Twenty-Eighth International Conference on Automated Planning and Scheduling (ICAPS 2018)

Representing and Planning with Interacting Actions and Privacy

Shashank Shekhar, Ronen I. Brafman
Dept. of Computer Science
Ben-Gurion University, Israel
{shekhar,brafman}@cs.bgu.ac.il

Abstract

Interacting actions – actions whose joint effect differs from the union of their individual effects – are challenging both to represent and to plan with due to their combinatorial nature. So far, there have been few attempts to provide a succinct language for representing them that can also support efficient centralized and distributed privacy preserving planning. In this paper we suggest an approach for representing interacting actions succinctly and show how such a domain model can be compiled into a standard single-agent planning problem as well as to privacy preserving multi-agent planning. We test the performance of our method on a number of novel domains involving interacting actions and privacy.

Introduction

What happens when multiple agents perform actions concurrently? In principle, every combination of actions performed concurrently by a group of agents – a *joint action* – may define a different state-transition function. But as the number of joint-actions is exponential in the number of agents, specifying an explicit model for each combination of single-agent actions is impractical except for very simple cases.

To be succinct, a representation for joint actions must be compositional. That is, there must be some way of deducing the effect of the concurrent execution of actions $a_1, \ldots, a_n$ from the effects of smaller combinations. One option is to use a logical language and describe the effects of such combinations via formulas in this language. But for this, classical logic does not suffice. If we want each set of formulas to yield a unique model (that is, to specify a unique transition function), we must use some sort of non-monotonic logic, such as (Lin and Shoham 1992; Poole 1997; Baral and Gelfond 1997). While this might yield a satisfying representation scheme, it makes planning exceedingly difficult – indeed, most planners have difficulty even handling classical logic deductions.

Thus, the primary challenge for planning for multi-agent systems with interacting actions is finding a model for joint actions involving large sets of agents and large sets of single-agent actions, that is both succinct in ”natural” settings and supports efficient planning.

Due to the difficulty of representing and planning with interacting actions, most work on multi-agent planning algorithms ignores this issue, and considers concurrent non-interacting actions or sequential actions only. In the former case, only actions that impact different variables, or have the same effect on shared variables are considered. In that case, the effect of a joint-action is the union of effects of its single-agent components. In the latter case, joint-actions are not considered, and sequential plans containing actions by different agents are generated.

While much can be achieved without considering joint interacting actions, there are many settings where agents must coordinate their actions carefully to obtain desirable effects: a single-agent may be unable to lift or push heavy items, whereas this is possible for multiple agents acting together; if a table is not lifted from both sides concurrently, objects on it will fall; in robot soccer, more advanced teams perform coordinated maneuvers, such as one agent passing the ball to a free region while the intended receiver moves to this area at the same time; and in more complex manipulation tasks, coordinated activities of two or more arms are needed. In these examples, the effect of each move on its own is quite different from the effect of the coordinated actions.

The primary contributions of this paper are an intuitive formalism for specifying joint-actions in a compositional way and the definition and empirical evaluation of a compilation-based approach to planning by teams of agents with interacting actions, as well as privacy, for which we introduce a number of new domains. In addition, this paper highlights and discusses subtle issues that arise when attempting to model and plan with interacting actions.

To define the effect of joint-actions, we introduce *collaborative actions*. A collaborative action is a minimal combination of single-agent actions that cannot be defined as the union of its components. A joint action is defined as a *well-defined* set of single-agent and collaborative actions. A joint-action is well defined if its components (single-agent and collaborative actions) cannot be combined to yield more complex components. For example, consider box-pushing agents. A single-agent *push* action is effective when the box is light. A two-agent collaborative action *2push* – composed of two concurrent single-agent *push* actions – is effective when the box is heavy, as well. Given this, a joint action consisting of two single-agent *push* actions is not well-defined because these actions can be combined to form the collaborative *2push* action.

The difficulty of planning with joint actions depends on what interactions are allowed, and whether a distributed and privacy preserving algorithm is required. In the simplest case, only non-interacting concurrent actions are allowed in order to reduce make-span. A slightly more interesting case is when concurrent actions can destroy each other's preconditions. For example, suppose that a building becomes locked once an agent is detected entering it. Here, sequential execution allows a single-agent to enter the building, but parallel execution allows more agents to enter the building. In both cases, the effect of concurrent execution is the union of effects of the single-agent actions. Thus, there is no representational issue. But while sequential planning with post-processing works in the first case, a sequential planner cannot insert two *enter-building* actions. More complicated is the case where the effects of actions performed together differ from the union of their effects. Finally, on top of each of these cases, one can introduce the goal of preserving privacy. We will describe our compilation methods that support all cases, as well as the notion of object cardinality constraints introduced by (Crosby, Jonsson, and Rovatsos 2014).

Reproducibility. The source code and the domains we used are available on the corresponding author's home page.

Concurrent Actions – Related Work

Most work in classical planning allows for concurrent non-interacting actions in order to reduce the plan make-span and, in some cases, the depth of the search tree (see e.g., (Blum and Furst 1997)). Actions can occur concurrently if they do not interact with each other. However, there is no real notion of multiple agents, and the accepted semantics of this "concurrency" is that actions can be executed in any order with the same effect. A sufficient condition for this is that the union of effects and preconditions of concurrent actions is consistent. An alternative, called the $\exists$ semantics (Dimopoulos, Nebel, and Koehler 1997; Rintanen, Heljanko, and Niemelä 2006) was studied in the context of planning as satisfiability. It allows for interacting actions, provided they can be ordered in some sequentially legal way. But this semantics does not address true parallelism. Rather, it is a means of generating plans with fewer steps, for search efficiency reasons. It can be ambiguous, when multiple orderings are possible, and moreover, some natural examples described later do not have even a single legal sequential ordering.

Recently, (Crosby, Jonsson, and Rovatsos 2014) (henceforth, CJR) introduced an approach for centralized planning for multi-agent systems. In their formalism a joint-action is executed in each time step. The joint-action contains one single-agent action per agent, where that action can be a *no-op*. The preconditions and effects of a joint-action are the union of the preconditions (and resp. the effects) of its single-agent actions. In addition, they introduce a limited form of interaction among actions through object cardinality constraints. Every action is associated with a set of objects, and every set of objects may have constraints limiting the number of agents that can manipulate these objects concurrently. For example, a ship can sail if at least two agents sail it concurrently, or a bridge can be crossed by at most three agents concurrently. A joint-action is applicable in a state s if (1) its preconditions hold in s; (2) its effects are consistent; and (3) it satisfies the cardinality constraints on objects that appear in it. This formalism captures a limited form of interaction via cardinality constraints in a natural manner, but still assumes that the effect of a set of concurrent actions is the combined effect of the single-agent actions in this set.

Boutilier and Brafman (BB) (Boutilier and Brafman 1997) were the first to extend STRIPS-like languages to address interacting actions and to propose an extension of a standard planning algorithm to handle such domains. This technique was later formalized in an extension of PDDL3.1 to multi-agent planning (Kovacs 2012).

BB's extension to STRIPS is conceptually simple: in addition to a list of preconditions, an action a has a concurrency condition that specifies which actions must or must not be executed concurrently with a for a's effects to hold. Effects can also be conditional on which actions are executed concurrently. For example, consider an action for lifting the sides of a table. If performed by two agents on both sides, objects on the table will remain. But if performed by a single agent, the objects will fall. Thus, the action of lifting the left side of a table will have, beyond its regular preconditions and effects, a conditional effect with concurrent effect condition that states that when the action of lifting the right side is not performed concurrently, objects on the table will no longer remain on the table. Another example is box pushing – if the box is heavy and one agent pushes it, it remains in place. If two agents push it then it will move. Thus, the box movement is a conditional effect with a concurrency condition requiring another push action.

To deduce the effect of a joint action, one must take the union of the effects of the individual actions (assuming they are consistent), where the effect of each individual action takes into account the other actions performed.

BB's method is clean and clear semantically, but it has some potential shortcomings: 1. It introduces the additional, non-standard, concurrency condition. 2. The list of conditional effects when interactions are more involved – especially if the effects are non-monotonic in the number of agents – can be quite complex, and its consistency must be ensured (as when complicated conditional effects are used). 3. Action schema generally requires existential quantifiers in their specification. If the effect of action a changes when a' is concurrently executed, this usually holds for multiple instantiations of parameters of a'. These must be existentially quantified in the concurrency condition. For example, all actions have an agent parameters whose identity usually does not impact the interaction. 4. Their planning algorithm is based on partial order planning, a method not competitive with the state of the art.

Brafman and Zoran (2014) consider an alternative formulation in which actions involving multiple agents (which we call *collaborative actions* here) are specified (Brafman and Zoran 2014). That work was preliminary, did not carefully consider the issue of subsumed actions (discussed later) nor did it support concurrent actions that affect each other's preconditions. To support this new input language, the authors modified the MAFS algorithm (Nissim and Brafman 2014) by introducing new messages types. Our work provides a more careful and general definition and treatment of joint actions. Our compilation approach makes it easy to use off-the-shelf, state-of-the-art privacy-preserving planners, such as (Maliah, Shani, and Stern 2017), which is appealing from the engineering point of view. In addition, we support object cardinality constraints and interacting actions that modify each others' preconditions, and provide a more complete experimental evaluation.

Earlier work in knowledge representation considered the issue of concurrent actions, too, e.g., (Lin and Shoham 1992; Baral and Gelfond 1997; Poole 1997). These work focus on the representational issue only, and use non-monotonic formalisms, often within a rich first-order language such as the situation calculus (Reiter 1991). As noted earlier, such formalism are hard to integrate efficiently within modern planning algorithms. Of these formalisms, it is worth noting the action language $\mathcal{A}_c$ (Baral and Gelfond 1997) which we find the simplest and most intuitive. It is also closest semantically to our approach and uses propositional logic. In this language, the basic statements are of the form: "p is an effect of A if c". That is, if the actions in set A are executed concurrently in a state satisfying c then p will hold. This implies that p is also an effect of every $B \supseteq A$ given c, as long as there is no other set of actions D such that $B \supseteq D \supseteq A$ and $\neg p$ is an effect of D given c. This formalism, too, is non-monotonic – as you add actions to a set, effects that held for the subset may no longer be true. Our approach can be viewed as a monotonic variant of this semantics that forces the planner to be more explicit about the desired effects and restricts the extension of sets that might invalidate them.

Modeling Joint Actions

We start with an informal overview: The language used to describe a domain has two types of actions: *single-agent* actions and *collaborative* actions. *Single-agent* actions describe the effect of an action executed by a single-agent when no other action is executed concurrently. *Collaborative actions* describe the effect of a *set* of single-agent actions executed concurrently when no other actions are executed concurrently. A collaborative action must be specified whenever its effect is different from the union of the effects of the single-agent actions it contains.

Ultimately, we seek to define what happens when all agents act concurrently – i.e., *joint* actions. To this end, we define *multi-actions*. A *multi-action* is a set of single-agent and collaborative actions with consistent preconditions and consistent effects such that no agent participates in more than one of the actions in this set. The effect of a *well-defined* multi-action is the union of the effects of the actions it contains. We associate a *joint action* with every well-defined multi-action. The elements of the joint action are the single-agent actions contained in this multi-action (some of which are contained in collaborative actions), with *no-op*s added for any non-acting agent. The transition function induced by this joint-action is that of its underlying multi-action. While we require the four types of actions for our definitions, later we will use the term *action* to refer to a single-agent or a collaborative action, and drop the distinction between *multi-actions* and *joint actions*, simply using the former term.

To illustrate these concepts, consider a domain with three agents on a one dimensional grid with boxes, where each agent can either move, push a box out of the grid, or do nothing, and boxes can be light or heavy. The single-agent actions are *move-left*, *move-right, push, no-op*. When an agent pushes a heavy box, it does not move out of the grid, but when two agents push it, it does. To capture this, we add a collaborative action *2push* for every pair of agents. When b is heavy, the effect of *2push*(a_1, a_2, b) is thus different from the union of the effects of the *push*(a_1, b) and *push*(a_2, b) actions it consists of. A multi-action would be $\{$*move-left*(a_1),*move-right*$(a_2)\}$ or $\{$*move-left*(a_1),*2push*$(a_2, a_3, b)\}$. $\{$*move-left*(a_1),*push*(a_2, b),*push*$(a_3, b)\}$ is also a multi-action, but as we will see, it is not well-defined because some of its components can be replaced by a collaborative action – i.e., $\{$*push*(a_2, b),*push*$(a_3, b)\}$ can be replaced by *2push*(a_2, a_3, b). Finally, the multi-set $\{$*move-left*(a_1),*2push*$(a_2, a_3, b)\}$ corresponds to the joint-action (*move-left*(a_1),*push*(a_2, b),*push*(a_3, b)), and the multi-set $\{$*move-left*(a_1),*move-right*$(a_2)\}$ corresponds to the joint-action (*move-left*(a_1),*move-right*(a_2),*no-op*$_3$).

Language A multi-agent planning domain specification consists of $\langle P, I, g, \Phi, \{A_1, \ldots A_n\}, A_c \rangle$, where P is a set of ground propositions, $I \subset P$ is the initial state, $g \subset P$ is the goal condition, Φ is a set of agent names, A_i is a set of *single-agent* actions, and A_c is a set of *collaborative* actions.

A literal l is a, possibly negated, proposition from P, i.e. $l = p$ or $l = \neg p$ for some $p \in P$. Given a set of literals L, let $L^+ = \{p \in P | p \in L\}$ (the positive propositions in L), and let $L^- = \{p \in P | \neg p \in L\}$ (the negative propositions in L). L is well-defined if $L^- \cap L^+ = \emptyset$.

A *single-agent action* has the form $a = \langle symbol, pre(a), eff(a) \rangle$, where *symbol* is the action name, and pre(a) and eff(a) are well-defined sets of literals. pre$(a)^+$ is the set of positive pre-conditions, pre$(a)^-$ is the set of negative pre-conditions, eff$(a)^+$ is the set of add effects, and eff$(a)^-$ is the set of delete effects.

A *collaborative action* has the form $a_c = \langle symbol, pre(a), eff(a), e = \{a_1, \ldots, a_k\} \rangle$, where *symbol*, pre$(a)$ and eff(a) are as above, and e is a set of single-agent action symbols, such that no two action symbols in e belong to the same agent in Φ.

To simplify notation, clarity and reduce clutter we use the generic name *action* to refer to either a single-agent action or a collaborative action whenever possible; we will drop the distinction between an action and its symbol; and we write $e(a)$ to denote the elements of an action a. When a is a single-agent action, $e(a) = \{a\}$, and in a collaborative action $e(a)$ is simply e, the set of single-agent actions in a's definition. We also write $Agt(a)$ to denote the set of agents acting in a: $Agt(a) = \{\phi_i | \exists a_i \in e(a), a_i \in A_i\}$, i.e., agents for whom a contains an element from their action set.

Model Our formal semantic model is essentially a transition system. Transitions correspond to joint actions, and hence their special structure needs to be reflected. A *multi-agent planning model* $\langle S, A, s_0, G, \Phi, \{A_i : 1 \leq i \leq n\}\rangle$ is defined as follows: S is a set of states; A is a set of joint actions; $s_0 \in S$ is the initial state, $G \subseteq S$ is the set of goal states, Φ is the set of agents, where $|\Phi| = n$ by convention; and A_i are the single-agent action symbols for agent $\phi_i \in \Phi$, where A_i will always contain *no-op*$_i$. Every action $a \in A$ consists of a partial function from S to S and a vector $(a_1, \ldots, a_n)$ of single-agent action symbols. We write $e(a)$ to denote the single-agent action symbols associated with a. We write $a(s)$ to denote the state obtained when applying a in state s. A plan $\pi = a_1, a_2, \ldots, a_k$ is a sequence of joint actions such that $a_k(\cdots(a_1(s_0))) \in G$.

Interpretation The correspondence between the domain specification and the model is defined as follows: The set of states S corresponds to all possible truth assignments to P. We often equate a state with the list of propositions satisfied in it. Thus, s_0 is the state associated with I. G consists of all states containing the propositions in g.

Given a specification of actions, we define a *multi-action* to be a set of actions $a_m \subseteq A_c \cup (\cup_{i=1}^{n} A_i)$ such that (1) for every $a, a' \in a_m : Agt(a) \cap Agt(a') = \emptyset$, and (2) $\cup_{a \in a_m} pre(a)$ and $\cup_{a \in a_m} \mathit{eff}(a)$ are both well defined. Condition (1) ensures that no agent will be an actor in more than one action in a_m. Condition (2) ensures that the effects and the preconditions of the actions in a_m do not conflict. In that case pre$(a_m) = \cup_{a \in a_m}$pre(a) and eff$(a_m) = \cup_{a \in a_m}$eff(a).

Given a multi-action a, the result of applying a in s, $a(s)$, is well defined if $pre^+(a) \subseteq s$ and $pre^-(a) \cap s = \emptyset$. In that case, $a(s) = (s \setminus \mathit{eff}^-(a)) \cup \mathit{eff}^+(a)$.

We extend the notation e and Agt to multi-actions in the natural manner: $e(a_m) = \cup_{a \in a_m} e(a)$; $Agt(a_m) = \cup_{a \in a_m} Agt(a)$. We will refer to members of $e(a_m)$, which are all single-agent actions, as its *elements*, and to the actions in the set a_m as its *members*.

A key concept in the interpretation is that of a *well-defined* multi-action. Let $a_c = \{a_1, \ldots, a_l\}$ and $a'_c = \{a'_1, \ldots, a'_m\}$ be two multi-actions. We say that a_c is *subsumed* by a'_c if: (1) $e(a'_c) = e(a_c)$; (2) for every $a_i \in a_c$ there is some $a'_j \in a'_c$ such that $e(a_i) \subseteq e(a'_j)$; and (3) $m < l$. That is, both multi-actions involve the same set of elements, and moreover, for every member of a_c, there is a member of a'_c that contains all the elements of a_c, and that this containment is strict in at least one case (and hence, $m < l$).

The rationality behind this definition is as follows: because we want the effect of a multi-action to be the union of effects of its member actions, we do not want to allow multi-actions that contain multiple actions that can be combined into a single action, as then the corresponding joint action (defined below) will have an incorrect transition function. As an example, consider our box-pushing domain with 3 agents, and assume that in addition to *push* and *2push*, we also have a three-agent push, *3push*, whose elements are three *push* actions, one for each agent. The multi-action $\{$*push*(a_1, b),*push*(a_2, b),*push*$(a_3, b)\}$ has 3 elements, which are also its members. It is subsumed by $\{$*2push*(a_1, a_2, b),*push*$(a_3, b)\}$ which has the same set of elements but only two member actions, and *push*(a_1, b),*push*(a_2, b) are mapped into *2push*(a_1, a_2, b). Both are subsumed by $\{$*3push(b)*$\}$. However, neither $\{$*2push*(a_1, a_2, b),*push*$(a_3, b)\}$ subsumes $\{$*push*(a_1, b),*2push*$(a_2, a_3, b)\}$ nor does the latter subsume the former. Of all these multi-actions, only $\{$*3push(b)*$\}$ is well-defined, as the others are subsumed by it. If, on the other hand, we had only *push* and *2push*, but not *3push*, then $\{$*push*(a_1, b),*push*(a_2, b),*push*$(a_3, b)\}$ would not be well defined because it is subsumed by $\{$*2push*(a_1, a_2, b),*push*$(a_3, b)\}$. The latter, as well as $\{$*push*(a_1, b),*2push*$(a_2, a_3, b)\}$, would be well defined.

Now we can complete the definition of the interpretation. The definition of A_i for every $\phi_i \in \Phi$ is immediate, as the input contains a set of single-agent actions for each agent. To define the set of joint actions A, we first associate a joint action with every well-defined multi-action a_m – the joint action obtained by adding *no-op*$_i$ to a_m for every agent $\varphi_i \notin Agt(a_m)$. Now a_m associates a single-agent action with every agent in Φ, and we just need to present them as a vector. Thus, in our example domain, the multi-action $\{$*2push*$(a_2, a_3, b)\}$, which contains a single collaborative action, corresponds to the joint action (*no-op*$_1$, *push*(a_2, b), *push*(a_3, b)). We define the set of joint actions A in our model to be the set of all joint actions that correspond to some well-defined multi-action.

Note one potential weakness of this interpretation. In our example above, $e(\{$*2push*(a_1, a_2, b),*push*$(a_3, b)\})=$ $e(\{$*push*(a_1, b),*2push*$(a_2, a_3, b)\})$: in both multi-actions all three agents perform a *push* action. Yet it is possible that the effects of these multi-actions will be different. For example, the agent doing the individual *push* action may become tired, but the agents doing the *2push* actions will not. This is consistent with our semantic model, but ideally, we would have liked to associate a single joint-action with every vector of single-agent actions. This is a price we pay for simplicity and monotonicity (the ability to add elements without invalidating previous effects) and the onus is on the modeler to address it, much like ensuring consistency of complex conditional effects. There are two ways to address it: specifying enough collaborative actions, or restricting multi-actions. In our example above, if the effect of three *push* actions differs from the union of *2push* and *push*, the modeler must specify a *3push* action. Alternatively, the modeler can also restrict the number of agents that can manipulate an object using cardinality constraints. In our example, we can restrict the number of agents manipulating a box to at most two, in which case, the multi-action $\{$*2push*(a_1, a_2, b),*push*$(a_3, b)\}$ could not be considered by the planner. In future we hope to augment our software with a verification tool that will analyze a given domain and point out different multi-actions that have the same set of elements – allowing the user to modify the domain description in case this is problematic.

Finally, our input language is not really based on ground actions, but we take a PDDL view of planning in which actions are instantiated from action templates by replacing parameters with suitable objects. An example of the *2push* action template would be:

```
2push(?agt1,?agt2,?box,?loc1,?loc2)
  pre: at(?agt1,?loc1) & at(?agt2,?loc1)
       &  at(?box,?loc1) & heavy(?box);
  effect: NOT at(?agt1,?loc1) & at(?agt1,?loc2)
       &  NOT at(?agt2,?loc1) & at(?agt2,?loc2)
       &  NOT at(?box,?loc1) & at(?box,?loc2)
  elements: Push(?agt1,?box,?loc1,?loc2),
            Push(?agt2,?box,?loc1,?loc2)
```

Acting agents typically appear as parameters, as above, but nothing prevents the parameters from including other agents as well. Sometimes these agents will be actors, as in a collaborative action, and sometimes they can be passive objects.

Object Cardinality Constraints CJR's object cardinality constraints constrain the set of legal joint actions. Their intuition is very appealing – actions typically interact through joint objects, and the number of agents that can manipulate a set of objects concurrently is often constrained. For example, there is a maximal number of agents that can cross a bridge at one time, or there is a minimal number of agents that can use a boat at the same time – e.g., because at least two are required to sail the boat. According to the semantics of CJR, a legal joint action is any combination of single-agent actions that satisfies the cardinality constraints, and the union of the preconditions and the union of the effects of its contained actions is well defined. In the service of simplicity and space, we do not discuss these constraints formally in this paper. It is not hard to use CJR's ideas to extend our description to support them, and our implementation does this. See (Crosby, Jonsson, and Rovatsos 2014) for more details.

Planning With Multi-Actions

We now consider planning with multi-actions, separating the treatment into two cases: multi actions whose member actions do non-interact, i.e., no action adds or deletes a precondition of another action (recall – effects are not deleted by definition), and the more general case, referred to as multi-actions with pre/eff interactions.

Non-Interacting Actions If multi-actions containing interacting actions are not allowed, then all allowed interactions are already captured by the use of collaborative actions. Hence, the only benefit of performing them jointly is make-span reduction. That is, the set of states reachable with multi-actions and with (single-agent and collaborative) actions is identical. Thus, we can use any single-agent classical planning algorithm to solve the problem by combining single-agent and collaborative actions together to obtain a single-agent planning algorithm in which the agents are simply objects. Once a plan is obtained, a parallelization algorithm, such as CJR's, can be used to reduce make-span by concurrently executing non-interacting components.

Object cardinality constraints can be supported similarly. Maximum constraints can be enforced directly by the parallelization algorithm without modifying the domain model or the planning algorithm. Minimum cardinality constraint can be compiled away as follows: replace the single-agent actions with a collaborative action (or multiple collaborative actions in some cases) involving the minimal number of agents. For example, if exactly two people are required to cross a bridge, we remove the *cross* action, replacing it with a *2cross* collaborative action consisting of two *cross* actions. If two or more can cross the bridge, we remove *cross* and add both *2cross* and *3cross* actions: any number of agents > 2 can cross the bridge by sequencing multiple *2cross* and *3cross* actions without losing completeness, because if the actions do not interact, there is no difference between executing k *cross* actions followed by m *cross* actions versus doing all $k + m$ actions at the same time.

To summarize, we can address the multi-agent model of CJR by introducing collaborative actions that capture minimality constraints, and use the original domain with these added actions + post-processing. At most two action schema with an arity of 3 are required – leading a number of ground actions cubic in the number of agents.

Multi-Actions with Pre/Eff Interactions The above compilation scheme may become both unsound and incomplete when we allow multi-actions that contain actions that delete or add preconditions of other actions. Such action interactions seem natural when we consider true concurrency. For example, there is no reason we would want to exclude two agents from concurrently pushing a box, even though each push action deletes the preconditions of the other, as it changes the location of the box. Note that this is an issue regardless of collaborative actions. For example, if sailing a boat changes its location, then multiple agents cannot sail the boat if we do not allow multi-actions that destroy each others' preconditions. In some cases, the goal may be reachable with multi-actions of this kind, and unreachable without them. In that case, post-processing action sequences will not suffice, and we need to actively generate well-defined multi-actions. This requires a non-trivial compilation scheme.

If we allow actions that delete preconditions of each other, we must also address the subtle semantic issue of when do two actions conflict. If we allow a multi-action containing *sail(a1, boat, origin, destination)* and *sail(a2, boat, origin, destination)*, why should we not allow a multi-action containing *sail(a1, boat, origin, destination1)* and *sail(a2, boat, origin, destination2)*? Intuitively, we view the effects: *at(boat, destination1)* and *at(boat, destination2)* as inconsistent. While this would be clear with a multi-valued formulation of the problem, it is not obvious in the boolean case, as the two propositions are logically consistent. In single-agent planning such situations (e.g., *on(a,b), on(a,c)*) do not arise when the initial state is consistent and actions are formulated properly. But as evident, this is no longer true in the multi-agent case. Thus, in this paper we assume that additional declarative information about when actions conflict is provided. We will use this information to rule out actions with inconsistent effects. In our implementation, we handle this by adding additional cardinality constraints on concurrent actions. For example, we constrain the number of possible destinations of sail actions for the same object to 1.

The compilation scheme We now explain how to compile problems with collaborative actions into single-agent planning problems. This part can be viewed as extending the compilation of CJR to (1) Properly address pre/eff conflicts; (2) Support collaborative actions; and (3) Ensure that multi-actions are well-formed. Their basic idea was to represent a joint-action as a sequence with the same effects. As noted, when actions in a multi-action do not interact, this is relatively straightforward. Our compilation alters the action description so that such serialization can still work in the more general case.

The description below strives for simplicity, rather than economy, and ignores the handling of cardinality constraints, which is identical to CJR. Furthermore, CJR assume that every multi-agent action manipulates a particular set of objects. This has practical benefits, but we ignore this optimization to simplify the presentation.

Given a specification of a multi-agent planning problem $\langle P, I, g, \Phi, \{A_1, \dots A_n\}, A_c\rangle$, we generate the classical planning problem $\langle P_{Cl}, A_{Cl}, I_{Cl}, g_{Cl}\rangle$. In what follows let $A = A_1 \cup \cdots \cup A_n \cup A_c$.

- $P_{Cl} = P \cup P_{act} \cup P_{neg} \cup P_{pos} \cup P_{taken} \cup \{in\}$, where $P_{act} = \{p_a : a \in A\}$; $P_{neg} = \{Neg\text{-}p | p \in P\}$; $P_{pos} = \{Pos\text{-}p | p \in P\}$; and $P_{taken} = \{taken_i | \phi_i \in \Phi\}$. Intuitively, $p_a \in P_{act}$ tells us that the current multi-action contains a; P_{neg} and P_{pos} keep track of changes in the multi-action; P_{taken} keeps track of which agents are involved in the current multi-action; and *in* tells us that the current multi-action has not ended yet.
- $A_{Cl} = \{a' | a \in A\} \cup \{a_{start}, a_{end}\}$, where each $a' \in A_{Cl}$ is a modification of some $a \in A$, a_{start} makes in true, and a_{end} marks the end of a multi-action and does some book keeping and updates.
- $I_{Cl} = I$
- $g_{Cl} = g \wedge \neg in$

We explain the role of the additional variables and the changes in the actions below.

1. a_{start} : pre(a_{start}) = $\{\neg in\}$, eff(a_{start}) = $\{in\}$. Together with a_{end} (defined below) it marks the start and end of multi-action. Thus, the multi-actions $\{a_1, a_2, a_3\}$ will appear in the compiled plan as $a_{start}, a'_1, a'_2, a'_3, a_{end}$.
2. Every action $a \in A$ is modified as follows:
   (a) Every effect p and $\neg p$ is replaced by $Pos\text{-}p$ and $Neg\text{-}p$, respectively. This is used to allow actions in a multi-action that destroy each other's preconditions. In the compiled problem, instead of destroying p, we add $Neg\text{-}p$. a_{end} will update the value of P at the end of the multi-action based on the value of these propositions.
   (b) We add p_a to its effects. In addition, if a is a single-agent action, we add p_a to the effect of every collaborative action a_c such that $a \in e(a_c)$.
   (c) For every action a', if the effects of a and a' are inconsistent, we add $\neg p_{a'}$ as a precondition to a. Recall our earlier discussion of inconsistent effect – the user may need to explicitly express the fact that certain effects are inconsistent.
   (d) If $e(a_c) = \{a, a_1, \dots, a_k\}$ for some collaborative action a_c, add $\neg(p_{a_1} \wedge \cdots \wedge p_{a_k})$ as a precondition to a. This is done to ensure the generation of well-defined multi-actions.
   (e) To ensure no agent acts more than once in a multi-action, an additional effect of agent ϕ_i's action is $taken_i$ and an additional precondition is $\neg taken_i$.
3. a_{end} has the effect $\neg in$, to denote that a multi-action has ended; the conditional effect $Pos\text{-}p \rightarrow p$ and $Neg\text{-}p \rightarrow \neg p$ for every proposition p, to update the state with the effects of all elements of the multi-action; and the effect $\neg p_a$ for every $a \in A$ and $\neg taken_i$ for every agent, to reset these propositions; and the effect $\neg Pos\text{-}p$ and $\neg Neg\text{-}p$ for every $p \in P$, to reset these variables.

We stress again, that a more economical representation is possible, where multi-actions consider a fixed set of objects only. On the one hand, this requires multiple copies of various propositions and actions, but each action is much smaller. In particular, the *end* action for a specific object will have to update only propositions relevant to this set of objects and actions that manipulate this set of objects.

Formal Properties The main property of our compilation is that it is sound and complete:

Lemma 1. *Let $\Pi_S = \langle P, I, g, \Phi, \{A_1, \dots A_n\}, A_c\rangle$ be a specification of a multi-agent planning problem. Let $\Pi_M = \langle S, A, s_0, G, \Phi, \{A_i : 1 \leq i \leq n\}\rangle$ be the model it specifies. Let $\Pi_{Cl} = \langle P_{Cl}, A_{Cl}, I_{Cl}, g_{Cl}\rangle$ be the classical planning problem into which Π_S is compiled. Π_M is solvable iff Π_{Cl} is solvable.*

Lemma 2. *The size of the classical encoding is worst-case cubic in the size of the multi-agent planning problem.*

For the proofs, see the longer version of the paper on the corresponding author's home page.

Adding Privacy

Privacy Preserving Planning (PPP) (Nissim and Brafman 2014) supports agents that wish to plan collaboratively without revealing private information about their local state, their private actions, and their cost. For example, producers cooperating on a joint product will want to expose the capabilities they can contribute to the project, without necessarily revealing the identity of their suppliers and employees, their internal processes, and their inventory level. PPP algorithms are able to compute a joint-plan in a distributed manner without revealing private information. We now explain how to modify our specification and compilation technique to support PPP with interacting actions.

The input to a PPP problem differs from that of a centralized multi-agent planning problem: Each agent has a separate domain specification that contain a description of its actions. This specification also differentiates between *private* and *public* propositions and between *private* and *public* actions. A proposition may be private to an agent only if it does not appear in the description of actions of other agents. An action can be private only if its description contains private propositions only. Public actions may contain both private and public propositions. Their *public projection* is obtained by removing all private propositions from their description.

Modifying the Representation In PPP, the domain description of each agent contains: a complete description of all its actions and the public projection of the public actions of other agents; together with public propositions and propositions private to the agent. We extend this description with collaborative actions. While a collaborative action is public by definition, some of its preconditions or effects could be private to one of the agents. For example, the action *2push*(a_1, a_2, b) may have the precondition $healthy(a_1)$ private to a_1, and a private effect $tired(a_1)$. The description of *2push* in a_2's domain description will not include these preconditions and effects, i.e., they are projected out. Notice that while the specification is now distributed among n agents, the semantics remains the same.

PPP with Collaborative Actions Existing PPP algorithms are distributed, and this raises the question of when to insert a collaborative action into the plan. One agent cannot commit to a collaborative action in the name of another agent because it does not know if the private preconditions of that agent hold. (Brafman and Zoran 2014) added a special message between the agents to address this. We believe that splitting collaborative actions as follows is a simpler solution which allows us to use any existing PPP planner.

1. Add the precondition $\neg$*in-joint* to all existing public non-collaborative actions;
2. For each collaborative action a_m involving k agents:
   (a) Separate a_m into k actions $a_m^1, \ldots, a_m^k$, where a_m^i is obtained from a_m by removing the private preconditions and effects of agents other than ϕ_i. The parameters of each new action that do not appear in its preconditions or effects are removed.
   (b) Add to a_m^1 precondition $\neg$*in-joint* and effect *in-joint*.
   (c) Add to a_m^k the effect $\neg$*in-joint*.
   (d) For all $i < k$, add to a_m^i the effect *next*-a_m^{i+1}.
   (e) For all $i > 1$, add to a_m^i the precondition *next*-a_m^i and the effect $\neg next$-a_m^i.

For example, we split *2push*(a_1, a_2, *b*) into *2push*$_1$(a_1, a_2, *b*) and *2push*$_2$(a_1, a_2, *b*). If a_2 is not mentioned in the description of *2push*$_1$ and a_1 is not mentioned in the description of *2push*$_2$, we obtain *2push*$_1$(a_1, *b*) and *2push*$_2$(a_2, *b*). Thus, the first pushing agent need not commit to the identity of the second pushing agent. *2push*$_1$ will have $\neg$*in-joint* as a precondition and *in-joint* $\wedge$ *next*-$2push_2$ as an effect. *2push*$_2$ will have *next*-$2push_2$ as a precondition and $\neg$*in-joint* $\wedge\neg$*next-2push*$_2$ as an effect.

There remains one subtle issue when a MAFS-based algorithms (Nissim and Brafman 2014) is used. In MAFS, agents must end every sequence of private actions with a public action. Imagine that we attempt to insert into a plan, a collaborative action such as *2push*(a_1, a_2, *b*) that has two preconditions: p_1 is private to a_1 and p_2 is private to a_2, and these preconditions are initially *false*. Suppose that the first agent uses private action a_{p_1} to achieve p_1 and then applies *push*$_1$(a_1, *b*) (which is public and was split as described above). At this point a_2 cannot apply *push*$_2$(a_2, *b*) because p_2 does not hold. Suppose a_{p_2} is private and achieves p_2. We must allow a_2 to perform a_{p_2} before applying *push*$_2$(a_2, *b*). This is indeed possible because $\neg$*in-joint* is not a precondition of private actions. However, the actions now appear in the order: a_{p_1}, *push*$_1$(a_1, *b*), a_{p_2}, *push*$_2$(a_2, *b*). But collaborative actions must be executed in the same time, so we must push back all intermediate private actions to before the first part of the collaborative action, to obtain a_{p_1}, a_{p_2}, *push*$_1$(a_1, *b*), *push*$_2$(a_2, *b*). Because private actions of one agent do not interact with actions of other agents, such re-ordering does not impact the result of the plan, and is correct.

Domain	Ins (#agt)	Size	Length	Makespan	Time (sec)
Maze	P01 (3)	50	12	12	0.5
	P02 (4)	50	44	40	11.6
	P03 (4)	51	37	33	18.6
	P04 (5)	49	27	25	4.4
	P05 (4)	52	29	24	182.0
	P06 (5)	54	51	38	811.6
Table-Mover	P01 (3)	9	23	15	1.0
	P02 (4)	11	29	20	9.1
	P03 (4)	13	64	49	392.3
	P04 (5)	16	56	41	399.6
	P05 (5)	15	60	43	2753.5
Box-Pushing	P01 (3)	8	15	11	0.03
	P02 (4)	14	82	51	6.1
	P03 (4)	24	185	125	614.4
	P04 (5)	27	244	157	1002.4
	P05 (5)	27	229	154	1108.5
Aprt-Mover	P01 (3)	14	27	19	1.5
	P02 (4)	24	97	69	24.2
	P03 (4)	32	146	106	151.1
	P04 (5)	32	144	106	1102.9
	P05 (5)	36	179	136	1565.5

Table 1: Performance on compiled domains without privacy for single-agent version using FD.

Empirical Evaluation

As there are no implemented algorithms to compare against and no established domains with interacting actions, our contribution consists of defining a new set of domains and instances and evaluating the scalability of our approach. For each domain we generate a centralized version and a distributed version with private elements.

Domains We used four MA-PDDL domains in our experiments. Two are new, and two modify earlier domains.
Maze. This is a modified version of CJR's Maze domain. Agents on a 2D-grid must reach their respective goal location by crossing bridges, rowing a boat, or passing through a door. Only one agent can pass through a *door*. A *bridge* can be crossed by multiple agents, but it collapses after the first crossing. A *boat* requires at least two agents to row. To open a locked door, a *switch*, present at an arbitrary location in the grid, must be pushed. We added a collaborative action (*2row*) and used the cardinality constraints. In the PPP settings, an agent's location is private.
TableMover. A complex domain with a number of tables and rooms. Each table has something fragile on top. The goal is to place each table in its destination while keeping objects intact. If an agent lifts or drops a table alone, objects on it are no longer intact. The two-agent collaborative actions *lift-table* and *drop-table* lift and drop the table respectively, and keep things on top of it intact. Agents can move a table only if they are charged. Charging points are available only in some rooms.
BoxPushing. A modified version of a domain from (Braf-

Ins (#agt)	Time (sec)	Length	Makespan	SA Time
P_{dp}01 (3)	62.2	3	1	8.0
P_{dp}02 (4)	270.1	25	22	10.7
P_{dp}03 (4)	104.8	17	10	13.0
P_{dp}04 (5)	407.9	17	10	15.7
P_{dp}05 (5)	142.7	26	15	18.4
P_{dp}01 (3)	2.9	7	5	0.7
P_{dp}02 (4)	23.1	20	16	2.8
P_{dp}03 (4)	65.5	28	22	5.6
P_{dp}04 (5)	**TO**	–	–	4.6
P_{dp}05 (5)	**TO**	–	–	39.9
P_{dp}01 (3)	2.9	8	4	0.7
P_{dp}02 (4)	5.72	12	7	2.4
P_{dp}03 (4)	322.2	26	16	321.0
P_{dp}04 (5)	766.6	25	18	420.9
P_{dp}05 (5)	1497.1	39	25	691.3
P_{dp}01 (3)	12.8	16	10	15.7
P_{dp}02 (4)	241	31	23	47.0
P_{dp}03 (4)	301.0	36	28	35.0
P_{dp}04 (5)	254.5	44	38	200.0
P_{dp}05 (5)	781.6	26	20	160.9

Table 2: GPPP's performance on compiled domains with privacy. SA Time: time of centralized solution with no privacy.

man and Zoran 2014). A set of boxes must be pushed to their goal positions. Action *push* shifts the box to a connected position. An agent pushing a box alone will hurt her back and cannot perform additional *push* actions. But when using the collaborative action (*2push*) the box moves without this side effect. An agent's location and whether her back hurts are private propositions.

ApartmentMover. A domain based on the classical *Depot* domain. Agents move furnitures and electronics between apartments *via* trucks. Electronic items are fragile and must be packed first. A packed box requires exactly two agents to load and unload, and there is a two-agent collaborative actions for this. Driving a truck requires exactly two agents. An agent can *walk* between two connected locations, and it becomes tired after executing a public action, *e.g.*, *drive-truck*. The location of an agent and whether she is tired is private.

Results Experiments were carried out on an Intel Core i5 3.20 GHz with 64-bit processor and 4GB of RAM. A time limit of 30 minutes was set per problem. Our translation scheme generates a standard single-agent planning problem if there is no privacy involved in the domain description which is identical to that of CJR's method's output when there are no pre/eff interactions and privacy, except for a small overhead associated with maintaining data-structures required to detect and correctly handle inputs with *Pre*/*Eff* Interactions. Hence, there is no point comparing the two. Their implementation overlooks *Pre*/*Eff* Interactions and can return incorrect solutions or miss valid solutions when such interactions exist.

Table 1 shows how our compilation algorithm scales in each domain with no privacy. For each problem, the translation step generates a single-agent problem, which is given to Fast-Downward (FD) (Helmert 2006). The search approach used in FD is *lazy-greedy* with h_{FF} heuristic (Hoffmann and Nebel 2001). *Size* refers to the number of objects, including agents. As can be seen in the *Maze* domain, it is not always correlated with the difficulty of planning. Indeed, in the *Maze* domain, the nature of the objects and how they constrain the possible path plays a more significant role.

Dom	Time	Original			Compiled	
		Predicate	Total Action		Predicate	Action
			SA	Collab.		
Maze	0.003	7	4	1	22	10
TM	0.002	8	5	3	32	17
BP	0.001	4	2	1	20	8
AM	0.007	13	7	4	57	29

Table 3: Original *vs* compiled domains.

Compilation time is negligible, while the size of the translated domain is roughly four times larger. This information is described in Table 3.

In *Maze*, we used CJR's random problem generator. Problems P03 and P04 have similar complexities with different number of agents. P04, with more agents, has a much reduced run-time. P05 is similar to P04, but with more switches, boats, and doors, but fewer bridges. This results in much larger running time, though similar plan length and makespan. *Table-Mover* is a complex domain, and the compilation does not scale well. For example, P04, involves 7 rooms, 4 tables, and 5 agents. In P05, we increase the number of rooms to 8, place each table in a different room initially, and decrease the number connections between them by 2, and the planner exceeds our time bound (but solves the problem in 2753.5 seconds). Memory is also an issue in this domain, unlike *Box-Pushing* and *Apartment-Mover*, where FD generates plans with 200 steps approximately, well within the time bound.

Table 2 shows results for distributed PPP, for which we used the distributed PPP solver GPPP (Maliah, Shani, and Stern 2017) with the distributed h_{FF} heuristic. This planner is far less optimized than FD (on single-agent problems it was 123 times slower, with average ratio per domains ranging from 40 to 287). Hence, we used simpler problems than in Table 1. For each problem, beyond showing the results of running GPPP on the compiled problem, we also describe (last column) the running time when the problem is solved by GPPP with a single-agent (that has access to all actions). This gives a sense of the relative difficulty associated with privacy, which we can see is non-negligible, generally between 4-20 times slower. The gap in *Table-Mover* is largest, and *Box-Pushing* and *Apartment-Mover*, smallest.

Summary

Building on earlier work, we presented a new approach to modeling and planning with interacting actions and privacy that is both intuitive and supports efficient planning. We described a compilation scheme from our input language to single-agent planning and distributed PPP, which is the first to extend both the language and algorithms of classical planning to handle these issues.

Acknowledgments We thank the reviewers for their useful comments. This work was supported by ISF Grant 933/13, by the Israel Ministry of Science and Technology Grant 54178, by the Helmsley Charitable Trust through the Agricultural, Biological and Cognitive Robotics Center of Ben-Gurion University of the Negev, and by the Lynn and William Frankel Center for Computer Science

References

Baral, C., and Gelfond, M. 1997. Reasoning about effects of concurrent actions. *J. Log. Program.* 31(1-3):85–117.

Blum, A., and Furst, M. L. 1997. Fast planning through planning graph analysis. *Artificial Intelligence* 90:281–300.

Boutilier, C., and Brafman, R. I. 1997. Planning with concurrent interacting actions. In *Proc. of the 14th National Conference on AI (AAAI '97)*, 720–726.

Brafman, R. I., and Zoran, U. 2014. Distributed heuristic forward search with interacting actions. In *Proc. of the 2nd ICAPS Workshop on Distributed and Multi-Agent Planning*.

Crosby, M.; Jonsson, A.; and Rovatsos, M. 2014. A single-agent approach to multiagent planning. In *ECAI 2014 - 21st European Conference on Artificial Intelligence, 18-22, August 2014, Prague, Czech Republic - Including PAIS 2014*, 237–242.

Dimopoulos, Y.; Nebel, B.; and Koehler, J. 1997. Encoding planning problems in nonmonotonic logic programs. In *Recent Advances in AI Planning, 4th European Conference on Planning, ECP'97, Toulouse, France, September 24-26, 1997, Proceedings*, 169–181.

Helmert, M. 2006. The fast downward planning system. *J. Artif. Int. Res.* 26(1):191–246.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: fast plan generation through heuristic search. *J. Artif. Int. Res.* 14(1):253–302.

Kovacs, D. L. 2012. A multi-agent extension of PDDL3.1. In *Proc. of the 3rd Workshop on the International Planning Competition (IPC-2012)*.

Lin, F., and Shoham, Y. 1992. Concurrent actions in the situation calculus. In *Proceedings of the 10th National Conference on Artificial Intelligence. San Jose, CA, July 12-16, 1992.*, 590–595.

Maliah, S.; Shani, G.; and Stern, R. 2017. Collaborative privacy preserving multi-agent planning - planners and heuristics. *Autonomous Agents and Multi-Agent Systems* 31(3):493–530.

Nissim, R., and Brafman, R. I. 2014. Distributed heuristic forward search for multi-agent planning. *J. Artif. Intell. Res.* 51:293–332.

Poole, D. 1997. The independent choice logic for modelling multiple agents under uncertainty. *Artificial Intelligence* 94(1-2):7–56.

Reiter, R. 1991. The frame problem in the situation calculus: a simple solution (sometimes) and a completeness result for goal regression. In Lifshitz, V., ed., *AI and Mathematical Theory of Computation: papers in honour of John McCarthy*. 359–380.

Rintanen, J.; Heljanko, K.; and Niemelä, I. 2006. Planning as satisfiability: parallel plans and algorithms for plan search. *Artif. Intell.* 170(12-13):1031–1080.