Planning with Complex Actions

Sheila McIlraith
Knowledge Systems Laboratory
Department of Computer Science
Gates Building, 2A wing
Stanford University
Stanford, CA 94305, USA.
sam@ksl.stanford.edu

Ronald Fadel
Knowledge Systems Laboratory
Department of Computer Science
Gates Building, 2A wing
Stanford University
Stanford, CA 94305, USA.
rfadel@ksl.stanford.edu

Abstract

In this paper we address the problem of planning with complex actions. We are motivated by the problem of automated Web service composition, in which planning *must* be performed using predefined complex actions or services as the building blocks of a plan. Planning with complex actions is also compelling in primitive action planning domains because it enables the exploitation of reusable subplans, potentially improving the efficiency of planning. This paper provides a formal, semantically-justified account of how to plan with complex actions using operator-based planning techniques. A key contribution of this work is the definition, characterization, and computation of preconditions and conditional effects for complex actions. While we use the situation calculus and Golog to formalize the task and our solution, the results in this paper are directly applicable to most action theories and planning systems. In particular, we have developed a PDDL-equivalent compiler that computes the preconditions and effects of complex actions, thus enabling wide-spread use of these results. Finally we provide an approach to planning that enables us to exploit deductive plan synthesis or alternatively ADL planners to plan with complex actions. Our approach to complex-action planning is sound and complete relative to the corresponding primitive action domain.

1 Introduction

Given a description of an initial state, a goal state, and a set of actions, the planning task is to generate a sequence of actions that, when performed starting in the initial state, will terminate in a goal state. Typically, actions are primitive and are described in terms of their precondition, and (conditional) effects. Our interest is in planning with complex actions as the building blocks for a plan. Complex actions are actions composed of primitive actions using typical programming language constructs. E.g.,complex actions move(obj,orig,dest) and goToAirpt(loc) are defined as:

move(obj,orig,dest) $\doteq$[1] pickup(obj,orig);putdown(obj,dest)
goToAirpt(loc) $\doteq$ **if** loc=Univ **then** shuttle(Univ,PA);
train(PA,MB);shuttle(MB,SFO) **else** taxi(loc,SFO)

Our primary motivation for investigating complex action planning is to automate *Web service composition* (e.g., [13]). Web services are self-contained Web-accessible computer programs, such as the airline ticket service at www.ual.com, or the weather service at www.weather.com. These services can be conceived as complex actions. Consider ual.com's buyAirTicket($\vec{x}$) service. This service can be described as the complex action locateFlight($\vec{x}$); **if** Available($\vec{x}$) $\wedge$ OKPrice($\vec{x}$) **then** buyAirTicket($\vec{x}$);...[2]. The task of automated Web service composition is to automatically sequence together Web services such as buyAirTicket($\vec{x}$) or getWeather($\vec{y}$) into a composition that achieves some user-defined objectives. The task of automated Web service composition is, by necessity, a problem of planning with complex actions. But how do we represent these complex actions (Web services) and how do we plan with them?

What makes planning with complex actions difficult is that the traditional characterization of actions as operators with preconditions and effects does not apply, making operator-based planning techniques such as Blackbox, FF, GraphPlan, BDDPlan, etc., inapplicable, at least at face value. In this paper we provide a formal, semantically-justified account of how to characterize, represent and precompile the preconditions and effects of complex actions, such as buyAirTicket($\vec{x}$), under a frame assumption [16]. This enables us to treat complex actions such as buyAirTicket($\vec{x}$) as planning operators and to apply standard planning tech-

[1] Denotes "defined as."
[2] Example is simplified for illustration purposes.

niques to planning with complex actions. Planning results in a plan in terms of complex actions from which a plan in terms of primitive actions is easily expanded, if desired[3].

A secondary motivation for this work is to improve the efficiency of planning by representing useful (conditional) plan segments as complex actions. As we show, our approach to planning with complex actions can dramatically improve the efficiency of plan generation by reducing the search space size and the length of a plan.

The idea of planning with some form of abstraction or aggregation is not new, and there has been a variety of work in this area including ABStrips (e.g., [17]), planning with macro-operators (e.g., [11] and [6]), and most notably HTN planning (e.g., [5]). Our work is fundamentally different from these approaches, and in particular from HTN planning, both in terms of i) the representation of complex actions (aka HTN *non-primitive tasks*), and ii) the method of planning. In this paper we precompile complex actions into planning operators described in terms of preconditions and effects that embody all possible evolutions of the complex action. In contrast, HTN planners do not use a declarative representation of the preconditions and effects of tasks. Rather, methods are associated with tasks, and tasks are pre-arranged into a network of compositions, without the full programming constructs we use to describe complex actions [18]. Further, HTN planners operate by searching for plans that accomplish task networks using task decomposition and conflict resolution. In contrast, having precompiled our complex actions, we can apply standard operator-based planning techniques to generate a plan, followed by plan expansion.

Our work is somewhat similar in methodology to [2], which proposes to encode planning constraints by compiling the constraints together with the original planning problem into a new unconstrained problem. The resultant planning problem can be solved using classical planning methods, and the resultant plan *decompiled* to provide a solution in the original problem domain. The general methodology of compilation and subsequent expansion is similar to what we propose. Nevertheless, the general problem is different. We are compiling complex actions into new plan operators. These complex actions represent Web services that we wish to reason with as black-box components. The constraints used in [2] are constraints upon the domain, and thus capture different types of planning information than our more procedural complex actions. Further the formal treatment and results are different.

We also contrast our work to the use of Golog (e.g., [12]) in planning. In this paper we use Golog as the formal language to describe complex actions, however the role these actions play in planning is very different. Golog complex actions are traditionally used to specify non-deterministic programs. In combination with deductive plan synthesis [7], a Golog program expands to a situation calculus formula which constrains the search space for a plan. This is similar to the role of domain-specific knowledge, as exemplified by systems such TALPlanner [4], BDDPlan [10] and ASP [18]. In all these systems, complex actions constrain the search space, but are not used as operators in plan construction.

The research presented in this paper is of both theoretical and practical significance. From a theoretical standpoint, we provide a semantically-justified means of characterizing the preconditions, effects and successor situations of complex actions under a frame assumption, that embodies all possible trajectories of a complex action. This enables us to not only use operator-based planning methods to plan with complex actions, but also to prove formal properties of our approach. In particular, we prove that our approach to planning is sound and complete relative to corresponding primitive action domains. From a practical perspective, analysis shows a significant increase in the efficiency of planning with complex actions, relative to primitive action planning. We illustrate potential speedup with some experiments on the briefcase domain, using the FF planner ([9]). Finally, this paper provides a principled approach to automating Web service composition, that has far-reaching application to automated component-based software composition

2 Background: Situation Calculus & Golog

We use the situation calculus and Golog to formalize the task and our solution. The expressive power and formal semantics of the situation calculus provide the theoretical foundations for our work, and for the later translation to PDDL.

Briefly, the situation calculus is a logical language for specifying and reasoning about dynamical systems [16]. In the situation calculus, the state of the world is expressed in terms of functions and relations (fluents) relativized to a particular situation s, e.g., $F(\vec{x}, s)$. A situation s is a history of the primitive actions, e.g., a, performed from an initial, distinguished situation S_0. The function $do(a, s)$ maps a situation and an action into a new situation. A situation calculus theory $\mathcal{D}$ comprises the following sets of axioms:

- domain independent foundational axioms, Σ.
- successor state axioms, $\mathcal{D}_{SS}$, one for every fluent F.
- action precondition axioms, $\mathcal{D}_{ap}$, one for every action a in the domain,which define $Poss(a, s)$.
- axioms describing the initial situation, $\mathcal{D}_{S_0}$.

[3] For many Web service applications, expansion is not relevant.

• unique names axioms for actions, $\mathcal{D}_{una}$.

Successor state axioms, originally proposed [15] to address the frame problem, are created by compiling effect axioms into axioms of this form[4]: $F(\vec{x}, do(a,s)) \equiv \Phi_F(\vec{x}, a, s)$ where $\Phi_F(\vec{x}, a, s) = \gamma_F^+(\vec{x}, a, s) \vee (F(\vec{x}, s) \wedge \neg\gamma_F^-(\vec{x}, a, s))$. (See [16, pg.28-35] for details.)

Example: In the interest of simplicity, we illustrate concepts in this paper in terms of an action theory with three actions $pickup(x), putdown(x)$ & $drop(x)$, and three fluents $holding(x), broken(x)$ & $hot(x)$. (1)-(3) comprise $\mathcal{D}_{ap}$, and (4)-(6) comprise $\mathcal{D}_{SS}$[5].

$$Poss(pickup(x), s) \equiv \neg holding(x, s) \quad (1)$$
$$Poss(drop(x), s) \equiv holding(x, s) \quad (2)$$
$$Poss(putdown(x), s) \equiv holding(x, s) \quad (3)$$
$$holding(x, do(a, s)) \equiv a = pickup(x) \vee holding(x, s) \wedge a \neq putdown(x) \wedge a \neq drop(x) \quad (4)$$
$$broken(x, do(a, s)) \equiv a = drop(x) \vee broken(x, s) \quad (5)$$
$$hot(x, do(a, s)) \equiv hot(x, s) \quad (6)$$

Golog (e.g., [12, 16, 3]) is a high-level logic programming language for the specification and execution of complex actions in dynamical domains. It builds on top of the situation calculus by providing extralogical constructs for assembling primitive situation calculus actions, into complex actions δ. [3] shows how these complex actions can be considered to be first-class objects in the language. $Do(\delta, s, s')$ is an abbreviation that macro-expands into a situation calculus formula, as defined inductively below. The formula says that it is possible to reach s' from s by executing a sequence of actions specified by δ [16].

Prim. action: $Do(a, s, s') \doteq Poss(a, s) \wedge s' = do(a[s], s)$
Test: $Do(\phi?, s, s') \doteq \phi[s] \wedge s = s'$
Seq.: $Do(\delta_1; \delta_2, s, s') \doteq \exists s''.Do(\delta_1, s, s'') \wedge Do(\delta_2, s'', s')$
Nondet. act.: $Do(\delta_1 \mid \delta_2, s, s') \doteq Do(\delta_1, s, s') \vee Do(\delta_2, s, s')$
Nondet. arg.: $Do((\pi x)\delta(x), s, s') \doteq \exists x.Do(\delta(x), s, s')$

The construct, **if** ϕ **then** δ_1 **else** δ_2 **endIf** is defined as $[\phi?; \delta_1] \mid [\neg\phi?; \delta_2]$. The Golog language also includes nondeterministic iteration, δ^*, which executes δ zero or more times. The while-loop construct,**while** ϕ **do** δ **endWhile** is defined in terms of nondeterministic iteration as $[\phi? : \delta]^*; \neg\phi?$. For now, we exclude nondeterministic iteration, and while-loops, whose macro-expansions are second order, and which may be non-terminating. Instead, we define a bounded notion of while, **while** $_k(\phi)$ δ, which is guaranteed to terminate, and is commonly used in Web services. **while** $_k(\phi)$ δ executes like the original while-loop except that it loops at most k times, even if ϕ still holds after the k^{th} iteration. Formally, **while** $_k(\phi)$ δ corresponds to k conditional branchings as follow:

$$\textbf{while}_1(\phi)\ \delta \doteq \textbf{if}\ \phi\ \textbf{then}\ \delta\ \textbf{endIf}\ [6] \quad (7)$$
$$\textbf{while}_k(\phi)\ \delta \doteq \textbf{if}\ \phi\ \textbf{then}\ [\delta; \textbf{while}_{k-1}(\phi)\delta]\ \textbf{endIf} \quad (8)$$

[4]For space, we will only consider relational fluents here.

[5]Notation: formulae are universally quantified with maximum scope unless noted. Action arguments suppressed.

A deterministic version of the choice construct (π') is defined in a longer paper. These constructs are used to specify complex actions such as buyAirTicket($\vec{x}$) or goToAirpt(loc). Traditional usage of Golog is to apply deductive plan synthesis to find a sequence of actions $\vec{a} = [a_1, \ldots, a_n]$ that realizes a Golog program, δ relative to domain theory, $\mathcal{D}$. I.e., $\mathcal{D} \models Do(\delta, S_0, do(\vec{a}, S_0))$. $Do(\delta, S_0, do(\vec{a}, S_0))$ denotes that the Golog program δ, starting execution in S_0 will legally terminate in situation $do(\vec{a}, S_0)$, where $do(\vec{a}, S_0)$ is an abbreviation for $do(a_n, do(a_{n-1}, \ldots, do(a_1, S_0)))$.

3 Problem: Planning with Complex Actions

Given a set of primitive actions, $\mathcal{A}$ together with an associated set of complex actions, $\Delta_{\mathcal{A}}$, our objective is to use an operator-based planner to compose complex and primitive actions to achieve some goal. To do this, we must characterize the preconditions, effects, and the situation resulting from performing a complex action.

3.1 Preconditions, Effects, Resulting Situations

For analysis, our actions $\mathcal{A}$ are axiomatized in a situation calculus action theory $\mathcal{D}$, and our complex actions $\Delta_{\mathcal{A}}$ are described in Golog. For now, we restrict our focus to terminating complex actions described in Section 2.

Resulting Situation: We wish to characterize the situation resulting from performing the complex action δ. Observe that many complex actions are nondeterministic. They may have several different executions, each terminating in a different situation. As such, we can't define a function analogous to $do(a, s)$. Instead, we introduce the abbreviation $do_{ca}(\delta, s)$ to denote a situation resulting from performing complex action δ in s. $do_{ca}(\delta, s)$ ranges over the set of *executable* situations and corresponds to a so-called *ghost situation* [16, pg.52-53], when δ is not physically realizable. The interpretation of $do_{ca}(\delta, s)$ is constrained by the following axiom, which is added to $\mathcal{D}$ producing theory $\mathcal{D}_{ca}$.

For all complex actions δ and situations s:

$$Do(\delta, s, do_{ca}(\delta, s)) \vee (\neg\exists s''.Do(\delta, s, s'') \wedge \neg executable(do_{ca}(\delta, s))) \quad (9)$$

where $executable(s)$ denotes a situation, all of whose actions in the situation action history are $Poss$-ible [16]. I.e., $executable(s) \doteq (\forall a, s^*).do(a, s^*) \sqsubseteq {}^7 s \rightarrow Poss(a, s^*)$ It follows that:

$$\mathcal{D}_{ca} \models \forall s.executable(s) \wedge Do(\delta, s, do_{ca}(\delta, s)) \rightarrow executable(do_{ca}(\delta, s)) \quad (10)$$

[6]**if -then -endIf** is the obvious variant of **if -then -else -endIf**.

[7]The order relation on situations in the situation tree [16].

Preconditions: $Poss_{ca}(\delta, s)$ denotes the preconditions of complex action δ. Intuitively, the preconditions of a complex action are the preconditions of all the actions that make up the execution of δ. E.g., for $a_1; a_2$,

$$Poss_{ca}(a_1; a_2, s) \equiv Poss(a_1, s) \wedge Poss(a_2, do(a_1, s)).$$

This is captured tidily in the inductive definition of Do. We define the precondition of complex action δ , $Poss_{ca}(\delta, s)$ as:

$$Poss_{ca}(\delta, s) \equiv \Pi^*_\delta(s) \quad (11)$$

where $\Pi^*_\delta(s) \equiv \exists s'.Do(\delta, s, s')$. These are *intermediate* action precondition axioms.

Proposition 1 (Properties of $Poss_{ca}(\delta, s)$)
These axioms follow from $\mathcal{D}_{ca} \cup (11)$.

$$\begin{aligned} Poss_{ca}(\delta, s) &\equiv Do(\delta, s, do_{ca}(\delta, s)) \\ executable(s) \wedge Poss_{ca}(\delta, s) &\equiv executable(do_{ca}(\delta, s)) \end{aligned}$$

Effects: Intuitively the effects of a complex action are the effects of each action in the execution of δ, modulo the effects of subsequent actions. We assume that fluents whose truth value is not changed by an action, persist. $F(\vec{x}, do_{ca}(\delta, s))$ denotes that fluent F is true in the situation resulting from performing complex action δ in s. We capture the effects of complex actions as successor state axioms. Since all but trivial complex actions involve multiple intermediate situations, strictly speaking, we cannot define successor state axioms for complex actions. Rather, we define the notion of a pseudo-successor state axiom. Here we define *intermediate* pseudo-successor state axioms, making them "Markovian" in the section to follow via regression.

$$Poss_{ca}(\delta, s) \rightarrow [F(\vec{x}, do_{ca}(\delta, s)) \equiv \Phi^*_F(\vec{x}, \delta, s)], \text{ where,}$$
$$\Phi^*_F(\vec{x}, \delta, s) \equiv \exists s'.Do(\delta, s, s') \wedge F(\vec{x}, s') \wedge s' = do_{ca}(\delta, s). \quad (12)$$

We need the $s' = do_{ca}(\delta, s)$ since some complex actions are nondeterministic. This enables us to identify the particular sequence of actions in the instantiation of the complex action that leads to the truth/falsity of the fluent F.

3.2 Pseudo-Markovian Complex Actions

In order to plan with complex actions as operators, we must make our characterization *pseudo-markovian*. That is, we wish to characterize the preconditions strictly in terms of the situation in which the complex action execution is initiated, and the effects, strictly in terms of the initiating and terminating situations of the complex action. To do so we appeal to regression rewriting [19], regressing over the successor state axioms for the *primitive actions* in our domain theory $\mathcal{D}$. Unfortunately, the formulae over which we need to regress are not, by definition, regressable using $\mathcal{R}$ [16, pg.62], since we are not regressing to S_0, and since the macro-expansion of $Do(\delta, s, s')$ does not yield a nested representation of situations. Since regression is a syntactic rewriting, this is problematic. We define a suitable (small) variant of Reiter's regression operator, $\mathcal{R}^s$, that first rewrites the macro-expansion of Do so that situations are expressed as nested do's, and that enables regression to an arbitrary situation s, rather than to S_0. We define the preconditions and effects of δ in terms of a set of action precondition axioms, $\mathcal{D}_{caap}$, of the form of (13) and a set of pseudo-successor state axioms, $\mathcal{D}_{caSS}$, of the form of (15).

Preconditions:
Action Precondition Axioms, $\mathcal{D}_{caap}$, one for every $\delta \in \Delta$:

$$Poss_{ca}(\delta, s) \equiv \Pi_\delta(s) \quad (13)$$

where $\Pi_\delta(s) \equiv \mathcal{R}^s[\Pi^*_\delta(s)]$ from (11), i.e. $\mathcal{R}^s[\exists s'.Do(\delta, s, s')]$.

Example (continued): Consider the complex action $pickup(x)$; **if** $hot(x)$ **then** $drop(x)$ **else** $putdown(x)$ **endIf**, which we denote as δ_1 for parsimony. Its action precondition axiom is defined as follows.

$$\begin{aligned} &Poss_{ca}(\delta_1, s) \equiv \\ &\quad \mathcal{R}^s[\exists s', s''.Poss(pickup(x), s) \wedge s'' = do(pickup(x), s) \wedge \\ &\quad ((hot(x, s'') \wedge Poss(drop(x), s'') \wedge s' = do(drop(x), s'')) \\ &\quad \vee (\neg hot(x, s'') \wedge Poss(putdown(x), s) \wedge \\ &\quad s' = do(putdown(x), s''))] \end{aligned} \quad (14)$$

Following our regression, $Poss_{ca}(\delta_1, s) \equiv \neg holding(x, s)$.

Successor State Axioms: Observe that while a situation calculus axiomatization has one successor state axiom for every fluent, we currently define one pseudo-successor state axiom for every fluent-complex action pair.

Pseudo-Successor State Axioms, $\mathcal{D}_{caSS}$, one for every fluent-complex action pair:

$$Poss_{ca}(\delta, s) \rightarrow [F(\vec{x}, do_{ca}(\delta, s)) \equiv \Phi_F(\vec{x}, \delta, s)] \quad (15)$$

where $\Phi_F(\vec{x}, \delta, s) \equiv \mathcal{R}^s[\Phi^*_F(\vec{x}, \delta, s)]$, $\mathcal{R}^s[\Phi^*_F(\vec{x}, \delta, s)] \equiv \mathcal{R}^s[\exists s'.Do(\delta, s, s') \wedge F(\vec{x}, s') \wedge do_{ca}(\delta, s) = s']$

Example (continued): The pseudo-successor state axiom for fluent $broken(x, do_{ca}(\delta_1, s))$ is:

$$\begin{aligned} &Poss_{ca}(\delta_1, s) \rightarrow [broken(x, do_{ca}(\delta_1, s)) \equiv \\ &\quad \mathcal{R}^s[\exists s', s''.Poss(pickup(x), s) \wedge s'' = do(pickup(x), s) \wedge \\ &\quad ((hot(x, s'') \wedge Poss(drop(x), s'') \wedge s' = do(drop(x), s'')) \\ &\quad \vee (\neg hot(x, s'') \wedge Poss(putdown(x), s) \wedge \\ &\quad s' = do(putdown(x), s'))) \wedge \\ &\quad broken(x, s') \wedge do_{ca}(\delta_1, s) = s'] \end{aligned} \quad (16)$$

Applying our $\mathcal{R}^s$ regression operator, (16) becomes:

$$\begin{aligned} &Poss_{ca}(\delta_1, s) \rightarrow (\ broken(x, do_{ca}(\delta_1, s)) \equiv \\ &\quad \neg holding(x, s) \wedge [hot(x, s) \wedge \\ &\quad do_{ca}(\delta_1, s) = do(drop(x), do(pickup(x), s)) \\ &\quad \vee\ \neg hot(x, s) \wedge broken(x, s) \wedge \\ &\quad do_{ca}(\delta_1, s) = do(putdown(x), do(pickup(x), s))]\ ) \end{aligned}$$

Though the computation looks complex, regression rewriting is a powerful tool and the final pseudo-successor state axiom is simple. Observe that a pseudo-successor state axiom not only defines the conditions under which fluent F is true after performing complex action δ, but it also defines the action trajectory upon which the truth of F is predicated. This is most valuable with nondeterministic actions.

Note that when the definition of $Poss_{ca}(\delta, s)$ and the intermediate pseudo-successor state axiom, ((11) and (12), respectively) are conjoined to $\mathcal{D}_{ca}$, they entail the complex action precondition axioms and the complex action pseudo-successor state axioms.

Proposition 2 $\mathcal{D}_{ca} \cup (11) \cup (12) \models \mathcal{D}_{caap} \cup \mathcal{D}_{caSS}$

Effect axioms: While we have encoded the effects of our complex actions, together with a solution to the frame problem in terms of pseudo-successor state axioms, many planners use effect axioms, rather than successor state axioms, solving the frame problem in the procedural code of their planner, rather than representationally. Hence, for completion we define effect axioms for complex actions, $\mathcal{D}_{caef}$.

Effect Axioms $\mathcal{D}_{caef}$, up to one positive effect axiom and one negative effect axiom for every fluent - complex action pair, where the execution of δ can potentially change the truth value of fluent $F \in \mathcal{F}$:

$$Poss_{ca}(\delta, s) \wedge \epsilon_F^+(\vec{x}, s) \rightarrow F(\vec{x}, do_{ca}(\vec{x}, \delta, s)) \quad (17)$$

$$Poss_{ca}(\delta, s) \wedge \epsilon_F^-(\vec{x}, s) \rightarrow \neg F(\vec{x}, do_{ca}(\vec{x}, \delta, s)) \quad (18)$$

Proposition 3 (Effect Axioms Entailment)

$$\mathcal{D} \cup \mathcal{D}_{caap} \cup \mathcal{D}_{caSS} \models \mathcal{D}_{caef}$$

I.e., the positive and negative complex action effect axioms are entailed by the pseudo-successor state axioms. Hence, we can easily extract effect axioms for complex actions from our pseudo-successor state axioms.

In this section we have provided a representation of the preconditions, successor state axioms and effects of complex actions under a frame assumption. They are characterized in terms of $\mathcal{D}_{caap}$, and $\mathcal{D}_{caSS}$, and follow from the semantically-justified account of actions in the situation calculus. In the section to follow, we show how these representations of complex actions lead to a simple approach to planning with complex actions.

4 Complex Actions Planning

Given our operator-based characterization of complex actions in terms of their preconditions and effects, we turn to the problem of operator-based planning with these complex actions. For now, we restrict our consideration to the subset of complex actions that are deterministic, I.e., primitive actions a, sequences $\delta_1; \delta_2$, conditional **if** ϕ **then** δ_1 **else** δ_2 **endIf**, and **while** $_k(\phi)$ δ, plus others described in a longer paper.

Following the problem statement in Section 3, our approach is to take as input [$\mathcal{T}_\mathcal{A}$,$\Delta_\mathcal{A}$] – an action theory $\mathcal{T}_\mathcal{A}$ and a set of complex actions $\Delta_\mathcal{A}$, both defined in terms of actions in $\mathcal{A}$. Following the results in the previous section, we **COMPILE** [$\mathcal{T}_\mathcal{A}$,$\Delta_\mathcal{A}$] into a new theory $\mathcal{T}_{\mathcal{A}'}$, in terms of actions $\mathcal{A}'$ (generally $\mathcal{A} \subseteq \mathcal{A}'$), where each complex action in $\Delta_\mathcal{A}$ corresponds to a new primitive action in $\mathcal{A}'$. Next, **PLAN**ning is performed in $\mathcal{T}_{\mathcal{A}'}$ to produce a plan in terms of $\mathcal{A}'$. To extract a plan in terms of the primitive actions, we **REWRITE** the theory, replacing primitive actions from $\mathcal{A}'$ by their corresponding complex actions, $\Delta_\mathcal{A}$. Finally, using $\mathcal{T}_\mathcal{A}$, the resulting sequence of primitive actions is **EXPAND**ed from the plan in $\mathcal{A}'$ into a plan in terms of $\mathcal{A}$.

Next, we show how this approach is realized, first using the situation calculus and deductive plan synthesis, and then using an arbitrary operator-based planning system that allows conditional effects of actions in PDDL.

4.1 Deductive Plan Synthesis and Expansion

The following is the theory with primitive actions $\mathcal{T}_\mathcal{A}$.

$\mathcal{T}_\mathcal{A} = \Sigma \cup \mathcal{D}_{ap} \cup \mathcal{D}_{SS} \cup \mathcal{D}_{una} \cup \mathcal{D}_{S_0}$.

(1) COMPILE[$\mathcal{T}_\mathcal{A}$,$\Delta_\mathcal{A}$] $\rightarrow$ $\mathcal{T}_{\mathcal{A}'}$:

- Define $\mathcal{D}_{caap}$ and $\mathcal{D}_{caSS}$ as described in Section 3.2.
- $\mathcal{D}'_{ap} \leftarrow \mathcal{D}_{caap} \cup \mathcal{D}_{ap}$. $\mathcal{D}'_{SS} \leftarrow \mathcal{D}_{caSS}$. $\mathcal{A}' \leftarrow \mathcal{A}$.
- $\forall \delta_i \in \Delta_A$: Create a primitive action a'_i. Substitute "a'_i" for "δ_i" in $\mathcal{D}'_{ap}$& $\mathcal{D}'_{SS}$. $\mathcal{A}' \leftarrow \mathcal{A}' \cup \{a'_i\}$.
- $\mathcal{D}'_{SS} \leftarrow$ MERGE($\mathcal{D}'_{SS}, \mathcal{D}_{SS}$). Update $\mathcal{D}_{una}$ to $\mathcal{D}'_{una}$.

COMPILE produces a situation calculus theory in actions $\mathcal{A}'$, comprising all the original primitive actions $\mathcal{A}$ plus new primitive actions corresponding to each complex action in $\Delta_\mathcal{A}$. $\mathcal{T}_{\mathcal{A}'} = \Sigma \cup \mathcal{D}'_{ap} \cup \mathcal{D}'_{SS} \cup \mathcal{D}'_{una} \cup \mathcal{D}_{S_0}$

(2) PLAN[$\mathcal{T}_{\mathcal{A}'}$,goal] $\rightarrow$ plan[$\mathcal{A}'$]: Given a goal formula, $Goal(s)$ in the language of $\mathcal{T}_\mathcal{A}$, planning can be achieved via deductive plan synthesis in $\mathcal{T}_{\mathcal{A}'}$. Following [7, 16], $\mathcal{T}_{\mathcal{A}'} \vdash \exists s.Goal(s)$. From the binding of s, we can read off a plan $[a'_1 \ldots, a'_n], a'_i \in \mathcal{A}'$, a plan in $\mathcal{A}'$. [16] describes a variety of situation calculus planners implemented in Prolog.

(3) REWRITE[plan[$\mathcal{A}'$]] $\rightarrow$ plan[$\mathcal{A}$,$\Delta_\mathcal{A}$]: Rewrite the plan $[a'_1 \ldots, a'_n], a'_i \in \mathcal{A}'$ as a plan $[\alpha_1, \ldots, \alpha_n]$ in ($\mathcal{A}$,$\Delta_\mathcal{A}$), where $\alpha_i = a'_i$, for all $a'_i \in \mathcal{A}$, otherwise α_i equals the corresponding δ_i from the compilation in Step (1).

(4) EXPAND[plan[$\mathcal{A}$,$\Delta_\mathcal{A}$],$\mathcal{T}_\mathcal{A}$] $\rightarrow$ plan[$\mathcal{A}$]: Use our same deductive machinery to extract a final plan in $\mathcal{A}$ from our plan in ($\mathcal{A}$,$\Delta_\mathcal{A}$), by expanding the complex actions in $[\alpha_1, \ldots, \alpha_n]$. We do so by trivially rewriting our plan as a

sequence of complex actions in Golog $\delta_G = \alpha_1; \alpha_2; \ldots; \alpha_n$. A Golog interpreter, written in Prolog will return a binding for situation s' where $\mathcal{T}_{\mathcal{A}} \vdash (\exists s').Do(\delta_G, s, s') \wedge Goal(s')$. From the situation s' we can read off a plan $[a_1 \ldots, a_m], a_j \in \mathcal{A}$.

Note that every plan our approach finds is also a plan in the original primitive action theory, and vice-versa.

Theorem 1 *$\mathcal{T}_{\mathcal{A}'}$ and $\mathcal{T}_{\mathcal{A}}$ are defined as in Section 4.1. Let $Goal(s)$ be a formula uniform in s such that $Goal(s) \in \mathcal{L}(\mathcal{T}_{\mathcal{A}}) \cap \mathcal{L}(\mathcal{T}_{\mathcal{A}'})$, the intersection of the languages of $\mathcal{T}_{\mathcal{A}}$ and $\mathcal{T}_{\mathcal{A}'}$ respectively. For all ground situations σ' of $\mathcal{T}_{\mathcal{A}'}$, $\mathcal{T}_{\mathcal{A}'} \models executable(\sigma') \wedge Goal(\sigma')$ iff there exists a ground situation σ of $\mathcal{T}_{\mathcal{A}}$ such that $\mathcal{T}_{\mathcal{A}} \models executable(\sigma) \wedge Goal(\sigma)$ and EXPAND[REWRITE[seq(σ')],$\Delta_{\mathcal{A}}$] = seq(σ), where $seq(do(\vec{a}, s)) = \vec{a}$.*

Proof Sketch: First, by construction of $\mathcal{T}_{\mathcal{A}'}$, $\mathcal{L}(\mathcal{T}_{\mathcal{A}}) \subset \mathcal{L}(\mathcal{T}_{\mathcal{A}'})$, and, for any action a in $\mathcal{T}_{\mathcal{A}}$, $\mathcal{T}_{\mathcal{A}'}$ contains the successor state and action precondition axioms of a in $\mathcal{T}_{\mathcal{A}}$. It follows that, for any term s which denotes a situation in the language of $\mathcal{T}_{\mathcal{A}}$, $\mathcal{T}_{\mathcal{A}} \models Goal(s) \wedge executable(s)$ iff $\mathcal{T}_{\mathcal{A}'} \models Goal(s) \wedge executable(s)$. Second, since in any situation s, the expansion of an executable complex action is also executable and has the same effects, for any executable complex plan $[a'_1, a'_2, ..., a'_n]$ in $\mathcal{T}_{\mathcal{A}'}$, EXPAND[REWRITE$[a'_1, a'_2, ..., a'_n]$,$\Delta_{\mathcal{A}}$]$= [a_1, a_2, ..., a_m]$ is an executable plan in $\mathcal{T}_{\mathcal{A}'}$, and $do([a'_1, a'_2, ..., a'_n], S_0)$ and $do([a_1, a_2, ..., a_m], S_0)$ are the same *states* in $\mathcal{T}_{\mathcal{A}'}$(i.e. fluents has the same truth value in both situations). Finally, by definition of the REWRITE and EXPAND steps, $a_1, a_2, ..., a_m$ are actions in $\mathcal{T}_{\mathcal{A}}$. It follows that $do([a_1, a_2, ..., a_m], S_0)$ is a term in the language of $\mathcal{T}_{\mathcal{A}}$ which denotes a situation, and thus $\mathcal{T}_{\mathcal{A}} \models Goal(do([a_1 ... a_m], S_0)) \wedge executable(do([a_1 ... a_m], S_0))$ if and only if $\mathcal{T}_{\mathcal{A}'} \models Goal(do([a'_1 ... a'_n], S_0)) \wedge executable(do([a'_1 ... a'_n], S_0))$.

Planning in $\mathcal{T}_{\mathcal{A}'}$ is sound and complete with respect to planning in $\mathcal{T}_{\mathcal{A}}$. Thus our approach to complex action planning via transformation of the theory is well-founded.

4.2 Exploiting Existing Operator-Based Planners

Our approach is not limited to planners realized in the situation calculus. Most popular planners don't use a successor state axioms representation of the effects of actions. E.g., all of the planners that participate in the AIPS Planning competition use PDDL as an initial specification of the action theory. In this section we show how to exploit an arbitrary operator-based planner that accepts PDDL planning domains with conditional effects [14], in order to plan with complex actions.

(1) COMPILE[$\mathcal{T}_{\mathcal{A}}$,$\Delta_{\mathcal{A}}$]: Rather than employing successor state axioms, PDDL describes the effects of actions in terms of (conditional) effects without a solution to the frame problem. Section 3.2 provides a semantic justification for an intuitive algorithm that compiles a PDDL representation of the preconditions and effects of actions in $\mathcal{T}_{\mathcal{A}}$, together with complex actions $\Delta_{\mathcal{A}}$ into a new PDDL representation of preconditions and effects in $\mathcal{T}_{\mathcal{A}'}$, without going through the intermediate stage of creating successor state axioms. (We have such an algorithm, but space precluded its inclusion in this paper.) Intuitively, the effects of a complex action are the effects of each action in the execution of δ, modulo subsequent effects.

(2) PLAN[$\mathcal{T}_{\mathcal{A}'}$,goal]: Given a compiled PDDL representation $\mathcal{T}_{\mathcal{A}'}$, we can generate a plan with any planner that accepts PDDL with conditional effects. (We used FF [9].)

(3) REWRITE & (4) EXPAND: We can use STEP (3)-(4) from Section 4.1. Alternatively, we can write a (fairly straightforward) algorithm to expand the final plan in $\mathcal{A}'$. For maximal efficiency, we would cache the conditions that uniquely determine the expansion of each complex action in a situation.

5 Elaborations on Complex Action Planning

In this section we examine elaborations on complex action planning. In particular, we examine the conditions underwhich adding complex actions to a theory causes other actions to be redundant and thus removable. Removing redundant actions is desirable because it reduces the plan search space. In an extended version of this paper, we discuss concurrency in complex action planning.

5.1 Removing Weaker Actions

When a complex action δ_1 is compiled into a primitive action theory as a new primitive action a_1, another primitive action, a_2 may become redundant in the sense that in any situation s, if a_2 is possible, a_1 is also possible and has exactly the same effects as a_2. More generally, we define the notion that primitive action a_1 is *stronger* than primitive action a_2, $a_1 \succeq a_2$ (and conversely that a_2 is weaker than a_1) as follows:

$$a_1 \succeq a_2 \leftrightarrow [\, Poss(a_2, s) \rightarrow Poss(a_1, s) \wedge SS(do(a_1, s), do(a_2, s))\,] \quad (19)$$

where $SS(s, s')$ is an abbreviation for the first-order formulae that is true iff situations s and s' have the same state. The relation $\succeq$ is a preorder (it is reflexive and transitive). It follows that for any situation calculus theory T and goal formula $G(s)$, $T \models \exists s.G(s)$ iff $T' \models \exists s.G(s)$, where T' is T with all weaker actions removed.

Note that removal of weaker primitive actions may result in removal of the optimal plan. In particular, if $a_1 \succeq a_2$ and a_1

is a compiled complex action that can expand into multiple primitive actions, then by removing a_2, we may lose the optimal plan with respect to the number of primitive actions in our initial domain. Also note that the notion of stronger actions does not capture all the conditions underwhich an action is redundant. In particular, a_2 may be conditionally redundant, or it might be redundant relative to a_1 in some situation, and redundant relative to a_3 in others.

Example: Let a_1 and a_2 be primitive actions in $\mathcal{T}_\mathcal{A}$, let $a_2$achieve the preconditions for a_1, and let $Poss(a_1)$ be the situation suppressed expression [16, pg.112] for $Poss(a_1, s)$. Define complex action δ_3 as **if** $\neg Poss(a_1)$ **then** a_2 **endIf** ; a_1. If we compile a_1, a_2 and δ_3 in $\mathcal{T}_\mathcal{A}$ into primitive actions a_1', a_2' and a_3' in $\mathcal{T}_{\mathcal{A}'}$, following Section 4.1, then it follows that $a_3' \succeq a_1'$.

5.2 Irrelevant Actions with Respect to a Goal

Let $G(s)$ be a goal predicate that is true iff s satisifies the goal formula. If the direct effect of an action a can never make $G(s)$ true, and if a cannot directly achieve the preconditions of any of the actions, then a is irrelevant with respect to goal predicate $G(s)$ and can be removed. Formally, given a primitive action a_1 and goal predicate G, we consider a_1 as G-irrelevant in $\mathcal{T}$ if and only if, for a_2 ranging over all actions in $\mathcal{T}$ except a_1, it follows from T that:

$$executable(do(a_1, s)) \leftrightarrow [\,(G(do(a_1, s)) \rightarrow G(s)) \wedge (Poss(a_2, do(a_1, s)) \rightarrow Poss(a_2, s))\,] \quad (20)$$

If a_1 is G-irrelevant, then a_1 will not be in any optimal successful plan to achieve G, and can be removed from the set of actions when planning to achieve G.

Example (continued): In the previous example, we showed that a_1' could be removed from $\mathcal{T}_{\mathcal{A}'}$. It then follows that, a_2' will never be needed to make a_1' $Poss$-ible. If a_2' can never directly achieve the preconditions for any other actions in the theory, then for all goal predicate $G(s)$ which are not among the effects of a_2', a_2' is G-irrelevant.

6 Web Service Composition

The primary motivation for our work was to be able to compose Web services using operator-based planning techniques. With the results of Sections 3 and 4, we have addressed a fundamental barrier to automated Web service composition. Service providers such as Amazon or United Airlines will describe their Web services (Web-accessible programs) as processes. In our vision of the Semantic Web, this will be done using the DAML+OIL Web service ontology, DAML-S [1], whose process description constructs are similar to Golog. (The relationship between DAML-S and the situation calculus is well-defined and has been used to define the semantics of DAML-S.) To produce black-box or compiled representations of Web services for automated composition, we can exploit the compilation techniques described in this paper. Using them, we compile process-oriented program descriptions of services into black-box component descriptions. Once Web services process descriptions have been compiled, we can use standard operator-based planning techniques to automatically compose Web services.

7 Efficiency of Complex Action Planning

A secondary motivation for our work was to potentially improve the efficiency of planning (e.g., [11, 8]) through our operator-based approach to complex action planning. We restrict our attention to complex action planning with the deterministic actions listed in Sect. 4. Compiling a complex action δ is polynomial in the number of primitive action occurrences in its definition. Note that this step can be performed offline, and is amortized over multiple planning runs. The expansion step is itself linear in the length of the plan, and in the number of branchings in the complex action definition. Of no surprise, plan generation dominates the computational cost. In particular: i) complex action operators tend to have more complex preconditions and effects than primitive actions, and ii) the size of the search space will be changed. However, i) causes only a linear slowdown and thus, the crucial point is ii).

Although the following analysis can be adapted to almost any classical planner, for simplicity of the argument, let's consider a breadth-first search forward planner. Given n ground actions, if the shortest successful plan is of length l, the size of the primitive action domain search space is $O(n^l)$. Adding d ground complex actions yields $n' = n+d$ ground actions in the compiled domain. [8] claims that adding actions that correspond to compositions of other actions will yield a larger search space. We identify conditions under which this is false.

Suppose the use of complex actions results in successful plans of length l/k, $k \geq 1$. One way to ensure this is by requiring complex actions to correspond to non-overlapping subplans in the shortest plan. In this case, the number of states visited to find a plan of length l/k will be $O(n'^{l/k})$ and the difference between the search spaces will be $O(n^l - n'^{l/k})$. If $(n^k - n')$ has a strictly positive lower bound for any problem, the new search space will be exponentially smaller than the old, as problem complexity increases. Informally, complex action planning reduces the planning search space when the complex actions significantly shorten the smallest successful plan relative to the increase they cause in the breadth of the search space.

Finally, in addition to this potential search space reduction,

some complex actions remove conflicts between the goals. This results in less backtrackings and enables the use of very efficient hill-climbing techniques (e.g., [9]).

8 Experimental Results

The techniques of Section 4.2 were implemented using the operator-based breadth-first search forward planner, FF [9]. FF supports conditional effects and uses its "enforced hill-climbing" whenever possible. We tested our approach on the ADL BRIEFCASE domain (BCD) [8]. This domain moves objects between locations using a briefcase. Three experiments were run on multiple instances of the problem, varying numbers of locations (#l) and portables (#p)[9].

The first experiment was simply BCD alone. Note that FF struggles as we increase (#p) and (#l). The next experiments involved the addition of the complex action Move-object. Move-object MO(locInit, locObj, Obj, locFinal) takes as input the location of the briefcase locInit, an object Obj, its location locObj, and a destination locFinal. It moves the briefcase to locObj, puts the object in the briefcase, moves the briefcase to locFinal, and removes Obj. MO is not a subplan of the shortest plan, so we would not necessarily expect it to do well. Further, it does *not* reduce the search space as it does not shorten the successful plan enough to compensate for the number of ground complex actions ($(n^k - n')$ is not positive). Nevertheless, adding the complex action move-object (BCD+MO) turns on FF's hill-climbing techniques, which reduce the number of nodes considered.

Finally, we designed a complex action that does correspond to subplans of the shortest successful plan and thus reduces the search space. The complex action LOC(loc-bc, loc), takes as input the location of the briefcase loc-bc, moves the briefcase to location loc, removes all the objects in the briefcase that should be at loc, and puts all other objects at loc in the briefcase. The goal defines where an object should be. To encode this complex action in PDDL, the action must know the goal at the time it executes. Hence, we added a persisting predicate $ShouldBeAt(Obj, Loc)$ to the domain. This predicate always has the same values as the $At(Obj, Loc)$ predicate in the goal statement. This complex action reduces the search space ($k \simeq \#p/\#l, d \simeq \#l$) and allows the use of hill-climbing techniques. Of no surprise, (BCD+LOC) presented the best results of all three experiment runs.

	#l:5, #p:20	#l:6, #p:30	#l:7, #p:42
BCD	5549 (1.39)	201006 (2261)	? (> 40h)
BCD+MO	859 (11.83)	2345 (201.47)	5195 (2211)
BCD+LOC	75 (.08)	139 (.27)	260 (.85)

Number of nodes (and time of run in seconds).

[8] http://rakaposhi.eas.asu.edu/domain-syntax.html

[9] Experiments run on Sun Sparc v9, 2×750GHz, 4GB of mem.

9 Discussion and Summary

The work in this paper was motivated by the problem of automating Web service composition. In particular, we posed the problem of composing Web services such as UAL's buyAirTicket($\vec{x}$) or CNN's getWeather($\vec{y}$) in order to achieve a user-defined goal. These Web services are describable as simple programs, using typical programming language constructs. We conceived this task as the problem of planning with complex actions, with the restriction that the complex actions *had* to be the primitive building blocks of a plan. Consequently, we posed the problem of how to represent and plan with complex actions, using operator-based planning techniques. To this end, we embarked upon a theoretical analysis of the problem of how to represent complex actions as operators. The situation calculus provided the formal foundation for our work, enabling us to provide a formal definition of the preconditions, successor state axioms, and effects of complex actions under a frame assumption.

With this representational problem addressed we turned to the practical matter of how to plan. We proposed a method of planning that produced sound and complete plans relative to a corresponding primitive action domain. We showed how to use our results to plan via deductive plan synthesis as well as using an arbitrary operator-based planning system that accepts ADL as input.

We are currently incorporating these representation and compilation results into DAML-S [1], an AI-inspired markup language ontology for Web services. We're also incorporating the results into ongoing Web service composition work [13].

Finally, the second motivation for this work was to potentially improve either the efficiency of planning or the quality of the plans generated, by exploiting complex actions that capture some preferred subplans. We have shown how, in some domains, using relevant complex actions will result in a dramatic speedup of the planning process. We discussed the impact of our approach on the planning search space and illustrated predicted speedup with experiments.

Acknowledgements

We would like to thank Srini Narayanan for conversations related to this work. We gratefully acknowledge the financial support of the US Defense Advanced Research Projects Agency DARPA Agent Markup Language (DAML) Program #F30602-00-2-0579-P00001.

References

[1] A. Ankolekar, M. Burstein, J. Hobbs, O. Lassila, D. Martin, S. McIlraith, S. Narayanan, M. Paolucci, T. Payne, K. Sycara, and H. Zeng. Daml-s: Semantic markup for web services. In *Proc. International Semantic Web Working Symposium (SWWS)*, 2001.

[2] M. Baioletti, S. Marcugini, and A. Milani. Encoding planning constraints into partial order planning domains. In *Proc. 6th Conference on Knowledge Representation and Reasoning*, pages 608 – 616, 1998.

[3] G. De Giacomo, Y. Lespérance, and H. Levesque. *ConGolog*, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121(1-2):109–169, 2000.

[4] P. Doherty and J. Kvarnstrom. TALplanner: A temporal logic based forward chaining planner. *Annals of Mathematics and Artificial Intelligence*, 30:119–169, 2001.

[5] K. Erol, J. Hendler, and D. Nau. HTN planning: Complexity and expressivity. In *Proc. of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, volume 2, pages 1123–1128, 1994.

[6] R.E. Fikes, P.E Hart, and N.J Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3:251–288, 1972.

[7] C. C. Green. Theorem proving by resolution as a basis for question-answering systems. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 183–205. American Elsevier, New York, 1969.

[8] Patrik Haslum and Peter Jonsson. Planning with reduced operator sets. In *Artificial Intelligence Planning Systems*, pages 150–158, 2000.

[9] Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

[10] S. Hölldobler and H.-P. Störr. BDD-based reasoning in the fluent calculus – first results. In *8th. Intl. Workshop on Non-Monotonic Reasoning (NMR'2000)*, 2000.

[11] R. E. Korf. Planning as search: A quantitative approach. *Artificial Intelligence*, 33(1):65–88, 1987.

[12] H. Levesque, R. Reiter, Y. Lesperance, F. Lin, and R. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31(1-3):59–84, April-June 1997.

[13] S. McIlraith, T. Son, and H. Zeng. Semantic Web services. In *IEEE Intelligent Systems (Special Issue on the Semantic Web)*, March/April 2001.

[14] E. Pednault. ADL: Exploring the middle ground between STRIPS and the situation calculus. In *Proc. KR'89*, pages 324–332, 1989.

[15] R. Reiter. The frame problem in the situation calculus: A soundness and completeness result, with an application to database updates. In *Proceedings First International Conference on AI Planning Systems*, College Park, Maryland, June 1992.

[16] R. Reiter. *KNOWLEDGE IN ACTION: Logical Foundations for Specifying and Implementing Dynamical Systems*. The MIT Press, 2001.

[17] E. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5:115–135, 1974.

[18] T. C. Son, C. Baral, and S. McIlraith. Planning with different forms of domain-dependent control knowledge – an answer set programming approach. In *6th International Conference on Logic Programming and Nonmonotonic Reasoning*, Programming Approach Proceedings, pages 226–239, 2001.

[19] R. J. Waldinger. Achieving several goals simultaneously. In E. W. Elcock and D. Michie, editors, *Machine Intelligence 8: Machine Representations of Knowledge*, pages 94–136. Ellis Horwood, Chichester, 1977.