Extending TIM domain analysis to handle ADL constructs

Stephen Cresswell, Maria Fox and **Derek Long**
Department of Computer Science, University of Durham, UK.
{s.n.cresswell,maria.fox,d.p.long}@durham.ac.uk

Abstract

Planning domain analysis provides information which is useful to the domain designer, and which can also be exploited by a planner to reduce search. The TIM domain analysis tool infers types and invariants from an input domain definition and initial state. In this paper we describe extensions to the TIM system to allow efficient processing of domains written in a more expressive language with features of ADL: types, conditional effects, universally quantified effects and negative preconditions.

Introduction

The analysis of a planning domain can reveal its implicit type structure and various kinds of state invariants. This information can be used by domain designer to check the consistency of the domain and reveal bugs in the encoding. It has also been successfully exploited in speeding up planning algorithms by allowing inconsistent states to be eliminated (Fox & Long 2000), by revealing hidden structure that can be solved by a specialised solver (Long & Fox 2000), or as a basis for ordering goals (Porteous, Sebastia, & Hoffmann 2001).

Many other researchers are also interested in domain analysis. Planners based on propositional satisfiability (Kautz & Selman 1998) and CSP (van Beek & Chen 1999) often require hand-coded domain knowledge, much of which could be derived from domain analysis. TLPlan (Bacchus & Kabanza 2000) and TALPlanner (Doherty & Kvarnstrom 1999) rely on control knowledge which might be inferrable from the underlying structures describing the behaviour of the domain. Our analysis produces not just invariants, but the underlying behavioural models from which they can be produced. These models provide a basis for further analysis, giving an advantage over other invariant algorithms, such as DISCOPLAN (Gerevini & Schubert 1998), which do not produce these structures.

The TIM system performs its analysis by constructing from the planning operators a set of finite state machines (FSMs) describing all the transitions possible for single objects in the domain. The type structure and domain invariants are derived from analysis of the FSMs.

Earlier versions of TIM have accepted planning problems expressed only in the basic language of STRIPS. A more expressive language for describing planning domains using features derived from ADL (Pednault 1989) is in widespread use. The use of these extensions makes life easier for the domain designer, but it is difficult to handle them efficiently in planning systems.

In this paper we describe work to extend the language handled by TIM to include the main features of ADL, whilst also attempting to preserve the efficiency of TIM processing. The features we describe here are types, conditional effects, universally quantified effects and negative preconditions.

For example, in the briefcase domain, consider the `move` operator, shown below. The operator includes a universally quantified conditional effect which says that when the briefcase is moved between locations, all the portable objects which are inside the briefcase also change their location. [1]

```
(:action move
  :parameters (?m ?l - location)
  :precondition  (is-at ?m)
  :effect (and (is-at ?l)
               (not (is-at ?m))
               (forall (?x - portable)
                  (when (in ?x)
                     (and (at ?x ?l)
                     (not (at ?x ?m)))))))
```

A typical invariant that we would like to get from this domain is that portable objects are at exactly one location:

$$\forall x : \text{portable} \cdot \forall y \cdot \forall z \cdot at(x, y) \wedge at(x, z) \rightarrow y = z$$
$$\forall x : \text{portable} \cdot (\exists y : \text{location} \cdot at(x, y))$$

TIM

For a full account of the TIM processing, see (Fox & Long 1998). Here we briefly summarise those parts of the algorithm which are relevant to the extension to ADL.

[1] This encoding of the domain is from IPP problem set.

We are concerned with extracting state descriptions for individual objects. These are described using *properties*, which describe a predicate and an argument position in which the object occurs. For example, if we have $at(plane27, durham)$ in the initial state, then we consider the object $plane27$ to have property at_1, and the object $durham$ to have property at_2.

To explain the processing of domain operators by TIM, we must introduce two kinds of derived structure — the *property relating structure* (PRS) and the *transition rule*.

The PRS is derived from an operator, and records the properties forming preconditions, add effects and delete effects for a single parameter of an operator. For example, consider the operator `fly`:

```
(:action fly
  :parameters (?p ?from ?to)
  :precondition (and (at ?p ?from) (fuelled ?p) (loc ?to))
  :effect (and (not (at ?p ?from))
               (at ?p ?to)
               (not (fuelled ?p))
               (unfuelled ?p)))
```

The parameters of the operator give us the following PRSs:

`?p`		`?from`		`?to`	
pre:	$at_1, fuelled_1$	pre:	at_2	pre:	loc_1
add:	$at_1, unfuelled_1$	add:		add:	at_2
del:	$at_1, fuelled_1$	del:	at_2	del:	

A *transition rule* is derived from a PRS, and describes properties exchanged. Transition rules are of the form:

$$Enablers \Rightarrow Start \rightarrow Finish$$

where $Enablers$ is a bag of precondition properties which are not deleted, $Start$ is a bag of precondition properties which are deleted, and $Finish$ is a bag of properties which are added. Empty bags are shown as $null$, or may be omitted in the case of enablers. The transition rules corresponding to the above PRSs are:

$$fuelled_1 \Rightarrow at_1 \rightarrow at_1$$
$$at_1 \Rightarrow fuelled_1 \rightarrow unfuelled_1$$
$$at_2 \rightarrow null$$
$$loc_1 \Rightarrow null \rightarrow at_2$$

The part of the TIM processing that we discuss in the rest of the paper consists of the following main stages:

1. Construct PRSs from operators.
2. Construct transition rules from PRSs.
3. Unite the properties in the *start* and *finish* parts of transition rules, to group properties into property and attribute spaces. Property spaces arise where transition rules define a finite number of possible states (as the rules involving at_1, $fuelled_1$, and $unfuelled_1$) — these define an FSM. Attribute spaces arise where transition rules allow properties to be gained without cost (as for at_2) — these spaces must be treated in a separate way, as it is not possible to enumerate all of their possible states.
4. Project the propositions from the initial state of the planning problem to form bags of properties representing the initial states for each state space.
5. Extend the states in each space by application of transition rules.

This process derives information that can be used to extract types and invariants. Each unique pattern of membership of property and attribute spaces for the domain objects defines a type. From the generated FSMs, domain invariants can be extracted.

ADL extensions to TIM

The original version of TIM (Fox & Long 1998) deals only with untyped STRIPS domains. We are interested in extension to a subset of ADL, and our extensions fall into four areas:

- Types
- Universally quantified effects
- Conditional effects
- Negative preconditions

A generic means of transforming ADL domain descriptions into simple STRIPS representations has been given by Gazen and Knoblock (Gazen & Knoblock 1997). Although it would be possible to apply this directly as a preprocessing stage to TIM, an undesirable feature of this approach is that several stages of the processing may lead to an exponential blow-up in the size of domain descriptions or initial state descriptions.

In particular, universally quantified effects lead to an expansion in the size of the operator proportional to the number of domain objects that can match the quantified variable. Conditional effects lead to generation of multiple operators, with one operator for each possible combination of secondary conditions.

In the following work, we seek to avoid the combinatorial blow-up with a combination of simplifying assumptions, and by avoiding operator expansions which are irrelevant to the TIM processing.

TIM analysis only needs to work with structures describing what transitions are available to individual objects, and what states they can reach. The transitions generated must capture all possible transitions, and generate all possible states of individual objects. The transitions are only partial descriptions of operators, and the object states are only partial descriptions of a planning state. This representation does not preserve or make use of all the conditions present in the original operator, and could not be correctly used in a planner. Hence, some of the information which is painstakingly preserved in the Gazen-Knoblock expansion (e.g. instantiation of `forall` effects) makes no difference to the resulting TIM analysis, and can be avoided.

Now we consider the processing of each language feature in more detail. In most cases, we describe the conversion by showing a transformation on the operator and domain description before the standard TIM analysis is performed on the modified descriptions.

In general, the transformations carried out on operator descriptions will not yield operators that are correct for use in a planner. They are correct for the TIM analysis, because they yield all possible legal transitions for objects in the domain.

Types

Part of the processing performed by TIM is to derive a system of types for the domain. Clearly, if types have already been specified in the domain, these should be taken into account. Type restrictions on the parameters of an operator are transformed into additional static preconditions on the operators.

Types declared for the objects in the domain also give rise to additional propositions in the initial state. Note that since PDDL allows for a hierarchy of types, each object must have a proposition for every type of which it is a member.

The predicates which are automatically introduced to discriminate will be recognised as static conditions by TIM, and will then be taken into account when determining the derived types inferred by TIM. Hence, we can be sure that TIM types will always be at at least as discriminating as the declared types.

Conditional Effects

Treatment of conditional effects is rather more challenging, but offers far more scope for interesting results from the analysis. Consider the following example operator:

```
(:action A
 :parameters (?x)
 :precondition (p ?x)
 :effect (and (not (p ?x)) (q ?x)
              (when (r ?x) (and (not (r ?x)) (s ?x)))))
```

From this operator we would like to infer (assuming no other operators affect the situation) that parameter `?x` can make transitions $p_1 \rightarrow q_1$ and $p_1 \Rightarrow r_1 \rightarrow s_1$. This would enable us to identify two potential state spaces and consequent invariants: p and q are mutex properties and s and r are mutex properties.

In considering treatment of conditional effects, several proposals were examined and these are discussed in the following sub-sections.

Separated dependent effects One technique by which we hoped to harness the power of existing TIM analysis was to construct a collection of standard transition rules that capture the behaviour encoded in conditional effects. The previous example shows that this is possible in certain cases. The first proposal for achieving this was based assuming all `(when ... ...)` clauses in an operator to be mutually exclusive, in which case there is no problem with exponential blow-up. Under this assumption, we can generate a separate pseudo-operator for each conditional effect. The pseudo-operator has the primary preconditions and effects from the original operator, and the `when` clause is absorbed by merging its preconditions into the primary preconditions, and its effects into the primary effects. We also create an operator with only the primary effects. These operators are pseudo-operators because they do not always encode the same behaviour as the original operators and could not be used for *planning*. However, this does not prevent them from being used to generate valid transition structures that reflect the transitions described by the original conditional effects operators.

Unfortunately, conditional effects do not always satisfy the assumption that at most one conditional effect is triggered during a single application of an action. In fact, the assumption required for correct behaviour can be weakened: it is only necessary that at most one conditional effect is triggered *to affect the state of each parameter of the operator*. Even this condition is stronger than is appropriate for some domains.

An example where the assumption is violated is the following:

```
(:action op_with_non_exclusive_conditions
 :parameters (?ob)
 :precondition (a ?ob)
 :effect (and (not (a ?ob)) (x ?ob)
              (when (b ?ob)  (and (not (b ?ob)) (y ?ob)))
              (when (c ?ob)  (and (not (c ?ob)) (z ?ob)))))
```

For an object with initial properties $\{a_1, b_1, c_1\}$, this operator should allow the state $\{x_1, y_1, z_1\}$ to be reachable.

Instead, we actually generate the rules $(a_1 \rightarrow x_1)$, $(a_1, b_1 \rightarrow x_1, y_1)$ and $a_1, c_1 \rightarrow x_1, z_1$, since the operator leads to the creation of three pseudo-operators, one with the pure simple effects of deleting a and asserting x and the others each taking a precondition from their respective conditional effect and the appropriate additional effect. With these rules it is not possible to reach the state $\{x_1, y_1, z_1\}$.

Failure to correctly generate all reachable states leads to the generation of unsound invariants, so this approach cannot be used unless it is possible to guarantee the necessary conditions for valid application.

Separating conditional effects The strong assumption, that at most one of the conditional effects of an operator will apply, can be replaced with a much weaker assumption: that any number of the conditional effects of an operator could be applicable. This assumption can be characterised using the original TIM machinery by creating pseudo-operators, one with the complete collection of primary preconditions and effects from the original operator and one for each conditional effect, adding the condition for the effect to the preconditions of the original and *replacing* the effect of the original with the conditional effect. In the example above, this would lead to the following transition rules: $a_1 \rightarrow x_1$, $b_1 \rightarrow y_1$ and $c_1 \rightarrow z_1$.

With these rules it is possible for extension (the process by which TIM generates complete state spaces from the initial state properties of objects) to generate all the states we want, and more besides. For instance, we could generate a state $\{a_1, y_1, c_1\}$, by applying one of

the conditional effects without the corresponding primary effect.

The weakened assumption leads to correspondingly weaker invariants, since the opportunity to apply rules that do not correspond to actual operator applications allows apparent access to states that are not, in fact, reachable. More seriously, however, we have separated the primary and secondary effects of the operator into distinct transitions which can only be applied sequentially. This does not fit with the intended meaning of the operator, which is that all the preconditions (primary and secondary) are tested in the state before the effects take place. In the previous example this does not make any difference to the behaviour of the rules.

The circumstances under which it makes a difference are:

- When any (primary or secondary) effect deletes a secondary precondition (for a different conditional effect). This is because sequentialising the rules will cause the deleted effect to be unavailable for the application of the rule with the secondary precondition if the deleting rule is applied first. However, rules can be applied in all possible orders so this leads to a problem only when the second rule deletes a precondition of the first rule, so that applying the rules in either order prevents them from both being applied.
- When any (primary or secondary) effect deletes a (primary or secondary) effect (where not both are primary components or are in the same conditional effect). In this case, as an operator, the classical semantics (Lifschitz 1986) causes the add effects to occur after the delete effects and the apparently paradoxical effects are resolved.

A simple example of the first case is an effect of the form:

```
(when (and (q ?y) (p ?x))
      (and (q ?x) (not (p ?x))))
(when (and (p ?y) (p ?x))
      (and (r ?x) (not (p ?x))))
```

In both secondary effects, `(p ?x)` is deleted. If both `when` conditions apply, the condition is deleted once. The proposed compilation of the operators into transitions will not deal with this correctly. The rules created from these effects (supposing no primary pre- or post-conditions affect them) will be of the form: $p_1 \rightarrow q_1$ and $p_1 \rightarrow r_1$. It looks as though a p_1 property must given up to gain either q_1 or r_1, but in fact both can be purchased by giving up a single p_1.

To handle this problem we can identify the common deleted literal and collapse the rules to arrive at a third rule: $p_1 \rightarrow q_1, r_1$. Notice that we cannot replace the other rules with this one, since the conditions cannot be assumed to always apply together. More generally, we cannot assume that two rules generated from the same operator that have a common element on their left-hand-sides will always be exchanging different instances of the property (even if derived from different variables – the variables could refer to the same object). Therefore, such rules have to be combined to collapse the exchanged property into a single instance. (It is interesting to observe that the pathological case, in which the rules derive from primary effects affecting different variables, can lead to a similar problem in the original TIM analysis). There is a minor problem in that the process of collapse could, in principle, be exponentially expensive in the size of the operator, since every combination of collections of rules sharing a left-hand-side element must be used to generate a collapsed rule (in which the shared collection is collapsed into a single instance of each property). In practice the size of these sets of rules is very small and the growth is not a problem. A more significant problem is that the combinations of these rules can lead to further weakening of the possible invariants through over-generation of states.

An example of the second case is:

```
(:action A
  :parameters (?x)
  :precondition (p ?x)
  :effect (and (not (p ?x)) (q ?x)
               (when (q ?x) (and (not (q ?x)) (r ?x)))))
```

Notice that in order to delete an effect that is added by the primary effect of an operator, or another conditional effect, the deleted condition must be a precondition of the effect. The overall behaviour of `(A a)` applied to a state in which `(p a)` and `(q a)` hold is to generate a state with `(q a)` and `(r a)`. This is because the delete effect is enacted before the add effect, so that the net effect on `(q a)` is for it to be left unchanged. Application of the operator to a state in which only `(p a)` holds will yield the state in which only `(q a)` holds.

The rules generated from this operator using the proposal of this section are $p_1 \rightarrow q_1$ and $p_1 \Rightarrow q_1 \rightarrow r_1$. Testing the enablers in application of these rules allows a precise generation of states in both cases. However, it is generally not possible to consider enabling conditions without compromising correct behaviour. Suppose that the additional primary precondition `(s ?x)` and primary effect `(not (s ?x))` are added to the previous operator. Then the rules will become: $p_1, s_1 \rightarrow q_1$ and $p_1, s_1 \Rightarrow q_1 \rightarrow q_1$. It is now impossible to apply the rules sequentially to the property space state $\{p_1, s_1\}$ if we take enablers into account, because the first rule will consume the s_1 property and prevent the second rule from being applied. Consequently, this approach cannot restrict rule application using enabling conditions and we will therefore be forced to generate the unwanted states, weakening the invariants.

Conditional transitions The most radical treatment of conditional effects involves a significant extension of the TIM machinery in order to extend the expressive power of the rules in parallel with the extended expressive power of the operators. This is a less attractive option, since it requires new algorithmic treatments, but this price must be offset against the improved analysis and the more powerful invariants that

can be inferred from domain encodings.

The proposal is to extend the expressiveness of the transition rules to include *conditional transitions*. The conditional component of the transition rule is an additional transition denoted by the keyword `if`. Satisfaction of the condition depends on the presence in a state of both enablers and the start conditions.

$$\begin{array}{lll} a_1 \rightarrow & x_1 & \\ & \text{if} & b_1 \rightarrow y_1 \\ & \text{if} & c_1 \rightarrow z_1 \end{array}$$

Generating conditional transitions Firstly, we must generalise the notion of a PRS into a nested structure to represent operators including `when` conditions. We include an extra field `cnd` to record an embedded PRS for the conditional part of the operator. `op_with_non_exclusive_conditions` then yields the following PRS.

$$\left[\begin{array}{lll} pre: & a_1 & \\ add: & x_1 & \\ del: & a_1 & \\ cnd: & \left[\begin{array}{ll} pre: & b_1 \\ add: & y_1 \\ del: & b_1 \end{array}\right] & \\ & \left[\begin{array}{ll} pre: & c_1 \\ add: & z_1 \\ del: & c_1 \end{array}\right] & \end{array}\right]$$

Now we use the generalised PRSs to generate conditional transition rules. PRS analysis for secondary conditions is essentially the same as the analysis for primary conditions, except that only the adds and deletes of the secondary rule are considered, but all the preconditions of the containing structures must be included.

In general this construction is straightforward, but there is an important case that presents a minor complication. If a conditional effect deletes a primary precondition then the primary precondition will be seen as enabling the outer rule, but it will not appear as a precondition for the conditional effect. One solution is to simply handle the proposition as if it were a precondition of the conditional effect, so that the property appears as a start condition for the conditional rule. However, this leaves the enabling condition outside, apparently required as an additional property for the application of the conditional transition rule. This presents the problem that if enablers are used in determining applicability of rules it will not be clear whether the rule demands one or two copies of the property to be applied. One way to solve this problem is to *promote* the precondition, so that deleted conditions in conditional effects are always explicit preconditions of the conditional effects. To achieve this, a new conditional effect must be added to the original operator, with the deleted literals as its preconditions and all of the original primary effects as its effects. The deleted literals are now removed from the primary preconditions and added explicitly as preconditions to all the other conditional effects. This transformation yields an operator which is equivalent in its effects on a state to the original, although it can, in principle, be applied to a larger collection of states (all states in which the deleted effects are not true – the effect of application is null). Analysis of this new operator yields a conditional rule with the correct structure, distinguishing the case where a property is genuinely an enabling condition from the case where it is actually a copy of the deleted condition in a secondary effect.

Uniting transition rules In TIM the process of uniting is that of combining the collections of properties into a partition such that each transition rule refers, in its start and end components, only to properties in a single set within the partition. This ensures that the construction of extensions of states in property spaces works within a closed subset of the properties used in the domain and that the initial state properties are properly divided between the property spaces to seed the extension process.

In the case of conditional rules, the properties which change between the start and end of each conditional component of each rule must be united into the same subset of the partition, and properties within different conditional components of the same rule are also combined.

$$\begin{array}{l} uniters((Ens \Rightarrow Start \rightarrow End) + Subrules) = \\ \quad (Start - End) \cup (End - Start) \cup \\ \quad \bigcup_{Sr \in Subrules} uniters(Sr) \end{array}$$

This form of uniting ensures that the property spaces remain as small as possible, which improves the quality of the invariants that can be generated and also the efficiency of the analysis. It does, however, impose additional difficulties in the use of rules, since the same rule can now affect the behaviour of objects in multiple spaces (conditional elements might refer to properties in entirely different spaces to the primary effects of the rule). Each rule must be added to every space that it applies to and considered during the extension of each of those spaces separately.

A further change from the process of setting the initial collection of property spaces in STRIPS TIM is that conditional rules can appear to contain attribute rules when, in fact, they are half of a transition rule that is completed by a second "attribute" rule in a conditional effect. For example:

```
(:action B
  :parameters (?x)
  :precondition (p ?x)
  :effect (and (not (p ?x))
               (when (s ?x) (q ?x))
               (when (not (s ?x)) (r ?x))))
```

This operator leads to the rule:

$$\begin{array}{lll} p_1 \rightarrow & \text{null} & \\ & \text{if} & s_1 \Rightarrow \text{null} \rightarrow q_1 \\ & \text{if} & \neg s_1 \Rightarrow \text{null} \rightarrow r_1 \end{array}$$

which might suggest that p_1, q_1 and r_1 should all be considered to be attributes and, consequently, have no useful invariant behaviours. However, it would be better to observe that the behaviour of these properties is actually equivalent to a pair of transitions: $s_1 \Rightarrow p_1 \rightarrow q_1$ and $\neg s_1 \Rightarrow p_1 \rightarrow r_1$ (exploiting negative preconditions, discussed below). Although it might be possible to convert the rules automatically, it becomes much harder to do this in the context of multiple rules referring to other parameters. It is actually easier to manage the rules during extension. The important thing, during initial property space construction, is to avoid labelling properties as attributes on the basis of the structure of *conditional* rules. A decision about which properties to label as attributes must be postponed to the extension phase.

The fact that the properties in a property space can be distributed between primary and conditional rules creates a complication for uniting: it is not enough to put together properties in the same rule. Properties must be combined when they appear in "attribute" rules such as the previous example. It will be noted, however, that the example relies on the form of the conditions of the conditional effects and this feature is further discussed later in the paper.

Extending the state spaces with the conditional transitions Extension is the stage most impacted by the introduction of conditional rules. Conditional rules must be applied to each state in the property space containing them in order to generate a set of reachable states. Thus, the key to exploiting these rules is to understand how to apply the conditional rules to a state.

When a standard transition rule is applied the start conditions are removed from the state and the end conditions added to the result to yield the (single) new state. Conditional rules are applied by removing the primary start properties and then, for each conditional rule, continuing expansion under the assumption that the condition applies and under the assumption that it does not apply. Therefore, there will be 2^n new states generated for n conditional effects (subject to repetition of previously visited states). Although this is potentially exponentially expensive, n is typically a very small value, so that the cost is not a problem. Nevertheless, conditional effects represent a potential source of considerable cost in the TIM analysis (just as they can in planning itself). The hope is that we will have saved significant cost by deferring this combinatorial aspect to the latest possible time, and some redundant processing has been avoided.

There are, in fact, several special cases that can be used to reduce the number of combinations of conditional rule elements that must be considered. Conditional rules can only be applied if the start properties are present in the state to which they are applied. Conditions that are restricted to propositions concerning only the variable that is affected by the transition can usually be restricted by the enabling or start conditions of the rule. Further, the observations of the next section provide for an important collection of situations.

Having allowed possible attribute conditional rules to be entered into property spaces it is extremely important that the extension process monitors the possible existence of increasing attributes in a property space. This possibility also exists in the original TIM analysis and is handled by checking to see whether newly generated states are proper super-states of previously generated states. In such cases, if there is path of transition rules leading from the sub-state to the super-state, the difference between the states represents a collection of attribute properties and they must be stripped from the property space and its rules before extension can be continued. This process might iterate several times before the space settles on a fixed collection of properties. If this collection is actually empty then the properties will, in fact, all be attributes due to the effects of the conditional rules in the space.

Hidden exclusivity In some operators, conditional effects are mutually exclusive because it is not possible for their preconditions to hold simultaneously. In such cases the extension process described above will generate permissive property spaces and weaker invariants. In order to improve the generation process it is necessary to avoid allowing rules to be applied simultaneously when their conditions will prevent it. Further improvement can be made by observing that in many cases the conditional effects are created to ensure that precisely one effect of a collection will be triggered. For example:

```
(:action op1
    :parameters (?y)
    :precondition (a ?y)
    :effect (and
              (not (a ?y))
              (b ?y)))

(:action op2
    :parameters (?y)
    :precondition (b ?y)
    :effect (and
              (not (b ?y))
              (a ?y)))

(:action op3
    :parameters (?x ?y)
    :precondition (p ?x ?y)
    :effect (and
              (when (a ?y) (q ?x))
              (when (b ?y) (r ?x))
              (not (p ?x ?y)) ))
```

In this example, properties a_1 and b_1 may only be exchanged for each other, as illustrated in Figure 1. If, in the initial state, objects only have at most one of the two properties, then the two `when` conditions in `op3` are mutually exclusive.

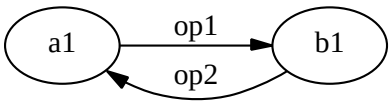

Figure 1: Property space and transitions for properties a_1 and b_1.

The generated transitions would be: $a_1 \rightarrow b_1$, $b_1 \rightarrow a_1$,

$$\begin{array}{ll} p_1 \rightarrow & \text{null} \\ & \text{if null} \rightarrow q_1 \\ & \text{if null} \rightarrow r_1 \end{array}$$

and $p_2 \rightarrow$ null. Failure to observe that the conditional effects are governed by preconditions of which precisely one must be true will mean that no invariants can be generated using the properties p_1, q_1 and r_1. In order to discover these we need two pieces of information to be available during extension: the fact that each conditional effect is governed by a property that, in this case, apply to another parameter and form part of another property space and the fact that the properties governing these effects are the two alternative states in that space. We therefore mark conditional rules with all of the properties that govern their application. Enabling conditions that are properties of other parameters are called *aliens*. We enclose them in square brackets to distinguish them from enablers applying to the parameter governed by the transition rule.

$$\begin{array}{ll} p_1 \rightarrow & \text{null} \\ & \text{if} \quad [a_1] \Rightarrow \text{null} \rightarrow q_1 \\ & \text{if} \quad [b_1] \Rightarrow \text{null} \rightarrow r_1 \end{array}$$

This provides the first piece of information. The second piece is derived from an analysis of the property space containing properties a_1 and b_1. It is important that the analysis makes the latter space available before the use of the rule that depends upon it. Therefore, we perform a dependency analysis to order the property spaces for appropriate expansion order. Where one state space depends on another, an order can be imposed between them that would allow information from the first to be used in the second. Space z depends on space y if transitions belonging to space z have enabling conditions which belong to space y.

Circular dependency amongst state spaces must be handled carefully. One way is to break the cycle arbitrarily and then follow the dependencies that remain. The first space in the chain will then be expanded with the more conservative assumption that no invariants affect the conditions governing the application of conditional effects, possible leading to weakened invariants. Because these weakened invariants can propagate their impact up the chain, it would obviously be best to break the chain in such a way as to minimize the impact that these weakened invariants might have. An alternative solution to the problem of cyclic dependency is to carry out extension on these property spaces in an interleaved computation, leading to a fixed-point. The approach is to apply conditional rules relative to the dependencies on property spaces using the states that have been generated *so far*. The extension process must then iterate around the cycle of interdependent spaces, adding new states as new enabling states are added to the spaces on which later spaces depend. This iterative computation will have to be restarted if any of the properties in a space is identified as an attribute, as described previously.

Managing combinations of subrules In this section we give more detail about how to use information from spaces for which the extension process is already completed, together with annotation of the subrules with alien enablers, to restrict the combinations in which subrules of a conditional rule can fire.

For a subrule to fire, we require that for each variable-space combination occurring as an alien enabler in the subrule, there is at least one state that satisfies the enabling property.

The coupling between subrules is captured because valid combinations of subrule firings are those in which the enabling conditions can be simultaneously satisfied. Enabling conditions are satisfied, if, for each variable, and each space in which it occurs, there is a non-empty set of satisfying states.

The approach adopted depends on amending the TIM algorithm with the following steps

1. During rule construction, each subrule is annotated with alien enablers, each consisting of a triple $aliens(subrule)$ is a set of $\langle property, space, var \rangle$, where var is the variable from which the enabler was generated in the planning operator.
2. A graph of dependencies between spaces is constructed from the alien enablers.
3. Spaces are constructed in order, according to topological sort of the dependency graph. Cycles must be broken as discussed above.

Now we define the valid combinations of rule firings by describing a set of variables, and constraints which must hold between them.

For each variable-space combination in alien enablers, we have a variable $state(variable, space)$ whose domain ranges over possible states of $variable$ in $space$.

$$state(var, sp) \in states(sp)$$

For each property-variable-space combination in alien enablers, we have a boolean constraint variable, $sat(property, variable, space)$, whose value indicates whether the property is satisfied.

$$sat(prop, var, sp) \in \{0, 1\}$$

For each subrule, we have a variable $f(subrule)$, whose domain is {0,1}, which records the whether or

not the subrule can fire. Note that since the possible values remaining in the domain are attached to the variable, we can describe never {0}, sometimes {0,1}, and always {1}.

Now we attach constraints between these variables as follows:

Between the $sat(property, variable, space)$ variables, and the $state(variable, space)$, we require that the property is satisfied iff var is in a compatible state in $space$.

$$\forall subrule \in subrules$$
$$\forall \langle prop, var, sp \rangle \in aliens(subrule).$$
$$sat(prop, var, sp) \Leftrightarrow has\_prop(state(var, sp), prop)$$

Between the $f(subrule)$ variables and the $sat(property, variable, space)$ variables of its alien enablers, we require that $subrule$ fires iff the conjunction of its alien enablers is satisfied.

$$f(subrule) \Leftrightarrow \bigwedge_{\langle prop,var,sp\rangle \in aliens(subrule)} sat(prop, var, sp)$$

These constraints are used to determine which combinations of subrules may fire together.

Universally Quantified Effects

Since we are concerned with the transitions made by individual objects, it is not necessary to expand the operator in advance with every instantiation of the quantified variable, as is done by (Gazen & Knoblock 1997).

```
(:action one_forall
   :parameters (?a - t1)
   :precondition (p ?a)
   :effect (and (not (p ?a))
                (q ?a)
                (forall (?b - t2)
                   (when (r ?a ?b)
                         (and (not (r ?a ?b)) (s ?a ?b) )))))
```

The effect inside the quantifier may occur as many times as there are instantiations for the quantified variable `?b`.

For `?b` itself, we can generate a transition rule exactly as if it was an ordinary parameter of the operator. The number of objects making the transition is not relevant, as any object belonging to this state space experiences only a single transition. In the above example, analysis for the variable `?b` gives the transition $r_2 \rightarrow s_2$.

Now consider the transitions for the variable `?a`. Outside the quantifier, `?a` undergoes a single transition $p_1 \rightarrow q_1$. Inside the quantifier, `?a` undergoes a transition $r_1 \rightarrow s_1$, but the number of times the transition may occur depends on the number of instantiations the quantified variable can take. This reasoning applies to any variable occurring inside the scope of the quantifier, which occurs in an effect together with the quantified variable.

We must again resort to a new notation to describe the resulting transition rules. We use * to indicate that the transition inside the quantifier may be performed an unknown number of times.

For the variable `?a`, we have:

$$\begin{aligned} p_1 \rightarrow \quad & q_1 \\ & (r_1 \rightarrow s_1)^* \end{aligned}$$

In the extension process, the interpretation of the starred rule is that the inner transition may occur any number of times.

Observe that in this example, the transition inside the quantifier is an exchange of properties. It may also occur that the transition inside the quantifier involves properties simply being gained or lost. Such properties are considered to be attributes in the TIM system, and are processed in a separate way. The presence of such rules otherwise leads to non-termination of the extension process, as attributes may be added without limit. Attribute rules are less useful for discovering invariants, and it is unfortunate if properties are considered as attributes unnecessarily. This is harmful to the TIM analysis because any property which becomes an attribute also makes anything for which it can be exchanged into an attribute.

It is our contention that properties will not be made into attributes unnecessarily as a result of analysing the quantified effect. Where properties are exchanged, the whole exchange will normally take place inside the quantifier, as in the example above.

An example where a quantified effect correctly gives rise to attribute rules can be seen in the briefcase `move` operator given above. There the at_2 properties are attributes — the portables at a location may be gained and lost without exchange.

An awkward example can be found in the Schedule domain, in which predicates of the form `(painted ?object ?paint)` are typical. All operators in the domain which mention `painted` include a deletion which is universally quantified over possible paint colours, as in the example operator shown below. Some of the operators also add a single `painted` effect. Hence all operators which touch an object's $painted_1$ property result in a state with either 0 or 1 instances of the property for that object. If this holds also in the initial state, it is an invariant.

```
(:action do-immersion-paint
    :parameters (?x ?newpaint)
    :precondition (and
                    (part ?x)
                    (has-paint immersion-painter ?newpaint)
                    (not (busy immersion-painter))
                    (not (scheduled ?x)))
    :effect (and
                (busy immersion-painter)
                (scheduled ?x)
                (painted ?x ?newpaint)
                (when (not (objscheduled))
                      (objscheduled))
                (forall (?oldpaint)
                    (when (painted ?x ?oldpaint)
                          (not (painted ?x ?oldpaint))))))
```

It is problematic that the quantified effect makes it appear that $painted_1$ is a decreasing attribute. In earlier versions of TIM, the appearance of decreasing attribute transitions always led to the creation of a separate attribute space. However, if the attribute may only decrease, it can lead to only a finite number of states, so it is safe to have rules of this kind in a property space. It is important to take this approach, as invariants will otherwise be lost.

Negative preconditions

Our treatment of negative preconditions is currently the most restrictive described here. We use a similar technique to (Gazen & Knoblock 1997): for each predicate that may appear as a negative precondition, we create a new predicate to represent the negative version of that precondition. This predicate must exactly complement the positive use of the predicate and it replaces, in preconditions, the use of the negative literal.

For those predicates which occur anywhere as a negative precondition, we must do the following to the effects of every operator:

- Wherever the positive version of the predicate appears as a delete effect, the negative version must appear as an add-effect.
- Wherever the positive version of the predicate appears as an add effect, the negative version must appear as a deleted effect.

We complete the initial state of the problem description as follows: For each predicate and each object that may instantiate the predicate, if there is no positively-occurring fact, we add the negative version of the fact.

In the case of negatively-occurring predicates with multiple arguments, the completion of initial state is very expensive, and would, in any case, lead to a much weaker domain analysis. For our processing we impose the restriction that only predicates of a single argument are handled. This also allows the transformation to be performed, not on the operator itself, but to be processed at the level of PRSs.

We believe that predicates with more than a single argument could be handled efficiently using a representation in which properties which properties are counted, but this remains an area of further investigation.

Results

A prototype system has been implemented which successfully produces the expected invariants for all the cases considered in this paper, except those relying on a full treatment of universal quantifiers. Work on the handling of universal quantifiers is currently in progress. Additional computational overheads for handling the conditional effects are negligible.

In the example of hidden exclusivity between conditional transitions, the system successfully detects that exactly one of the subrules may fire, and thus extension does not over-generate states or unnecessarily weaken the analysis by creating an attribute space. The resulting space for the properties p_1, q_1 and r_1 is shown in Fig. 2.

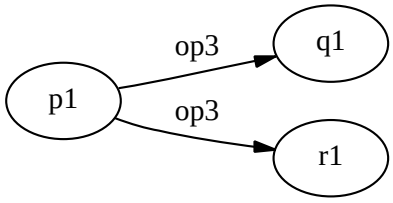

Figure 2: Property space and transitions for properties p_1, q_1 and r_1.

Directly from the state space, we get that states in this space allow exactly one of the properties $\{p_1, q_1, r_1\}$, and from this the following invariants are generated:

```
FORALL x:T1. (Exists y1:T0. p(x,y1) OR r(x) OR q(x))
FORALL x:T1. NOT (Exists y1:T0. p(x,y1) AND r(x))
FORALL x:T1. NOT (Exists y1:T0. p(x,y1) AND q(x))
FORALL x:T1. NOT (r(x) AND q(x))
```

Related work

Both the system of Rintanen (Rintanen 2000) and Gerevini and Schubert's DISCOPLAN (Gerevini & Schubert 1998) rely on generating an initial set of candidate domain invariants. These are then tested against the operators to see if they hold after the application of the operator. Candidate invariants which are not preserved are either discarded or strengthened by adding extra conditions and tested further. In both systems, discovered invariants can be fed back, to help discover further conditions.

The current version of DISCOPLAN (Gerevini & Schubert 2000) has been extended to handle conditional effects, types and negative preconditions, but does not handle universal quantification.

An interesting question is whether DISCOPLAN can detect and exploit the occurrence of mutually exclusive secondary conditions. Our tests with the current version of DISCOPLAN indicate that it cannot. The account in (Gerevini & Schubert 2000) discusses the feeding back of discovered invariants into the process, by using this information to expand operator definitions. It appears that currently, only the implicative constraints are fed back in this way. To correctly resolve this problem would require XOR constraints to be fed back into the processing of secondary conditions.

Conclusion

In this paper we have explored the extension of TIM to handle a subset of ADL features. The features we have not considered are those that allow fuller expressiveness in the expression of preconditions: quantified variables and arbitrary logical connectives. There are significant difficulties in attempting to handle these features, since

the opportunity to identify the specific properties associated with particular objects is obscured by the possibility for disjunctive preconditions combining properties of different objects, universally quantified variables that allow reference to arbitrary objects, and nested expressions that can involve exponential expansion if conversion to disjunctive normal form is used. We have also not considered a full treatment of equality and inequality propositions, nor of arbitrary negated literals.

The features that have been considered, and for which there are extensions that allow TIM to extract invariants, include conditional effects, quantified effects, types and a restricted form of negative preconditions. These features form the core of those used in existing ADL domain encodings, with the exception of the ADL Miconics-10 domain used in the 2nd International Planning Competition in 2000. The most difficult feature to handle is the use of conditional effects and we have shown that there are a variety of possible approaches, each with advantages and disadvantages. The most powerful approach is the extension of the underlying TIM machinery, rather than an attempt to preprocess conditional effects out of the operators in order to reuse the STRIPS TIM machinery. This extended machinery complicates the entire sequence of analysis phases conducted by TIM and we have described the effects that are implied for each stage in turn.

The next stages of this work include completion of a full implementation of all of the features described in this paper, a further exploration of the treatment of negative preconditions and experimentation with the system on existing ADL domains.

Acknowledgements

The work was funded by EPSRC grant no. R090459.

References

Bacchus, F., and Kabanza, F. 2000. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence* 116.

Doherty, P., and Kvarnstrom, J. 1999. TALplanner: An empirical investigation of a temporal logic-based forward chaining planner. In *Proceedings of the 6th Int'l Workshop on the Temporal Representation and Reasoning, Orlando, Fl. (TIME'99).*

Fox, M., and Long, D. 1998. The automatic inference of state invariants in TIM. *Journal of Artificial Intelligence Research* 9:367–421.

Fox, M., and Long, D. 2000. Utilizing automatically inferred invariants in graph construction and search. In *International AI Planning Systems conference, AIPS 2000, Breckenridge, Colorado, USA.*

Gazen, B. C., and Knoblock, C. A. 1997. Combining the expressivity of UCPOP with the efficiency of graphplan. In *ECP*, 221–233.

Gerevini, A., and Schubert, L. K. 1998. Inferring state constraints for domain-independent planning. In *AAAI/IAAI*, 905–912.

Gerevini, A., and Schubert, L. 2000. Extending the types of state constraints discovered by DISCOPLAN. In *Proceedings of the Workshop at AIPS on Analyzing and Exploiting Domain Knowledge for Efficient Planning, 2000.*

Kautz, H., and Selman, B. 1998. The role of domain-specific knowledge in the planning-as-satisfiability framework. In *Proceedings of the 4th International Conference on AI Planning Systems.*

Lifschitz, E. 1986. On the semantics of STRIPS. In *Proceedings of 1986 Workshop: Reasoning about Actions and Plans.*

Long, D., and Fox, M. 2000. Recognizing and exploiting generic types in planning domains. In *International AI Planning Systems conference, AIPS 2000, Breckenridge, Colorado, USA.*

Pednault, E. 1989. ADL: Exploring the middle ground between strips and the situation calculus.

Porteous, J.; Sebastia, L.; and Hoffmann, J. 2001. On the extraction, ordering, and usage of landmarks in planning. In *Proceedings of European Conference on Planning.*

Rintanen, J. 2000. An iterative algorithm for synthesizing invariants. In *AAAI/IAAI*, 806–811.

van Beek, P., and Chen, X. 1999. Cplan: A constraint programming approach to planning. In *Proceedings of the 16th National Conference on Artificial Intelligence, Orlando, Florida.*