Action Invariants and System Constraints in STRIPS

Norman Foo
Computer Science and Engineering
University of New South Wales
NSW 2052, Australia
e-mail: norman@cse.unsw.edu.au

Pavlos Peppas
Business Administration
University of Patras
26110 Patras, Greece
e-mail: ppeppas@otenet.gr

Yan Zhang
Computing and Information Technology
University of Western Sydney,
NSW 2747, Australia
e-mail: yan@cit.uws.edu.au

Abstract

In [Foo, et.al. 04] we re-visited the problem of converting action specifications into system constraints by re-examining it in the very simple setting of STRIPS, motivated by the simplicity of its simplicity as an action specification language. We showed how to extract action invariants, and then system laws, from STRIPS specifications. This was the consistency part of our method, and we deferred its adequacy to a later report. This paper provides the missing part, thus closing the circle. Here we show that the invariants extracted by our method can be tested by using them to complete partial specifications that may match those which were used as inputs to the extraction phase. Even when the match is incomplete there is interesting ontological information that can be inferred from it. Fixed point theory is used to account for some of the proposed algorithms.

Keywords: action specifications, STRIPS, constraints, invariants, fixed points, ontology.

1 Introduction

STRIPS [Fikes and Nilsson 71] is possibly the most widely used method for specifying actions in dynamic domains. It precedes much more sophisticated languages like the situation calculus [Shanahan 97], but unlike them its operational features need no logic. However, it is also amenable to a simple logical interpretation. Moreover it is easily and efficiently programmable in Prolog, or procedural languages like Java. For a large number of applications in areas such as planning, device specifications, and network protocols that do not require elaborate (as distinct from large) theories, the simplicity of STRIPS outweighs the known limitations [Lifschitz 86] of its expressiveness that is rectified in, say, the situation calculus. Deterministic actions are naturally represented in it, and rapid prototyping of action specifications for testing and revising is facilitated by its simplicity. AI domains that are *reactive* are particularly suited for STRIPS-like action specifications because they can be described as domains in which the action plans are very shallow and action ramifications [Shanahan 97] are completely known. In such domains the typical action can be paraphrased as "if the system is in a state in which condition A holds, do action B so that it gets to a state in which C holds". An popular example of a highly reactive domain is the robot soccer competition.

In this paper, we focus on the kinds of simple domains that occur widely in practice and re-visit a topic that we had treated in much greater generality in the past [Foo, et.al. 97] [Zhang and Foo 96], and which has been addressed again very recently [Lin 04]. There are several over-lapping motives for this re-visit, but all stemming from the simplicity attribute of STRIPS. First, its restricted expressiveness for simple domains enables us to highlight ontological issues without distraction by technicalities. Ontology here is not about the usual taxonomies of objects, etc., but about choices in the modelling of "real" systems, and involve issues such as the correctness of specifications relative to these choices, the adequacy of the language chosen, and the extraction of implicit systems information from action specifications. Second, the concepts of system laws, and the consistency and adequacy of the specifications are easily grasped in this framework. Finally, there is no need for the relatively heavy logical and model-theoretic machinery that were used in [Zhang and Foo 96] and Lin [Lin 04] but were necessary there because they allowed more expressive languages. In particular, both of these approaches invoked powerful reasoning about minimal changes and action attribution that were expressible only in these languages.

To assist intuition, we use the well-worn blocks world domain as a running example; other examples are adduced to discuss ontology.

We begin with a brief and hopefully self-contained review of STRIPS. Then action constraints, action invariants and system constraints (laws) are defined. A case is then made for an underlying declarative semantics of operationally understood STRIPS action specifications. Using this argument we propose a method for generating candidate action constraints and then invariants from the specifications. The process is then reversed, using the invariants so derived to complete partial action specifications. If the cycle does not close we suggest how that apparent deficiency can be exploited. Finally we stand back and look at how individual action invariants can be combined to yield overall system constraints. Figure 5 summarizes the structure of the paper.

Portions of this work appeared in preliminary form [Foo,

et.al. 04], but there we did not close the cycle.

2 Review of STRIPS

The language STRIPS [Fikes and Nilsson 71] was invented to specify *actions operationally* in the context of discrete-time *state change*. Lifschitz [Lifschitz 86] proposed a declarative semantics of STRIPS, and indeed the semantic descriptions below are similar to his. The states are represented as a (usually finite) set of logical *ground atoms*, effectively a Herbrand model. An action has the effect of changing this set, by deleting some atoms and adding new ones. Thus the *post-conditions* of actions consists of specifying a *deletelist* and an *addlist* of atoms. Actions can be parametrized by variables that are instantiated or bound to particular constants for any one action instance. Moreover, before an action can be executed, it has to satisfy a *precondition* which is often merely a test to see if certain ground atoms are in the state. While STRIPS can be understood procedurally, it is quite natural to interpret the presence of an atom A in a state S as *A is true in S*, or *A holds in S*. In this way a propositional semantics can be given to STRIPS.

As an example, consider a blocks world where we adopt the convention that upper case letters stand for variables and lower case for constants which name particular blocks. The number of blocks is finite. The only predicates[1] are $on(_,_)$, $table(_)$ and $clear(_)$, where, as usual, $on(b, c)$ means the b is on block c, $table(b)$ means the block b is on the table, and $clear(b)$ means the block b has nothing on it.

A state of the world is (a Herbrand model) given by a collection of such ground atoms. For instance, let state $S1$ be:

$on(b, c)$, $on(c, e)$, $table(e)$, $clear(b)$, $on(d, f)$, $table(f)$, $clear(d)$.

We use the standard notation that $S \models \alpha$ to signify that the propositional formula α holds (is true) in a state S; if α happens to be an atom, equivalently in the setting of STRIPS, this simply means that $\alpha \in S$. Thus, for this example $S1 \models on(b, c)$ and $S1 \models table(e) \vee clear(e)$. It is conventional to assume unique names (UNA), and also to invoke the closed world assumption (CWA) which is the formal analog of Prolog negation-as-failure. The UNA says that all constants are distinct, and the CWA (in this simple encoding of states) says that the negation of a ground atom holds in a state if that atom is missing from it. For instance, from the UNA $S1 \models b \neq c$, and from the CWA $S1 \models \neg table(b)$ and $S1 \models \neg on(b, e)$. In the presence of the CWA, all states are effectively *complete*, i.e., for each *literal* α, either $S \models \alpha$ or $S \not\models \alpha$.

Here are some standard blocks world actions, to which we assign names for ease of continuing reference.

$unstack(X)$
$precondition : on(X, Y), clear(X)$
$addlist : table(X), clear(Y)$
$deletelist : on(X, Y)$

$move(X, Z)$
$precondition : on(X, Y), clear(X), clear(Z)$
$addlist : on(X, Z), clear(Y)$
$deletelist : on(X, Y), clear(Z)$

$stack(X, Y)$
$precondition : table(X), clear(X), clear(Y)$
$addlist : on(X, Y)$
$deletelist : table(X), clear(Y)$

In these specifications, multiple predicate occurrences in the components signify conjunctions, e.g., the precondition for $unstack(X)$ is that both $on(X, Y)$ and $clear(X)$ must be present, and its addlist says that the atoms $table(X)$ and $clear(Y)$ are to be added as effects. The presence of *variables* like X, Y and Z allows the actions to be parametrized in much the same way that procedures in imperative and object-oriented languages use variables that have to be instantiated at the time of call. So, while the variables in these specifications suggest a first-order rather than a propositional logic, the device of *unification* can be used to relate the two in the usual manner. The way that it can be done for STRIPS is *operational* — concrete actions are interpreted as follows: given a state S (of ground atoms), we can execute an action ϕ if (in its specification) there is some *unifier* σ of its precondition atoms with S, in which case delete from S those (unified) atoms in iϕ's deletelist, and add to S those atoms in ϕ's addlist. This is equivalent to applying a Prolog meta-rule (the || signifies parallel execution) of the form

$assert(addlist) \mid\mid retract(deletelist) \leftarrow precondition$

if the atoms of S appear as facts in the program, and $addlist$), $deletelist$ and $precondition$ are lists atoms in those components of the action ϕ.[2] If $precondition$ fails to unify, the rule (action) cannot be executed. For example, in the state $S1$ above, one unifier yields the action $unstack(d)$ with deletelist $on(d, f)$, and addlist $table(d)$, $clear(f)$. The resulting state $S2$ is $on(b, c)$, $on(c, e)$, $table(e)$, $clear(b)$, $table(f)$, $clear(d)$, $table(d)$, $clear(f)$.

3 PDL

It is convenient to adopt the language and semantics of *PDL — propositional dynamic logic* [Goldblatt 87], which was introduced to reason about imperative programs. We will only need the simple features of PDL, extending it just enough for our purpose here. While [Goldblatt 87] should be consulted for the details of PDL it suffices for us to briefly outline its features. The syntax and semantics of a PDL sentence are defined exactly as in standard modal logic, except that there can be as many modal operators as there are "actions", where in the context of programs an action is a (sequence of) elementary program statements whose execution can change variable bindings. For example consider the sentence $\delta \to [A]\gamma$. The semantic relation $S \models \delta \to [A]\gamma$ is is defined in the usual Kripke manner by breaking it down to mean $S \not\models \delta$ or $S \models [A]\gamma$. So if $S \models \delta$ and $S \models \delta \to [A]\gamma$, then $S \models [A]\gamma$. As in Kripke semantics $S \models [A]\gamma$ holds if for every state S' such that S can access S' via the relation corresponding to action A, $S' \models \gamma$. It is convenient to use

[1] For simplicity we omit the *in-hand(_)* predicate

[2] In fact, this can be the basis of a simple Prolog program that realizes STRIPS succinctly.

the notation $S[A]S\prime$ to abbreviate this state transition. Thus if $S \models \delta$ then $S \models \delta \rightarrow [A]\gamma$ ensures that γ will hold in all states S' that result from executing the action A in state S, or in abbreviated form for all states S' such that $S[A]S\prime$. The advantage of PDL for us is its brevity, which we rely on to make our definitions of constraints, invariants and laws precise.

4 States, Constraints and Invariants

An *action constraint* is a formula that holds in all states that result from the action. Thus any state that arises from the action will satisfy the action constraint. In the PDL notation:

Definition 1 *The formula β is a constraint of the action ϕ if $S \models [\phi]\beta$ for any state S.*

In the framework of STRIPS with variables in specifications there is a slight technical difficulty with this definition. Consider the the precondition for the $unstack(X)$ action, viz., $\{on(X,Y), clear(X)\}$. In any concrete successful application to a state S of this action, say $unstack(b)$ for a specific block b, the precondition will be unified with atoms $on(b,c)$ and $clear(b)$ in S. At the conclusion of the action, $on(b,c)$ is deleted and $table(b)$ is added to S. In this case all we can infer is the *specific* constraint $\neg(on(b,c) \wedge table(b))$, i.e., $[unstack(b)]S \models \neg(on(b,c) \wedge table(b))$, with no reference to other blocks in S. So, in what sense can the constraint with variables, $\neg(on(X,Y) \wedge table(X))$ be applicable to constants other than b and c? One way to generalize the specific constraint to the general one is as follows. Let $target(S, table(_)) = \{table(d) \mid \exists S' \ such \ that \ S'[unstack(d)]S\}$. Hence, $target(S, table(_))$ is the set of atoms $table(d)$ for some constant d which could have been the result (target) of applying action $unstack(d)$ to some predecessor state S'. It is not hard to see that $S \models \neg(on(X,Y) \wedge table(X))$ for every unification of $table(X)$ with atoms in $target(S, table(_))$. It is this property that suggests $\neg(on(X,Y) \wedge table(X))$ might be also an invariant of the general action $unstack(X)$ in the following sense: if $\neg(on(X,Y) \wedge table(X))$ holds in S and $unstack(X)$ is performed, then $\neg(on(X,Y) \wedge table(X))$ remains true in the resulting state. Formally

Definition 2 *The formula α is an invariant for the action ϕ if $S \models \alpha \rightarrow [\phi]\alpha$, for every state S.*

Corollary 1 *Every action constraint is an action invariant.*

This corollary follows from the definitions of action constraint and invariant since an action constraint is necessarily satisfied in any resulting state. It is exploited in section 5 to suggest candidates for invariants by producing a set of constraints for each action using insights related to the definition of the $target$ predicate.

Our definition of an action invariant should strike a chord with readers familiar with imperative program development or proving. Reasoning about segments of programs (loops being the most common) often involve the discovery of invariants that are the core of the operational meaning of the segments. Trivial invariants abound — any tautology will do — so the interesting invariants are those that are "tight" or stringent, and the latter are often arrived at by examining the preconditions and postconditions of the segment. So it is with actions in STRIPS where any tautology is a system constraint, an action constraint and an action invariant. Indeed so is any contradiction. Hence, are there any guiding principles we can adopt when searching for invariants?

4.1 Which Invariants?

One way to motivate the heuristics that we propose later is to examine how an action ϕ acts on sets of states.

Let us define a function on sets induced by an action ϕ, with precondition denoted by $pre(\phi)$, as follows. Given a set Γ of states, let $\phi(\Gamma) = \{S' | S \models pre(\phi) \ and \ S[\phi]S'\}$. Here we have overloaded the symbol ϕ by using it for both the name of the action and the function it induces, but the context is clear. In order to match the standard statement of a well-known result that will shed light on the selection of invariants, we identify reverse set containment with a lattice order $\leq$, i.e., $U_1 \subseteq U_2$ iff $U_2 \leq U_1$. Under $\leq$ power set of Ω (the set of all the states of a system) form a complete lattice with bottom element Ω and top element the empty set $\emptyset$. We now observe that that the function $\phi : 2^\Omega \rightarrow 2^\Omega$ is monotonic in $\leq$. The Knaster-Tarski theorem (see any book on lattice theory, but the original paper is [Tarski 55]) then says that ϕ has a least fixed point U_0, and furthermore it can be computed as the limit of repeated applications of ϕ starting from the bottom element Ω. Figure 1 indicates one such step in the iteration. The fixed point U_0 is reached when the range of ϕ coincides with its domain, i.e., starting from states in the set U_0 we end up with states that are exactly those in U_0, no more and no less. Ideally, if there is a formula α that expresses U_0, i.e., $U_0 = \{S | S \models \alpha\}$ then α is the invariant we seek for action ϕ. Once α is found, there are many weakenings of α that are also invariants, the weakest being any tautology (it defines Ω). In fact, any set of the form $\phi^k(\Omega)$, the k-th iterate in the repeated application of ϕ leading to the least fixed point U_0, that is definable by some δ yields an invariant that is a weakening of α. We may distinguish "loose" invariants such as these iterates from the fixed points which may be informally called "tight" invariants.

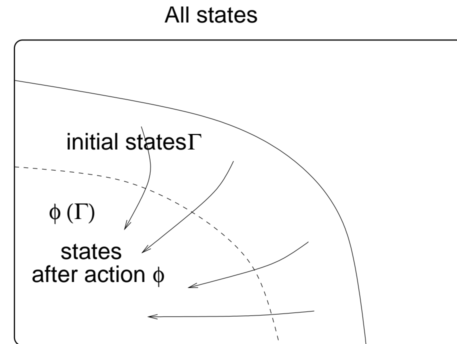

Figure 1: A stage in the application of function ϕ

The upshot of the above remarks can be made more concrete by the following observation.

Observation 1 *If both α and β are invariants of ϕ and $\models \alpha \rightarrow \beta$, then α should be preferred to β.*

The reason for this is that the set defined by α is contained in that defined by β, so one may hope that the former set is "closer" to the fixed point of ϕ.

Hence, in looking for interesting invariants we are really looking for *implicates*, i.e., the logically strongest formulas in the lattice of invariants.

Some action invariants may be general enough to be *system constraints* or *laws*. These are formally defined and discussed in detail in section 7; here the idea can be summarized as follows. Suppose there are a finite number of actions $\phi_1, \ldots, \phi_k$ with respective action invariants $\lambda_1, \ldots, \lambda_k$. Then any λ_i is also a system constraint if it is an action invariant for all ϕ_i for $1 \leq i \leq k$. Action constraints are therefore candidates for action invariants, so ultimately it is action constraints that will be the candidates for system constraints.

Figure 2 summarizes the relationships among action constraints, action invariants and system invariants.

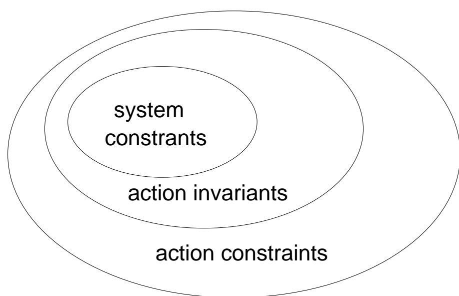

Figure 2: Relationships between constraints and invariants

5 Possible Constraints

The subsections below examine the components in STRIPS specifications and show that they contain implicit information that can be used to extract action constraints as candidates for action invariants, and also suggest ontological alternatives. In these subsections, by constraint we will mean action constraint.

5.1 Reasoning about Addlists and Deletelists

The presumed intention of the addlist and the deletelist as effects of an action is for *correct state update*.

This can be seen in, e.g., the $unstack(d)$ instantiated action on state $S1$ above. Suppose, contrary to intention, we use its addlist components $table(d)$, $clear(f)$ to add to $S1$ but neglect to remove its deletelist component $on(d, f)$. The resulting "state" $S3$ will have a subset of atoms $table(d)$, $clear(f)$, $on(d, f)$. This is intuitively wrong as now block d is both on the table and on block f; moreover it says that block f is clear, so adding to the confusion. A little reflection will show that for any action, adding atoms of an addlist but not removing atoms of a deletelist is the same as permitting both the atoms of the addlist and deletelist to co-exist in the resulting "state", which is an incorrect update. In fact, incorrect updates can also result from co-existence of *parts* of an addlist and deletelist.

We may reason similarly with the other action specifications. This leads to a postulate that captures these intuitions.

Postulate 1 (Addlist-Deletelist Consistency Postulate) *If state S is updated to state S' via a STRIPS action with addlist atom set $Add = \{a_1, \ldots, a_n\}$ and deletelist atom set $Del = \{d_1, \ldots, d_m\}$ then $\neg(a'_1 \wedge \ldots \wedge a'_j \wedge d'_1 \wedge \ldots \wedge d'_k)$, where $\{a'_1, \ldots, a'_j\} \subseteq Add$ and $\{d'_1, \ldots, d'_k\} \subseteq Del$, are candidates for action constraints.*

We can generalize this by lifting the constants to variables in the obvious manner. For brevity, call the generalized version the **Add-Del Postulate**.

Besides this intention there is an implicit assumption that a specification is *parsimonious* — it does not try to delete an atom that is not in the state to be updated, nor does it try to add an atom that is already present. Moreover, any atom in the deletelist is part of the precondition.

For the opportunity of clarifying the above, we thank a reviewer for raising a exemplary objection to the above assumption using an example in which the precondition is simply the atom $true$, and the addlist is $\{a, b\}$. Since this action can execute in any state including the one which already has a in it, one of the assumptions above is violated. This difficulty really resides in the limited expressiveness of STRIPS in its inability to use classical negation in its precondition, addlist and deletelist. A way to circumvent it in this example is to introduce atoms *neg-a* and *neg-b* to mimic the absence of a and b respectively, and to use these to drive the addlist and deletelist accordingly, e.g., with a in the addlist and *neg-a* in the deletelist only when *neg-a* is in the precondition. In the familiar blocks world the atom $clear(a)$ was similarly used to encode a host of classically negative literals like $\neg on(b, a)$, $\neg on(c, a)$, etc. If one did not worry about connections between STRIPS and logic, the example here would be perfectly reasonable in the construction of *mulitlists* in which multiple copies of a and b can exist in the states. Statements like $S \models b$ are then no longer meaningful as they cannot distinguish between one or more copies of b in S. Admittedly considerations like these place a burden of care on the designers of STRIPS specifications that can be cumbersome.

For reference we record the above assumptions as:

Postulate 2 (Parsimony Postulate) *If an action on state S has atoms $a_1, \ldots, a_n$ in its addlist and atoms $d_1, \ldots, d_m$ in its deletelist, then the atoms $d_1, \ldots, d_m$ are all in S but none of $a_1, \ldots, a_n$ are in it. Moreover, $\{d_1, \ldots, d_m\} \subseteq precondition$. Conversely, in the updated state S' all of the atoms $a_1, \ldots, a_n$ are in it, but none of $d_1, \ldots, d_m$.*

Logically, the closed world assumption equates the absence of an atom to its negation. Hence any of the possible constraints proposed by the Add-Del postulate is satisfied by both S and S'. But the action is only possible if S also satisfies its precondition. Thus we have the following:

Proposition 1 *The possible constraints from the Add-Del Postulate are action invariants.*

We observe that this proposition is actually a concrete realization of corollary 1.

In the examples below, distinct variables should (as is the convention with STRIPS specifications) be interpreted as unifying with distinct constant names, denoting distinct objects.

Applying the above to the $unstack(X)$ action above yields the following formulas.

$\neg(on(X,Y) \wedge table(X) \wedge clear(Y))$
$\neg(on(X,Y) \wedge table(X))$
$\neg(on(X,Y) \wedge clear(Y))$

The second and third formulas imply the first. The interesting question suggested by the example is this: which (subset) among the three formulas are the *correct* or truly intended constraints? Of course in a simple and familiar setting such as the blocks world we can quickly make a judgement — the second and third formulas suffice, and are the *essential* correct ones. The first formula is therefore redundant. This is actually a concrete realization of observation 1 above which suggested preferring stronger invariants.

In section 6 we describe a method that test these judgements (assuming that the specification correctly captures the modelling intention).

The potential for combinatorial explosion is revealed by considering what the Add-Del postulate suggests for the $move(X,Z)$ action. It gives the following possible candidates for action constraints:

$\neg(on(X,Y) \wedge on(X,Z))$
$\neg(on(X,Y) \wedge clear(Y))$
$\neg(clear(Z) \wedge on(X,Z))$
$\neg(clear(Z) \wedge clear(Y))$
$\neg(on(X,Y) \wedge clear(Z) \wedge on(X,Z))$
$\neg(on(X,Y) \wedge clear(Z) \wedge clear(Y))$
$\neg(on(X,Z) \wedge clear(Y) \wedge on(X,Y))$
$\neg(on(X,Z) \wedge clear(Y) \wedge clear(Z))$
$\neg(on(X,Z) \wedge clear(Y) \wedge clear(Z) \wedge on(X,Y))$

Which subset among these are the essential correct ones, and which are redundant? One way to attempt an answer is to notice that the shorter ones (the first four) imply the longer ones (the last five), so if any of the the shorter ones can be established to be correct, some of the longer ones will be redundant by observation 1. On the other hand, for any of the longer ones to be essential, it must be the case that the shorter ones that imply it are incorrect. Because of the familiarity of this domain, we can again easily judge which ones are essential. The first formula $\neg(on(X,Y) \wedge on(X,Z))$ is about the *uniqueness of block locations*. The next two *define the meaning* of $clear(X)$ as nothing is on block X; given that these formulas are satisfied in a state S only when suitable bindings exist, they translate to the equivalent constraint $clear(X) \leftrightarrow \neg\exists Y\, on(Y,X)$. The fourth formula does not convey any useful information, but (like the rest) is nevertheless an action invariant since $clear(Z)$ is true in S but false in S', and $clear(Y)$ is false in S but true in S'. Due to subsumption, we may ignore the remainder. As with the $unstack$ action, The method in section 6 may be used to provide suggestions of correct invariants should intuitive ones not be available.

5.2 Ontology

A merit of the simplicity of STRIPS as an action specification language is that the simple attendant logic lends itself to revisions and extensions of ontological assumptions about which the designer may not have been consciously aware. The two subsections below explain by example some circumstances in which these may arise.

5.3 Wrong Invariants

In less familiar domains the kind of judgement that we exercised in the blocks world to select the correct action constraints from the possible ones suggested by the Add-Del postulate may not be so immediate. The next example illustrates this. Consider a domain in which there are two switches and in an electric circuit in which there is also a light. The STRIPS specification of this system uses three propositions — $sw1$, $sw2$ for saying that the respective switches are turned on, and $lightoff$ for saying that the light is off. Here is an attempted specification of an action for turning on the light.

$TurnOn$
$precondition : lightoff$
$deletlist : lightoff$
$addlist : sw1, sw2$

The Add-Del postulate suggests these as possible action constraints:

$\neg(sw1 \wedge lightoff)$
$\neg(sw2 \wedge lightoff)$
$\neg(sw1 \wedge sw2 \wedge lightoff)$

The first formula is incorrect if there is a state SS such that $SS \models sw1 \wedge lightoff$. Hence we should look for a system in which this is so. A system in which there is such a state is shown in figure 3. As this system can also invalidate the second formula, this leaves only the longer third formula as the correct constraint. An alternative is a system in which the first and second formulas are indeed constraints, and the third is therefore redundant. A system in which this is the case is shown in figure 4. This example shows how questions about which among the possible formulas suggested by the Add-Del postulate are actual action constraints can trigger off a search for alternative *ontologies*. This search is *extra-logical*, for there is nothing in the implicit logic of STRIPS that can inform the ultimate choice. However, it is interesting that it can nevertheless *suggest* what to look for. For many domains the *modelling* enterprise is tentative and iterative, for we may be guessing at the internal structure of a black box. The use of the preceding method is to decide which experiments to run — or what questions to ask the persons who wrote the action specifications — in a search for possible invalidation of (short) possible constraints so that an ontological judgement can be made.

5.4 Action Relaxation

In the *move(X,Z)* action whose invariants were considered in section 5 we observe that two of the strongest ones, viz. $\neg(on(X,Y) \wedge on(X,Z))$ and $clear(X) \leftrightarrow \neg\exists Y\, on(Y,X)$ are *independent* in the usual sense in logic. The actual invariant of the *move(X,Z)* action is of course a *conjunction*

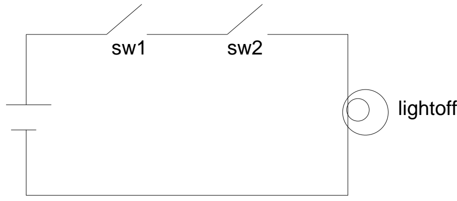

Figure 3: Switches in series

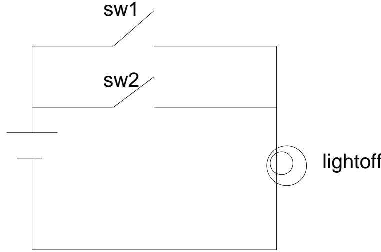

Figure 4: Switches in parallel

of the two. However, whenever an invariant comprises independent conjuncts, it suggests that there are *more relaxed* actions that can be used as the bases of the action, albeit each such action is associated with unintended ontologies. The relaxed action (call it *move-1(X,Z)*) corresponding to the component invariant $\neg(on(X,Y) \wedge on(X,Z))$ is one in which the $clear(X)$ atom is not in the ontology, i.e., with precondition $on(X,Y)$, addlist $on(X,Z)$ and deletelist $on(X,Y)$. Curiously, *move-1(X)* may be performed even when a *tower* of blocks sit on block X, so the effect is to move the entire tower which has block X at its base to the top of block Z. Moreover, it does not worry that block Z may already have other blocks on it. On the other hand the relaxed action (call it *move-2(X,Z)*) corresponding to the component $clear(X) \leftrightarrow \neg\exists Y on(Y,X)$ is one that permits block X to be moved to block Z when there is no block on either of them, i.e., with precondition $on(X,Y)$,$clear(X)$, $clear(Z)$, addlist $on(X,Z)$ and deletelist $on(X,Y), clear(Z)$. It is not worried about the possibility that a block may be in two or more blocks at the same time. These are unintended ontologies in the same sense as non-standard interpretations.

There is a way to think about such relaxed actions relative to the originally specified action from the perspective of the Knaster-Tarski function iterates to the least fixed point. From this perspective the invariant of *move-1* expresses its least fixed point U_1, and that of *move-2* expresses its least fixed point U_2. Both U_1 and U_2 are weaker than U_0, the least fixed point of *move*, hence in the lattice ordering of section 4.1 we have $U_1 \leq U_0$ and $U_2 \leq U_0$. Further, it can be seen informally that the fixed point U_0 is also a fixed point of both *move-1* and *move-2*, but of course not their least fixed point. This observation is consistent, and in fact suggested by, a corollary of the Knaster-Tarski theorem (op.cit.) which says that the fixed points of a monotonic function are themselves are a complete (sub-) lattice. Hence we can describe the "decomposition" of an action into more relaxed components as one which searches for actions that have larger state sets as their "tight" invariants.

5.5 Reasoning about Preconditions

Analogous to our dissection of the intended meaning of addlists and deletelists, we now examine the precondition component of an action specification. For an action $\phi(X)$ consider why its (non-trivial) precondition $\pi(X)$ might be written. The intention appears to be that the action a state S satisfying $\pi(c)$ the action $\phi(c)$ can be safely executed by updating S with the addlist and deletelist accordingly. Importantly, if S does not satisfy $\pi(c)$, then $\phi(c)$ must *not* be executed. This suggests that whenever S does not satisfy $\pi(c)$ but (parts of) the addlist and deletelist are nevertheless used to update S, the resulting state is incorrect. This looks rather formidable except that in fact much of its apparent complexity is already accounted for by the Add-Del postulate. Let the set of atoms in the precondition be Pre. Then the possible constraints can be expressed as:

$\neg(\neg(p_1 \wedge \ldots \wedge p_k) \wedge C)$

where $\{p_1,\ldots,p_k\} \subseteq Pre$ and C is one of the candidate action constraints from the Add-Del postulate.

An example of this is the precondition for the $stack(X,Y)$ action. Assume that the component $clear(Y)$ does not hold. By the constraint above this is equivalent to the existence of some Z (distinct from X) such that $on(Z,Y)$ holds. Then one possible constraint is $\neg(\neg clear(Y) \wedge on(X,Y))$ where the $on(X,Y)$ is from the addlist, and this formula is equivalent to $\neg(on(Z,Y) \wedge on(X,Y))$, another familiar constraint.

Since an action invariant is of the form "if $S \models \alpha$, then after action ϕ $S' \models \alpha$", the possible constraints that arise from considering preconditions are trivially action invariants because the precondition α is false in each of them.

6 Testing the Invariants

The method described in this section is similar to a well-known technique used in databases (see e.g., [Lawley, Topor and Wallace 93]) for doing the reverse of invariant discovery. However we consider the case where the invariants may not suffice to recover all facets of the original specification. The formal rendition of that work in our vocabulary would be this: *Start with invariants; then given the add-list of actions, derive appropriate preconditions and delete-lists.* As is well-known from standard work in programming language semantics, there is a trade-off among these entities but it is the weakest preconditions that are sought (presumably to make the scope of action application as wide as possible). Relative to that work our approach here is therefore a kind of reverse-engineering. It treats the action specifications as primary entities, hypothesizing that they express an intended, underlying and implicit semantics that are action constraints, action invariants and system constraints. A remark made by one reviewer of an earlier version of this paper is helpful: both action and system constraints can be viewed as static, the former being local and the latter being global; on the other hand action invariants are dynamic.

To test the invariants proposed by the methods of section 5 above we use the latter and attempt to recover the original specification. A schematic diagram of the overall method is shown in figure 5. The method is informally explained by

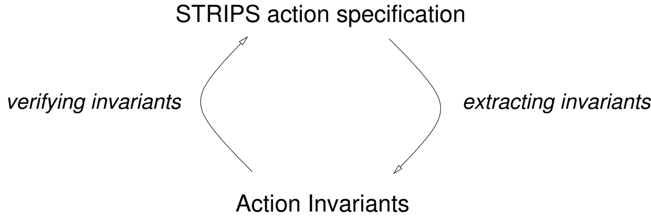

Figure 5: From specifications to invariants and back

an example. Consider the *unstack(X)* action that we saw earlier. Below is a partial version of it with the precondition and deletelist unspecified. The aim is to complete this precondition and deletelist so that they match the original specification from which the invariant was extracted.

unstack(X)

$$precondition :? \quad (1)$$

$$addlist : table(X), clear(Y) \quad (2)$$

$$deletelist :?? \quad (3)$$

The invariants for the *unstack(X)* actions are:

$$\neg(on(X,Y) \wedge table(X)) \quad (4)$$

$$\neg(on(X,Y) \wedge clear(Y)) \quad (5)$$

From the atom $table(X)$ in the addlist we know that after the action, in the new state S' $table(X)$ must hold, i.e., is $true$. This, together with the fact that $\neg(on(X,Y) \wedge table(X))$ is an invariant for the action, leads to the conclusion that in S' the atom $on(X,Y)$ is $false$, i.e., it has to be in the deletelist, which is indeed the case in the original specification for the *unstack(X)* action. The other invariant yields the same conclusion for the deletelist, so it has only one atom, viz, $on(X,Y)$. From the Parsimony Postulate we may also conclude that $on(X,Y)$ is in the precondition. What then about the other atom $clear(X)$ in the original precondition? There is nothing in our method that says anything about it, and for good reason. Consider a slightly different action $unstack'(X)$ defined by:

$$unstack'(X)$$

$$precondition : on(X,Y) \quad (6)$$

$$addlist : table(X), clear(Y) \quad (7)$$

$$deletelist : on(X,Y) \quad (8)$$

It can be intuitively seen that the effect of $unstack'(X)$ is to pick up the *tower* of blocks with X as its base and then place this tower on the table. Thus, the invariants for the $unstack(X)$ action are not strong enough to exclude this *unintended interpretation* of the action. In fact we can exploit this in the following manner. If the method of completing the precondition and deletelist yields a precondition (list) that is smaller than the original specification, then there is an alternative unintended action specification consistent with the invariants.

We can generalize these observations as follows. Any atom δ in state S that is not in the deletelist will not change as a result of the action.

Definition 3 *An atom δ such that $S \models \delta \rightarrow [\phi]\delta$ for all states S is ϕ-persistent.*

In the $unstack(X)$ action the $clear(X)$ atom is persistent.

Corollary 2 *A ϕ-persistent atom cannot be in its deletelist.*

As in sub-section 5.5, let $\pi(\phi)$ denote the precondition of action ϕ. Also, call the invariants obtained via the Add-Del postulate the Add-Del invariants.

Proposition 2 *If δ is ϕ-persistent and $\delta \in \pi(\phi)$ there is an alternative action ϕ' with specification the same as that of ϕ except that $\pi(\phi') = \pi(\phi) \setminus \{\delta\}$, and the Add-Del invariants of ϕ' are the same as that of ϕ.*

Because the alternative action precondition is weaker than that in the original specification it may be regarded as a suggestion to the designer to consider a more general action.

The remaining possibility is that this method may also yield a deletelist that is smaller than the original. This is an indication that the user selection of invariants is incorrect because they admit resulting states that violate the Add-Del or Parsimony postulates. It is a suggestion to re-examine the candidate list of invariants to select stronger ones.

There is another aspect that we are currently investigating, viz., the interaction of invariants from different actions, as in system constraints below, that can be used also as inputs into specification completion. We expect to report on this in the near future.

7 System Constraints

We now examine how action invariants can be elevated to system constraints. In preparation for this we need some concepts that are analogous to well-known ones in dynamical systems theory. By the notation $S\phi S'$ we mean that applying action ϕ to state S yields the state S'. This notation extends naturally to a sequence of actions $\phi^1, \ldots, \phi^n$ where $S_0\phi^1 S_1 \ldots \phi^n S_n$ has the obvious meaning, and we say that S_n is *reachable from* S_0 (via that action sequence). The actions ϕ^i will be from a set $\Phi = \{\phi_1, \ldots, \phi_m\}$ of actions, so to describe arbitrary sequences of actions on states using such actions we may say that the previous sequence $S_0, S_1, \ldots, S_n$ is a Φ-*trajectory*. Thus S' is reachable from S if there is a Φ-trajectory that begins with S and ends with S'. $Reach(S, \Phi)$ is the set of states reachable from S via the set Φ of actions; if Σ is a set of states, $Reach(\Sigma, \Phi)$ is the set $\bigcup_{S\in\Sigma} Reach(S, \Phi)$. Thus, $Reach(_, \Phi)$ may be viewed as a map from sets of states to sets of states, i.e, if Γ is the set of all states, $Reach(_, \Phi) : \Gamma \rightarrow \Gamma$.

Given an action ϕ let $\Sigma(\phi)$ denotes the states that satisfy the action invariants of ϕ, i.e. $\Sigma(\phi) = \{S | S \models \psi \text{ and } \psi \text{ is an invariant of } \phi\}$.

Proposition 3 $\Sigma(\phi)$ *is a fixed point of* $Reach(_, \{\phi\})$

Proof: If $S \in \Sigma(\phi)$ then by definition of $\Sigma(\phi)$, if S' is the result of action ϕ on S, $S' \in \Sigma(\phi)$. Hence, by induction, $\Sigma(\phi)$ is closed under any number of applications of ϕ, and therefore $\Sigma(\phi) \subseteq Reach(\Sigma(\phi), \{\phi\})$. On the other hand, if $S \in Reach(\Sigma(\phi), \{\phi\})$, there is a sequence $S_0, S_1, \ldots, S_n = S$ such that $S_0\phi S_1, S_1\phi S_2, \ldots, S_{n-1}\phi S$

and $S_0 \in \Sigma(\phi)$. By closure of $\Sigma(\phi)$ under repeated application of ϕ, $S \in \Sigma(\phi)$, so $Reach(\Sigma(\phi), \{\phi\}) \subseteq \Sigma(\phi)$.

What is the largest collection of such fixed points across all actions? To answer this question, let us consider two actions ϕ_1 and ϕ_2, and the sets $\Sigma(\phi_1)$ and $\Sigma(\phi_2)$. Also, for brevity we write $\phi(S)$ to mean the state S' that results after applying action ϕ to state S. Recall that if the invariants of ϕ are also invariants for all other actions then these invariants are system constraints. So if ϕ_1 and ϕ_2 were the only actions, a guess at the generalization of proposition 3 might be the following: $\Sigma(\phi_1) \cap \Sigma(\phi_2)$ is a fixed point of $Reach(_, \{\phi_1, \phi_2\})$. There is a slight problem with this. While certainly $S \in \Sigma(\phi_1) \cap \Sigma(\phi_2)$ implies $\phi_1(S) \models \psi_1$ and $\phi_2(S) \models \psi_2$ for invariants ψ_1 of ϕ_1 and ψ_2 of ϕ_2, it may not be the case that $\phi_1(S) \models \psi_2$ or $\phi_2(S) \models \psi_1$. If we want ψ_1 and ψ_2 to be system invariants, what we really need is for each of them to be invariants also for the other action. In effect we need to have $\psi_1 \wedge \psi_2$ be an action invariant for both actions. This motivates the generalization below.

Let $\Sigma(\Phi)$ denote the states that satisfy the action invariants of every ϕ in Φ, i.e. $\Sigma(\Phi) = \{S | S \models \psi,\ \psi\ is\ an\ invariant\ of\ \phi,\ and\ \phi \in \Phi\}$. The following proposition has a proof which is a generalization of that of proposition 3.

Proposition 4 *$\Sigma(\Phi)$ is a fixed point of $Reach(_, \Phi)$.*

As an example, the action constraints in the blocks world domain above are also system constraints.

We conclude with some observations about anomalous components of states in the blocks world that exemplify similar situations in other domains. A *local anomaly* is a part of a state that violates system constraints. In the STRIPS convention of ground atoms representing state, this is simply a collection of atoms (subset of the state) that do not satisfy a system constraint. We can summarize the observations below by saying that local anomalies can be *quarantined*.

Consider a state that has an atom $on(b, b)$. Ontologically this is nonsense, but nothing in the object-level STRIPS excludes it. It formally fails the precondition for all actions (block b is never $clear$!) that either tries to delete or move it, or to stack on it. So, if we begin with a state that has this we are stuck with it forever. But if we start with "normal" states we can never reach one that has such a local anomaly. What some people may find disturbing is this: unless we write a constraint that precludes such atoms, none of the action (and therefore, systems) constraints can exclude states from containing anomalous atoms. However, we may console ourselves with two facts. If we begin with non-anomalous states, then all trajectories will remain non-anomalous. And, if we had such anomalous atoms, in a sense they will be *irrelevant* as they can never participate in any action.

Now consider another kind of local anomaly for which there is provably no first-order constraint that excludes it. This example suffices to highlight the problem: let there be a chain of atoms $on(a_1, a_2), on(a_2, a_3), \ldots, on(a_{k-1}, a_1)$. This is just an elaboration of the pervious one, but to exclude it requires a formula for transitive closure — none exists if the chain length is not known. But the same consoling remarks apply to such chains.

8 Conclusion

A method was proposed to extract implicit constraints from STRIPS action specifications, and another method to test these constraints by attempting to re-generate action specifications from them. Specification design issues were addressed as a bonus of these methods. The roles and connections between action constraints, invariants and system constraints were elucidated.

References

[Fikes and Nilsson 71] Fikes, R. E. and Nilsson, N. J., " STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving", *Artificial Intelligence*, 2, 1971, 189-208.

[Foo, et.al. 97] Foo, N., Nayak, A., Pagnucco, M., Peppas, P., and Zhang, Y., "Action Localness, Genericity and Invariants in STRIPS", Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI'97, pp. 549-554, Nagoya, August 1997, Morgan Kaufmann.

[Foo, et.al. 04] Foo, N., Peppas, P., and Zhang, Y., "Constraints from STRIPS — Preliminary Report", Proceedings of the 17th Australian Joint Conference on Artificial Intelligence, AI'04, Eds. G. Webb and X. Yu, Springer LNAI no. 3339, pp 670-680. , Cairns, December 2004.

[Goldblatt 87] Goldblatt, R., *Logics of Time and Computation*, Lecture Notes 7, CSLI Publications, 1987.

[Lawley, Topor and Wallace 93] Lawley, M., Topor, R. and Wallace, M., "Using Weakest Preconditions to Simplify Integrity Constraint Checking", Proceedings of the Australian Database Conference, 1993.

[Lifschitz 86] Lifschitz, V., "On the Semantics of STRIPS", in *Reasoning about Actions and Plans*, ed. M. Georgeff and A. Lansky, Morgan Kaufmann Publishers, 1986, 1-9.

[Lin 04] Lin, F., "Discovering State Invariants", Proceedings of the Ninth International Conference on Principles of Knowledge Representation and Reasoning, KR'04, 536-544, Whistler, 2004.

[Shanahan 97] Shanahan, M., *Solving the Frame Problem: A Mathematical Investigation of the Common Sense Law of Inertia*, MIT Press, 1997.

[Tarski 55] Tarski, A., "A lattice-theoretical theorem and its applications", Pacific Journal of Mathematics, vol. 5 (1955), pp 285-309.

[Zhang and Foo 96] Zhang, Y. and Foo, N., "Deriving Invariants and Constraints from Action Theories" , Fundamenta Informaticea, vol. 30, 23-41, 1996.