malte Helmert**
Albert-Ludwigs-Universität Freiburg
Institut für Informatik
Georges-Köhler-Allee 52
79110 Freiburg, Germany
{seipp, helmert}@informatik.uni-freiburg.de

Abstract

Fluent merging is a reformulation technique for classical planning problems that can be applied automatically or semi-automatically. The reformulation strives to transform a planning task into a representation that allows a planning algorithm to find solutions more efficiently or to find solutions of better quality. This work introduces different approaches for fluent merging and evaluates them within a state-of-the-art planning system.

Introduction

In classical planning we try to find plans in a fully observable world. While searching for a plan we move from one state to another. Each state is a function that assigns values to a number of variables. The set of values a variable can have is called its domain.

In the original definition of planning problems in PDDL notation (McDermott et al. 1998) there are only *Boolean variables*. Recent research has shown that combining several such variables into more general *finite-domain variables*, a process that has been called *fluent merging* (van den Briel, Kambhampati, and Vossen 2007), can make plan search more efficient. Helmert (2009) discusses a substantial number of planning approaches that benefit from such a conversion. Success stories include a SAT-based planner (Chen, Zhao, and Zhang 2007), a planner based on integer programming (van den Briel, Vossen, and Kambhampati 2005), symbolic planning with BDDs (Edelkamp and Helmert 2001) and heuristic search planners using pattern databases (Edelkamp 2001; Haslum et al. 2007), merge-and-shrink abstractions (Helmert, Haslum, and Hoffmann 2007), the causal graph heuristic (Helmert 2004) or the context-enhanced additive heuristic (Helmert and Geffner 2008). In all these cases, finite-domain fluents are derived by combining groups of Boolean variables that cannot be true simultaneously (i. e., which are *mutex*).

In this paper, we examine additional ways of merging fluents in order to facilitate planning, using a representation where mutex propositions have already been combined as a starting point. Our work is inspired by an article by van den Briel, Kambhampati, and Vossen (2007), in which the authors describe the possible benefits of general fluent merging for planning. They claim that only combining mutex propositions is too conservative and propose the combination of finite-domain variables that have "strong dependencies". They mention two possible methods for discovering such dependencies. The first method merges variables with the property that *all* operators that change one of the variables also mention the other variable in a precondition or effect. The second method merges variables with the property that at least one operator has a precondition but no effect on the first variable and an effect on the second variable. (However, this is a very general criterion and is often satisfied by thousands of variable pairs in a planning task, not all of which can be merged within practical resource limits.)

Van den Briel et al. present examples that indicate that their ideas could lead to improved planner performance, but do not report experimental results or provide a precise algorithm. They suggest that further research on the topic is necessary.

Formal Semantics

We formalise finite-domain planning tasks using the SAS$^+$ formalism (Bäckström and Nebel 1995), largely following the notation of Helmert, Haslum, and Hoffmann (2007). (We briefly remark that our implementation has been extended to cover finite-domain representations allowing conditional effects, but we limit ourselves to the easier SAS$^+$ case here for simplicity of presentation.)

Definition 1 (SAS$^+$ planning task)
An **SAS$^+$** ***planning task*** *or* **SAS$^+$** ***task*** *for short is a 4-tuple* $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$ *with the following components:*

- $\mathcal{V} = \{v_1, \ldots, v_n\}$ *is a set of* ***state variables*** *or* ***fluents****, each with an associated finite domain* $\mathcal{D}_v$*. If* $d \in \mathcal{D}_v$ *we call the pair* $v = d$ *an* ***atom****.*
  A ***partial variable assignment*** *over* $\mathcal{V}$ *is a function* s *on some subset of* $\mathcal{V}$ *such that* $s(v) \in \mathcal{D}_v$ *wherever* $s(v)$ *is defined. If* $s(v)$ *is defined for all* $v \in \mathcal{V}$*,* s *is called a* ***state****.*
- $\mathcal{O}$ *is a set of* ***operators****, where an operator is a triple* $\langle name, pre, eff \rangle$ *where name, the* ***name*** *of the operator is a unique symbol that distinguishes this operator from others, and pre and eff are partial variable assignments called* ***preconditions*** *and* ***effects****, respectively.*
- s_0 *is a state called the* ***initial state****, and* $s_\star$ *is a partial variable assignment called the* ***goal****.*

We assume that the reader is familiar with the semantics of planning tasks (operator application, plans, etc.) and refer to the literature for details (e. g., Helmert, Haslum, and Hoffmann 2007). To clarify notation, we only recall one important definition, that of *transition systems*.

Definition 2 (transition systems)
*A **transition system** is a 5-tuple $\mathcal{T} = \langle S, L, T, s_0, S_\star \rangle$ such that*

- *S is a finite set called the set of **states** of $\mathcal{T}$,*
- *L is a finite set called the set of **transition labels** or **labels** of $\mathcal{T}$,*
- *$T \subseteq S \times L \times S$ is the set of **transitions** of $\mathcal{T}$,*
- *$s_0 \in S$ is the **initial state** of $\mathcal{T}$, and*
- *$S_\star \subseteq S$ is the set of **goal states** of $\mathcal{T}$.*

We write $(s \xrightarrow{l} s') \in T$ or simply $s \xrightarrow{l} s'$ when T is clear from context to denote that $\mathcal{T}$ has a transition from s to s' with label l, i. e., to denote $\langle s, l, s' \rangle \in T$.

Planning semantics are defined in terms of transition systems. Put briefly, each planning task Π defines a transition system whose states are the states of Π (i. e., the complete assignments to the fluents of Π), whose initial and goal states match the initial and goal states of Π, and whose transitions are defined by the semantics of operator application for the operators of Π, with transition labels corresponding to operator names.

In addition to the transition system of the complete planning task Π, for the purposes of fluent merging we are also interested in more localised views that only capture the semantics of planning tasks with respect to *a particular fluent*. This can be achieved by considering transition systems *induced by atomic abstractions*. Again, we only provide the definition for the limited case that is important for this work and refer to the literature for more general definitions of abstractions and induced transition systems (Helmert, Haslum, and Hoffmann 2007).

Definition 3 (atomic abstractions)
Let $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$ be an SAS$^+$ *task, and let $v \in \mathcal{V}$ be one of its state variables. The **transition system induced by the atomic abstraction to** v, or more succinctly the **transition system for** v, is the transition system $\mathcal{T}_v = \langle S^v, L^v, T^v, s_0^v, S_\star^v \rangle$ such that:*

- *$S_v = \mathcal{D}_v$ (i. e., the domain of v forms the states of $\mathcal{T}_v$),*
- *L^v is the set of operator names in $\mathcal{O}$,*
- *there is a transition $d \xrightarrow{l} d'$ whenever the transition system defined by Π has a transition $s \xrightarrow{l} s'$ with states s and s' such that $s(v) = d$ and $s'(v) = d'$,*
- *$s_0^v = s_0(v)$ (i. e., the initial state in $\mathcal{T}_v$ is the value of v in the initial state of Π), and*
- *$S_\star^v$ consists of all possible values $d \in \mathcal{D}_v$ that can occur in goal states of Π.*

Even though they are defined *semantically*, based on the exponentially large transition systems of planning tasks, transition systems induced by atomic abstractions in SAS$^+$ tasks can be computed *syntactically*, i. e., using efficient operations directly on the compact task representation Π. In particular, an operator $\langle \mathit{name}, \mathit{pre}, \mathit{eff} \rangle$ induces a transition transition $d \xrightarrow{\mathit{name}} d'$ in the transition system for v iff

- d is compatible with $\mathit{pre}(v)$, i. e., $\mathit{pre}(v)$ is undefined or $\mathit{pre}(v) = d$, and
- $\langle d, d' \rangle$ is compatible with $\mathit{eff}(v)$, i. e., $\mathit{eff}(v)$ is undefined and $d = d'$, or $\mathit{eff}(v) = d'$.

Moreover, it is not necessary to iterate over all (possibly exponentially many) goal states in order to determine $\mathcal{S}_\star^v$: rather, the set of abstract goal states is simply the complete domain $\mathcal{D}_v$ if $s_\star(v)$ is undefined, and $\{s_\star(v)\}$ otherwise.

A transition system for a variable v can be viewed as a labeled directed graph, and it shares many similarities with *domain transition graphs (DTGs)*, introduced by Jonsson and Bäckström (1998). Van den Briel et al. use DTGs as the basis for defining fluent merging. However, there are some semantic differences between the two kinds of graphs, which make definitions of fluent merging based on DTGs slightly more complicated. In particular, atomic abstractions for v represent the behaviour of *all* operators with respect to v, while DTGs only consider operators that change the value of the represented variable.

Given only the DTGs of a planning task, it is not possible to reconstruct which operators have preconditions on variables that they do not modify. Given only the transition systems for individual variables, however, it is possible to reconstruct the complete transition system of an SAS$^+$ task (Helmert, Haslum, and Hoffmann 2007, Theorem 8 and following discussion). This is done through the operation of computing *synchronised products* (Helmert, Haslum, and Hoffmann 2007), which also provide the formal underpinnings of fluent merging.

Definition 4 (synchronised product)
Let $\mathcal{T}^1 = \langle S^1, L, T^1, s_0^1, S_\star^1 \rangle$ and $\mathcal{T}^2 = \langle S^2, L, T^2, s_0^2, S_\star^2 \rangle$ be transition systems with the same labels.

*The **synchronised product** of $\mathcal{T}^1$ and $\mathcal{T}^2$ is defined as $\mathcal{T}^1 \otimes \mathcal{T}^2 = \langle S, L, T, s_0, S_\star \rangle$, where*

- *$S = S^1 \times S^2$,*
- *$(\langle s^1, s^2 \rangle \xrightarrow{l} \langle t^1, t^2 \rangle) \in T$ iff $(s^1 \xrightarrow{l} t^1) \in T^1$ and $(s^2 \xrightarrow{l} t^2) \in T^2$,*
- *$s_0 = \langle s_0^1, s_0^2 \rangle$, and*
- *$S_\star = S_\star^1 \times S_\star^2$.*

Synchronised products play an important role in the computation of so-called *merge-and-shrink abstractions*; they correspond to the *merge* steps in the abstraction computation (Helmert, Haslum, and Hoffmann 2007). They also provide a clean and direct semantics for fluent merging in SAS$^+$ tasks. As discussed above, SAS$^+$ tasks can be equivalently represented through the set of all atomic transition systems, $\{\mathcal{T}_v \mid v \in \mathcal{V}\}$. *Fluent merging* then means choosing two fluents $u, v \in \mathcal{V}$, removing their transition systems $\mathcal{T}_u$ and $\mathcal{T}_v$ from the set, and replacing them with their synchronised product, $\mathcal{T}_u \otimes \mathcal{T}_v$. If we interpret this at the task level, this can be seen as replacing fluents u and v with a *product fluent* $u \otimes v$, a view which we will take on in the following.

There is one small issue that makes this symmetry between state variables and transition systems imperfect: in cases where a goal value is defined for one of u and v but not the other, there is no clean way of defining a goal value for the product fluent $u \otimes v$. However, this can easily be addressed by standard compilation techniques to compile away disjunctive goals (Gazen and Knoblock 1997).

Fluent Merging in Fast Downward

Based on the theoretical definition of Fluent Merging we briefly discuss how we implemented the technique by integrating it into the Fast Downward planning framework (Helmert 2006). First we briefly explain the framework itself. In order to find a plan, Fast Downward proceeds in three main stages (Helmert 2006):

- In the *translation* stage, a PDDL (McDermott et al. 1998) problem is parsed, normalised, grounded and translated into an SAS^+ planning task.
- In the *knowledge compilation* stage a relevance analysis is performed and some data structures are prepared for the last stage.
- In the last stage the actual *search* is executed. For this purpose many different heuristics and search methods are available.

We integrate fluent merging between the first and second stage. The fluent merging algorithm reads the SAS^+ planning task that was written by the translator and outputs a new SAS^+ planning task with merged variables. The new component can be integrated easily since none of the original components have to be altered.

Fluent Merging Algorithm

The fluent merging algorithm is composed of two steps. In the first step, we select groups of variables that are then merged in the second step. We tested many different *selection* methods and will discuss them later. This section explains the second step, the generic *merging* procedure.

The merging procedure follows the definition of synchronised products that provides the formal semantics of fluent merging, but unlike that definition works directly on the task description level.

Let $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$ be an SAS^+ planning task and $a, b \in \mathcal{V}$ the variables that we want to replace with a merged variable $a \otimes b$.

Merged Variable The new variable is assigned the domain $\mathcal{D}_a \times \mathcal{D}_b$.

Initial State We set $s_0(a \otimes b) = \langle s_0(a), s_0(b) \rangle$.

Operators Every operator that mentions a or b in its precondition or effect needs to be updated. In some cases, a single operator needs to be replaced by multiple new operators. All new operators are assigned the same name as the original operator, so that plans for the modified problem (described as sequences of operators represented by their names) can be used verbatim as plans for the original problem. The algorithm for adapting an operator is shown in Figure 1. The special cases in lines 2–6 are not strictly necessary, but speed up the computation and lead to a more compact result in common cases.

procedure adapt-operator(ops $\mathcal{O}$, op o, var a, var b)

1. $\langle \mathit{name}, \mathit{pre}, \mathit{eff} \rangle := o$
2. **if** o mentions neither a nor b **then**
3. **return**
4. **if** $\mathit{eff}(a)$ and $\mathit{eff}(b)$ are defined **and** $\mathit{pre}(a)$ and $\mathit{pre}(b)$ are undefined **then**
5. $\mathit{eff}(a \otimes b) := \langle \mathit{eff}(a), \mathit{eff}(b) \rangle$
6. **return**
7. $\mathrm{poss}_a := \{\mathit{pre}(a)\}$ if defined, else $\mathcal{D}_a$
8. $\mathrm{poss}_b := \{\mathit{pre}(b)\}$ if defined, else $\mathcal{D}_b$
9. **foreach** $a_{\mathit{pre}} \in \mathrm{poss}_a$ **do**
10. **foreach** $b_{\mathit{pre}} \in \mathrm{poss}_b$ **do**
11. $a_{\mathit{eff}} := \mathit{eff}(a)$ if defined, else a_{pre}
12. $b_{\mathit{eff}} := \mathit{eff}(b)$ if defined, else b_{pre}
13. $\mathit{pre}_{\mathit{new}} := \mathit{pre}$
14. $\mathit{eff}_{\mathit{new}} := \mathit{eff}$
15. $\mathit{pre}_{\mathit{new}}(a \otimes b) := \langle a_{\mathit{pre}}, b_{\mathit{pre}} \rangle$
16. $\mathit{eff}_{\mathit{new}}(a \otimes b) := \langle a_{\mathit{eff}}, b_{\mathit{eff}} \rangle$
17. $o_{\mathit{new}} := \langle \mathit{pre}_{\mathit{new}}, \mathit{eff}_{\mathit{new}} \rangle$
18. $\mathcal{O} := \mathcal{O} \cup \{\langle \mathit{name}, \mathit{pre}_{\mathit{new}}, \mathit{eff}_{\mathit{new}} \rangle\}$
19. $\mathcal{O} := \mathcal{O} \setminus \{o\}$

Figure 1: Algorithm that adapts an operator $o \in \mathcal{O}$ during the merge of variables a and b.

Goal We have to distinguish three cases:

- $s_\star(a)$ and $s_\star(b)$ undefined:
  Nothing to do.
- $s_\star(a)$ and $s_\star(b)$ both defined:
  Set $s_\star(a \otimes b) = \langle s_\star(a), s_\star(b) \rangle$.
- Exactly one of $s_\star(a)$ and $s_\star(b)$ is defined:
  Without loss of generality, we assume that $s_\star(a)$ is defined. Then any of the values in the set $\{\langle s_\star(a), d \rangle \mid d \in \mathcal{D}_b\}$ should be treated as a possible goal value for $a \otimes b$. Since SAS^+ cannot represent goals of this form directly, we use a standard compilation technique for first compiling the actual goal into an operator (Gazen and Knoblock 1997) and then adapt this operator as described previously.

After these steps, all references to the old variables a and b can be removed from the task description.

While the algorithm as described only merges two variables at a time, it can be invoked repeatedly to merge "groups" or "clusters" of more than two variables. For example, to merge variables a, b and c into a single group, we would first merge a and b into $a \otimes b$ and then $a \otimes b$ and c

into $(a \otimes b) \otimes c$. Synchronised product operations are associative and commutative modulo isomorphism of transition systems, so the precise merge order does not matter.

Variable Selection

There are many possible criteria for finding variables to merge. In this section we present the methods that we have implemented in the course of this work. A *selection method* is an algorithm that has the following input:

- An SAS$^+$ planning task $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$
- m, the number of variables that should be merged into a single "group" or "cluster"
- n, the maximum number of such groups to form

It returns a set of variable groups to be merged by the combination algorithm. Usually it will produce m groups of size n each, unless this is not possible because $m \cdot n > |\mathcal{V}|$. A selection method may also opt to produce smaller or fewer groups if it cannot find a sufficient number of suitably large promising candidate groups to merge, but this is not the case for the simpler methods described in this section.

The first set of experiments was conducted with the selection methods below. In these methods, each group is assigned a score that represents its suitability to be merged. This assignment is done by an evaluation function $e : \mathcal{V} \times \mathcal{V} \to \mathbb{R}$ that assigns a numerical value to all pairs of variables in the planning task. If $m \geq 3$, i. e., we seek to merge groups of three or more variables, variable groups G of size m are scored by computing the sum of all pairwise scores for pairs of variables in G. In the last step of the selection algorithm all groups are sorted in descending order of scores and we repeatedly pick the first group in the list until the maximum number of merges n has been reached. Groups to be merged are not allowed to overlap: once a variable has formed part of a merge, all groups that contain it are eliminated from further consideration.

We experimented with the following evaluation functions:

- Random variables (rand)
  Randomly select variable groups.
  $$e(a, b) = \text{random}()$$
- Mutex variables (mutex)
  Prefer groups with variables whose domains are maximally mutex, i. e., contain as many value pairs as possible that cannot be simultaneously true according to the mutex information generated by Fast Downward's translation algorithm.
  $$e(a, b) = |\{(d_a, d_b) \in \mathcal{D}_a \times \mathcal{D}_b \mid d_a \text{ and } d_b \text{ mutex}\}|$$
- Number of atoms (size)
  Choose variables whose merging minimises the total number of atoms of the planning task.
  $$e(a, b) = -(|\mathcal{D}_{a \otimes b}| - (|\mathcal{D}_a| + |\mathcal{D}_b|))$$
- Connected variables (conn)
  Prefer variable groups that are heavily connected in the *causal graph*.
  $$\text{conn}(a, b) = \text{cg\_weight}\ (a, b) + \text{cg\_weight}\ (b, a)$$
  $$e(a, b) = \text{conn}(a, b)$$
  Here, the cg_weight of a causal graph edge is the number of operators that induce it (Helmert 2006).
- Two-cycle pairs (cycles)
  Prefer variables that form a two-cycle in the causal graph.
  $$e(a, b) = \begin{cases} \text{conn}(a, b) & \text{if } \langle a, b \rangle \in E \\ & \text{and } \langle b, a \rangle \in E \\ \text{conn}(a, b) - 10^9 & \text{else} \end{cases}$$
  Here E is the set of directed edges of the causal graph.
- Goal variables (goals)
  Prefer variables that appear in the goal description.
  $$e(a, b) = \begin{cases} \text{conn}(a, b) & \text{if } s_\star(a) \text{ or } s_\star(b) \text{ defined} \\ \text{conn}(a, b) - 10^9 & \text{else} \end{cases}$$
- Variables minimizing number of operators (ops)
  Prefer variable groups whose merging minimise the number of new operators.
  $$e(a, b) = |\mathcal{O}| - |\mathcal{O}_{new}|$$
  Here O_{new} is the set of operators that would result from merging a and b.

Domain	no-merge	rand	mutex	size	conn	cycles	goals	ops
blocks (35)	35	35	35	35	31	31	31	35
driverlog (20)	20	17	13	16	19	14	18	15
grid (5)	5	1	1	5	0	1	1	2
gripper (20)	20	20	15	20	20	20	20	20
logistics00 (28)	28	28	28	28	28	28	28	28
logistics98 (35)	35	28	35	35	20	20	21	11
miconic (150)	150	150	150	150	150	150	150	150
mprime (35)	35	30	35	35	34	29	35	20
psr-small (50)	50	49	48	48	47	48	47	49
zenotravel (20)	20	20	16	16	20	20	19	15
depot (22)	17	11	14	12	**15**	**15**	13	14
freecell (80)	78	75	**77**	76	72	72	57	37
pathways (30)	15	14	**16**	**17**	14	14	13	15
pipes-nt (50)	38	5	8	**16**	14	14	9	**16**
pipes-t (50)	24	9	3	**17**	11	8	9	15
rovers (40)	34	31	34	**35**	34	34	34	24
schedule (150)	60	58	59	59	54	52	39	**60**
tpp (30)	28	20	**24**	**24**	22	**24**	23	16
trucks (30)	17	15	14	**16**	14	14	**16**	6
Total (880)	709	616	625	**660**	619	608	583	548

Table 1: Comparison of solved tasks for different fluent merging methods with the h^{cea} heuristic, $n = 5$ and $m = 2$. Number of tasks in each domain is shown in parentheses. In the domains above the separator line *no-merge* already solves all instances, so no improvement is possible. Below the separator the best results among the different selection methods are highlighted in bold and cases where the performance of the *no-merge* method is exceeded are underlined.

Table 1 shows the number of tasks solved by a greedy best-first search with deferred evaluation (Richter and Helmert 2009) and the h^{cea} heuristic in a number of IPC domains after applying fluent merging with the selection methods mentioned above. For the experiments we allowed the

search component of the planner to run for at most 30 minutes and use 2 GB of memory. The column *no-merge* reports results without performing fluent merging. The maximum group size m is set to 2 in these experiments, while the maximum number of groups n is set to 5. Other experiments ($n \in \{10, 20, \ldots, 100, 125, 150, 175, 200, 250, 300\}, m \in \{3, 4, 5, 6, 7\}$) all led to worse performance, i. e., fewer plans found in more time.

Although some of the entries in the table show improvements by performing fluent merging, we were not satisfied with the overall results. While looking for a smarter selection method we found the *same object method*, which performed significantly better in the Schedule and Pathways domains. (Unfortunately, other domains were not tested at the time.) Between the initial tests with the above mentioned methods and the experiments with the same object method, the implementation of the h^{cea} heuristic in Fast Downward was improved, making a direct comparison to the other methods' results difficult. We are currently rerunning all experiments with the improved h^{cea} version.

Same Object Method

The "same object method" exploits the fact that planning tasks are typically not given directly in a grounded representation like SAS$^+$, but instead use a first-order PDDL representation based on logical predicates and objects. As a first approximation, it is not entirely unreasonable to assume that SAS$^+$ fluents that stem from PDDL propositions that talk about the same object are more closely related than ones which do not. For example, even without looking at any operator definitions, a human problem solver might suspect that the two grounded propositions `(painted chair1)` and `(polished chair1)` are more closely related to each other than the two grounded propositions `(painted chair1)` and `(polished table3)` because they speak of the same object, `chair1`.

The same object method starts by associating exactly one object from the input PDDL representation with each SAS$^+$ fluent. Recall that each SAS$^+$ fluent v (before we apply our fluent merging algorithm) is formed from a group A_v of mutually exclusive PDDL atoms. The object associated with v is simply the object that occurs most frequently as a term in the atoms A_v. (Tie-breaking rules are applied when there is no unique such atom.) For example, variable v might be derived from mutex group $A_v = \{$`(at c2 loc1)`, `(at c2 loc2)`, `(at c2 loc3)`$\}$, which mentions the objects `c2`, `loc1`, `loc2` and `loc3`. Of these objects, `c2` is mentioned most frequently and hence becomes the object associated with v.

We only allow merges of fluents that are associated with the same object. However, since this can still lead to merged fluents with very large domains, we again limit the number of variables to be merged into a single group and the number of groups to merge, as in the previous algorithms. After some experimentation and following the comparatively good performance of the "size" method in the previous experiment, we decided to use the combined domain size of a group, i. e., the product of the domain sizes of the involved variables, as the quality measure for a merge, preferring groups whose combined domain size is as low as possible.

Experiments

In our initial experiments, the same object method significantly outperformed the other variable selection methods we tried, so all our subsequent experiments were based on this approach. After discouraging initial results with the merge-and-shrink heuristic (Helmert, Haslum, and Hoffmann 2007) and the landmark-cut heuristic (Helmert and Domshlak 2009), we concentrated our further experiments on satisficing configurations of Fast Downward. We again used greedy best-first search with deferred evaluation and tested three different heuristics:

- h^{cea}: the context-enhanced additive heuristic (Helmert and Geffner 2008)
- h^{FF}: the FF/additive heuristic (Hoffmann and Nebel 2001; Keyder and Geffner 2008)
- h^{CG}: the causal graph heuristic (Helmert 2004)

In all our experiments, forming groups of only two fluents ($m = 2$) produced better results than using larger clusters, so we only present results for this case. For the second fluent merging parameter, the number n of groups to form, the picture is more varied and differs significantly from domain to domain. Therefore, we report results for different values of this parameter, taken from the set $\{0$ (no merges), $2, 5, 10, 15, 20, 30\}$. For all experiments we used a 30 minute timeout for the search component of the planner.

The h^{cea} heuristic could not be significantly improved by fluent merging. Here only three more planning tasks could be solved by combining variables, and the overall coverage never improved.

The other two heuristics however showed some stronger potential benefits from fluent merging. As Table 2 shows, a total of 6 problem instances for h^{FF} and 12 instances for h^{CG} could be solved by some variation of fluent merging that eluded the same algorithm in the original problem representation. We should note that those two heuristics are already highly competitive planning methods. For h^{FF}, some parameter settings also achieve better *overall* performance in the tested domains. Table 3 provides detailed results for a particularly positive case, the challenging Sokoban domain, in which fluent merging increases the coverage of h^{FF} from 24 to 29 (out of 30) solved instances.

Conclusions and Future Work

We have provided the first general implementation and experimental evaluation of fluent merging for classical planning. Our results show that the approach holds promise: in some domains and with some combination methods, simply reformulating a problem instance by merging certain pairs of fluents improved problem solving performance.

However, our results also make it obvious that fluent merging does not improve heuristic accuracy across the board, and that further research is needed to find out which and how many fluents to merge, and whether fluent merging is useful for a given planning task at all.

	Merges h^{FF}							Merges h^{CG}						
Domain	0	2	5	10	15	20	30	0	2	5	10	15	20	30
airport (50)	25	25	25	25	25	25	25	**21**	**21**	**21**	**21**	**21**	20	20
assembly (30)	30	30	30	30	30	30	30	6	**7**	**7**	**7**	**7**	**7**	**7**
depot (22)	19	18	19	**20**	**20**	**20**	**20**	12	12	12	12	**13**	**13**	**13**
driverlog (20)	20	20	20	20	20	20	20	**20**	**20**	18	18	18	18	18
freecell (80)	76	**80**	78	77	79	78	75	72	70	70	**75**	74	74	72
miconic (150)	**150**	**150**	**150**	**150**	**150**	80	80	**150**	**150**	**150**	**150**	**150**	80	80
pprinter (30)	**23**	22	22	22	22	22	22	**24**	23	23	22	22	22	22
pipes-nt (50)	**43**	41	42	42	**43**	42	42	24	23	24	25	25	25	**26**
pipes-t (50)	38	**39**	38	37	**39**	37	37	17	**18**	17	15	16	15	15
psr-small (50)	50	50	50	50	50	50	50	50	50	50	50	50	50	50
rovers (40)	**40**	**40**	**40**	**40**	**40**	**40**	37	**32**	31	**32**	31	31	**32**	**32**
satellite (36)	34	34	34	34	34	34	34	34	34	34	34	34	34	34
schedule (150)	**150**	149	149	149	149	149	148	149	149	149	149	149	149	149
sokoban-sat (30)	24	28	**29**	28	28	28	28	**27**	26	24	25	25	25	25
storage (30)	**20**	**20**	**20**	**20**	19	19	19	20	20	20	20	20	20	20
tpp (30)	30	30	30	30	30	30	30	**27**	**27**	**27**	**27**	**27**	**27**	26
trucks (30)	**19**	17	17	18	18	18	18	10	11	11	11	11	**12**	11
wood-sat (30)	**29**	**29**	28	28	28	28	**29**	11	11	11	11	11	**14**	12
Total (908)	820	822	821	820	**824**	750	744	**706**	703	700	703	704	637	632

Table 2: Comparison of solved tasks for different maximum numbers of merges using the heuristics h^{FF} and h^{CG}. The maximum group size is set to 2 in this experiment. Number of tasks in each domain is shown in parentheses. Best results for each heuristic are highlighted in bold.

	h^{FF} **0 Merges**			h^{FF} **2 Merges**			h^{FF} **5 Merges**		
Inst.	Cost	Exp.	Time	Cost	Exp.	Time	Cost	Exp.	Time
Sokoban									
#16	345	29682	4.18	**329**	**21743**	**3.97**	371	24656	5.39
#17	**114**	**8543**	**1.27**	215	29104	5.31	247	38733	8.51
#18	497	2421586	314.03	**275**	1033770	172.36	301	**752976**	**152.67**
#19				93	**351433**	**182.57**	**52**	776873	476.85
#20									
#21	256	75853	11.23	**224**	**41600**	**8.62**	236	81226	19.9
#22				**389**	4024596	**767.93**	409	**3824691**	833.27
#23	343	**24924**	**2.98**	311	31248	5.46	**299**	27809	5.68
#24	**137**	173933	**26.48**	165	219517	40.26	**137**	**139241**	30.57
#25	221	**71862**	**12.14**	**185**	74991	14.67	245	89785	20.78
#26	402	**577660**	**110.98**	**320**	794300	200.1	434	1404490	420.94
#27				113	**529446**	**197.51**	**97**	923336	412.6
#28				536	**1983688**	**557.24**	**486**	3858328	1287.57
#29							**779**	**4673208**	**914.51**
#30	502	886401	134.08	494	719545	**116.84**	**470**	**677857**	127.8

Table 3: Detailed results for the sokoban-sat08-strips domain (15 smallest tasks omitted), using greedy best-first search with deferred evaluation and h^{FF}. The maximum group size was set to 2. For each number of merges we report the plan cost, number of expansions and search time in seconds. Best results are highlighted in bold.

Additionally, it can be expected that the parameters for the maximum number of merges and the maximum group size were not optimally set in our experiments. With current planning systems obtaining more and more knobs to tweak, the automatic parameter tuning methods like the ones found in the ParamILS framework appear worth investigating (Hutter et al. 2009).

Finally, we remark that in this work, we used the finite-domain representations generated by Fast Downward's translation component as a starting point. As van den Briel, Kambhampati, and Vossen (2007) observe, the reformulation performed by this translator is already a form of fluent merging (based on mutexes of the planning instance at hand), and it is far from clear whether the particular merging choices performed by the translation algorithm are ideal. Hence, another interesting question is whether we can derive a general fluent merging algorithm that starts from a regular Boolean encoding of a planning task and leads to a better representation than the one found by Fast Downward's default algorithm.

References

Bäckström, C., and Nebel, B. 1995. Complexity results for SAS^+ planning. *Computational Intelligence* 11(4):625–655.

Chen, Y.; Zhao, X.; and Zhang, W. 2007. Long-distance mutual exclusion for propositional planning. In Veloso, M. M., ed., *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI 2007)*, 1840–1845.

Edelkamp, S., and Helmert, M. 2001. The model checking integrated planning system (MIPS). *AI Magazine* 22(3):67–71.

Edelkamp, S. 2001. Planning with pattern databases. In Cesta, A., and Borrajo, D., eds., *Pre-proceedings of the Sixth European Conference on Planning (ECP 2001)*, 13–24.

Gazen, B. C., and Knoblock, C. A. 1997. Combining the expressivity 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 1997)*, volume 1348 of *Lecture Notes in Artificial Intelligence*, 221–233. Springer-Verlag.

Haslum, P.; Botea, A.; Helmert, M.; Bonet, B.; and Koenig, S. 2007. Domain-independent construction of pattern database heuristics for cost-optimal planning. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence (AAAI 2007)*, 1007–1012. AAAI Press.

Helmert, M., and Domshlak, C. 2009. Landmarks, critical paths and abstractions: What's the difference anyway? In Gerevini, A.; Howe, A.; Cesta, A.; and Refanidis, I., eds., *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS 2009)*, 162–169. AAAI Press.

Helmert, M., and Geffner, H. 2008. Unifying the causal graph and additive heuristics. In Rintanen, J.; Nebel, B.; Beck, J. C.; and Hansen, E., eds., *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS 2008)*, 140–147. AAAI Press.

Helmert, M.; Haslum, P.; and Hoffmann, J. 2007. Flexible abstraction heuristics for optimal sequential planning. In Boddy, M.; Fox, M.; and Thiébaux, S., eds., *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling (ICAPS 2007)*, 176–183. AAAI Press.

Helmert, M. 2004. A planning heuristic based on causal graph analysis. In Zilberstein, S.; Koehler, J.; and Koenig, S., eds., *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS 2004)*, 161–170. AAAI Press.

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

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

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

Hutter, F.; Hoos, H. H.; Leyton-Brown, K.; and Stützle, T. 2009. ParamILS: an automatic algorithm configuration framework. *Journal of Artificial Intelligence Research* 36:267–306.

Jonsson, P., and Bäckström, C. 1998. State-variable planning under structural restrictions: Algorithms and complexity. *Artificial Intelligence* 100(1–2):125–176.

Keyder, E., and Geffner, H. 2008. Heuristics for planning with action costs revisited. In *Proceedings of the 18th European Conference on Artificial Intelligence (ECAI 2008)*, 588–592.

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL – The Planning Domain Definition Language – Version 1.2. Technical Report CVC TR-98-003, Yale Center for Computational Vision and Control.

Richter, S., and Helmert, M. 2009. Preferred operators and deferred evaluation in satisficing planning. In Gerevini, A.; Howe, A.; Cesta, A.; and Refanidis, I., eds., *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS 2009)*, 273–280. AAAI Press.

van den Briel, M.; Kambhampati, S.; and Vossen, T. 2007. Fluent merging: A general technique to improve reachability heuristics and factored planning. In *ICAPS 2007 Workshop on Heuristics for Domain-Independent Planning: Progress, Ideas, Limitations, Challenges*.

van den Briel, M.; Vossen, T.; and Kambhampati, S. 2005. Reviving integer programming approaches for AI planning: A branch-and-cut framework. In Biundo, S.; Myers, K.; and Rajan, K., eds., *Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling (ICAPS 2005)*, 310–319. AAAI Press.