Automatic Discovery and Exploitation of Domain Knowledge in Planning [*]

Steven A. Wolfman
Department of Computer Science & Engineering
University of Washington, Box 352350
Seattle, WA 98195–2350 USA
wolf@cs.washington.edu

Abstract

Recent advances in AI planning technology have drastically improved the capabilities of modern planners. However, these advances have left research in domain specialization behind; many older specialization techniques are no longer applicable to modern planners. This paper explores how to automate domain specialization in a modern planning as constraint satisfaction engine using existing analysis and reformulation techniques. In particular, the paper examines how TIM, PABLO, and Crawford et al.'s symmetry-breaking predicates aid in automating the CPlan planning as constraint satisfaction system.

[*]Thanks to Jeremy Buhler, Craig Kaplan, Rachel Pottinger, and Denise Pinnel for their help. Thanks also to my generals committee for their effort and ease of scheduling. This work was funded in part by the ARCS Foundation Tom and Barbara Cable Fellowship and by a National Science Foundation Graduate Fellowship.

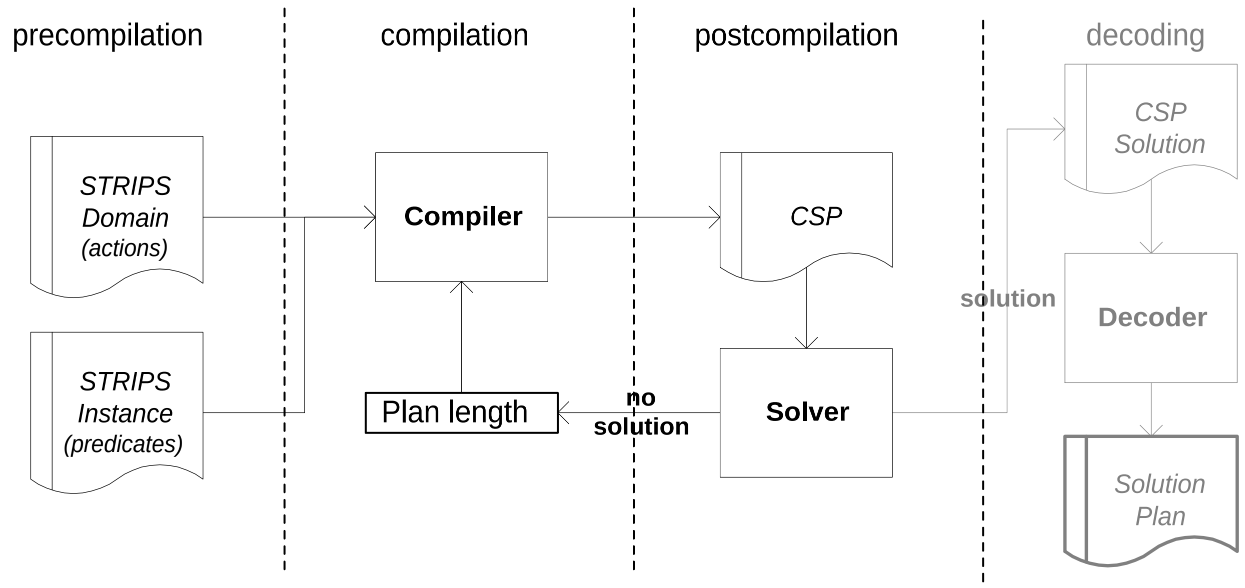


Figure 1: The standard structure of a planning as constraint satisfaction system. The system has four stages. The user inputs the planning problem (domain and instance) in the precompilation STRIPS stage. Then, the system compiles the problem into a series of constraint satisfaction problems (CSPs) each instantiated for a different plan length. These CSPs are then solved by a constraint solver. If the CSP is inconsistent, the compilation begins anew with a larger plan length; otherwise, the decoding stage transforms the solution into a plan satisfying the original STRIPS problem. This paper focuses on the first three stages.

1 Introduction

The traditional approach to solving planning problems has been to create special purpose algorithms for reasoning about planning [Penberthy and Weld, 1992; Blum and Furst, 1995]; however, rapid advances in algorithms for constraint satisfaction and its variants have now made it feasible and highly effective to solve general planning problems by first compiling them into constraint satisfaction problems (CSPs) and then solving them with an existing constraint solver. Figure 1 illustrates a planning as constraint satisfaction system (CSP planner).

These planning as constraint satisfaction systems (CSP planner) take planning problems as input[1] and compile them at successively increasing plan lengths into constraint satisfaction problems (CSPs). The CSPs produced by planning as constraint satisfaction systems have solutions if and only if the original planning problem has a solution plan of the appropriate length. Such a solution can then be decoded into a plan which solves the original problem.

Unfortunately, much of the extensive research into automatic discovery and exploitation of domain knowledge in previous planners does not transfer well to these new constraint satisfaction planners. Without the capacity to automatically analyze planning domains, current CSP planners rely on either expert human knowledge or a handful of generally useful constraint satisfaction heuristics to exploit the special structure of each domain.

The CPlan system is a CSP planner which relies on expert human knowledge [van Beek and Chen, 1999]. CPlan's compilation and decoding processes are performed by hand using a highly expressive variant of constraint satisfaction. The result is an extremely fast planner, but the price is painstaking expert scrutiny of each new domain. Automating CPlan's domain specialization process would result in a powerful and convenient new planning system.

In this paper, I describe the domain knowledge that has proven useful in CPlan, and I explain how several

[1]In this paper, I restrict my attention to STRIPS planning problems [Fikes and Nilsson, 1971]. A STRIPS planning problem is a domain — a set of actions which require certain preconditions and can add or delete facts, called predicates, from the world state — and an instance component — the initial world state and the set of goal facts. The solution to a STRIPS problem is a plan, a series of actions which lead from the initial state to the goal.

existing analysis and reformulation techniques can be used to automate the discovery and exploitation of this knowledge. I focus on three systems which treat different stages of the planning as CSP solution process: the TIM system [Fox and Long, 1998] works at the precompilation STRIPS stage, analyzing planning problems for type and invariant information; the PABLO abstraction system [Christensen, 1990] suggests a new style of compilation which may result in better focused search by initially ignoring unimportant details; and Crawford et al.'s symmetry-breaking system [Crawford *et al.*, 1996] works at the postcompilation CSP level, detecting and breaking *symmetries*, interchangeable groups of variables or values, in the compiled problem.

Throughout the paper, I will use the logistics planning domain to illustrate examples. The logistics domain consists of packages, trucks, and planes. The packages can be loaded into and unloaded from the trucks and planes. Trucks can move between sites in a city, and planes can move between special sites (airports) in different cities. The most important predicates are the **at** and **in** predicates. **at**(x, y) states that object x inhabits site y. If a truck or plane y contains a package x, x is not **at** a site at all but **in** y: **in**(x, y). All other predicates are unary type predicates (*e.g.*, packages are **small**, sites are **location**s, and trucks and planes are **vehicle**s). The `Load` and `Unload` actions move packages into and out of vehicles, and the `Drive` and `Fly` actions move trucks and planes, respectively, between sites.

The paper is organized as follows. Section 2 describes the CPlan methodology. Each of the three following sections describes one of the analysis and reformulation techniques mentioned above and how it can be applied to automating CPlan: Section 3 describes TIM, Section 4 describes PABLO, and Section 5 describes Crawford et al.'s symmetry-breaking system. Section 6 mentions some related work, Section 7 draws conclusions about the effort to automate CPlan, and finally Section 8 describes future research directions which stem from this work.

2 CPLAN

CPlan is a methodology for solving planning problems [van Beek and Chen, 1999]. The CPlan architecture describes a technique for hand-compiling planning problems into Constraint Satisfaction Problems (CSPs)[2]. The resulting CSP is then solved using an existing constraint satisfaction algorithm [Prosser, 1993], and the solution is transformed by hand into a plan which solves the original problem. CPlan is an extremely fast planner but relies on expert knowledge; automating the discovery of this knowledge would produce a powerful new planning system. This section describes the CPlan methodology then highlights the most important uses of domain knowledge in CPlan.

The CSPs which CPlan produces are composed of variables and constraints. Each variable has a finite domain, and a solution to the CSP assigns exactly one value to each variable from its domain. Each constraint acts on a set of variables. When passed values for the set of variables it affects, a constraint returns either *true* — the value assignment is consistent with the constraint — or *false* — the value assignment is inconsistent with the constraint. The reasoning process by which the constraint decides whether a set of values is consistent is described by a user-defined function. Therefore, the system cannot understand or reason with the constraints themselves; rather, it invokes them and reasons with their return values. Such non-declarative constraints are termed *implicit constraints.*

The CPlan methodology describes two classes of constraints which together comprise the CSP encoding. Core constraints are the normal constraints produced by the compilation process illustrated in Figure 1 and are required for soundness and completeness. The core constraints in the CPlan methodology are the action constraints which ensure that changes in the state of the world are justifiable by action applications and the state constraints which ensure consistency within a given state [Schubert, 1994; van Beek and Chen, 1999].

Heuristic constraints provide domain-specific information which facilitates the solution process, but they prune no more *invalid* solutions than those already removed by the core constraints. Each CPlan encoding uses a variety of highly domain specific and sometimes quite intricate heuristic constraints to improve its performance; however, Van Beek and Chen identify the following handful of heuristic constraints as the most important and effective:

Distance constraints provide upper and lower bounds on the number of steps to change a predicate's value. Lower bounds allow the planner to avoid overly optimistic planning choices. Upper bounds allow the planner avoid ineffectual action sequences and actually terminate rather than endlessly searching for longer plans.

Symmetric values constraints are described by Fox and Long as constraints which eliminate redundant search over symmetric values for CPlan variables. Depending on the encoding style, however, similar symmetries might arise between variables rather than between variable values. The fundamental quality

[2]For a review of CSPs, see [Meseguer, 1989].

of these symmetries is that they arise between interchangeable objects or sets of objects from the original planning problem.

Action choice constraints eliminate redundant search over different action orderings. An action choice constraint allows the planner to ignore all but one of a group of reordered action sequences, all of which have the same eventual effect.

Capacity constraints are constraints which make effective use of CPlan's capacity for numerical reasoning. CPlan's CSPs can contain numerical variables with finite domains such as integer variables with both upper and lower bounds.

Domain constraints are restrictions on the allowed values of variables. These can be viewed as super-specializations of object types which restrict functionality according to the circumstances of a particular instance or domain.

In addition, CPlan uses careful choices of variables and values in its encodings to exploit regularities in domains (*e.g.*, the **in** and **at** predicates in the logistics domain can be represented by encoding packages as variables with domains that includes sites, trucks, and planes)[3].

It is the domain specialized instantiation of heuristic constraints and the careful use of variables and values which set CPlan apart from other CSP planners. CPlan's methodology crosses the boundary between Artificial Intelligence (AI) and Operations Research. CPlan's carefully tuned, hand-crafted encodings are an application to an AI problem of the meticulous investigation of individual scenarios common in Operations Research. The result is a planning system which almost uniformly outperforms other planners in speed and success rate.

Still, from an AI point of view, painstakingly encoding each domain and instance with expert knowledge is unsatisfying as a solution to the problem of general planning. So, CPlan should not be seen as a new contender among the state-of-the-art planning systems, rather it should be viewed as a guide for where best to expend effort in automating the domain specialization process.

An automated version of CPlan must produce core constraints and heuristic constraints and make effective use of CPlan's variant of constraint satisfaction. However, because every CSP planner must produce core constraints, there is already a substantial body of literature on automating their compilation. One style of encoding core constraints, the "state-based" encoding of Kautz and Selman [Kautz *et al.*, 1996], strongly resembles CPlan's action and state constraints. However, two other substantial groups of encodings exist: state-space encodings [Kautz *et al.*, 1996], of which state-based encodings are a member, and causal encodings [Mali and Kambhampati, 1999]. Many of the different encodings offer tradeoffs which make them attractive as core constraints for automated CPlan. These tradeoffs are described in [Ernst *et al.*, 1997] and [Mali and Kambhampati, 1999]; thus, I will only mention in this paper those benefits and costs which directly bear on automating CPlan. Most of these encodings have already been automated and could be used as is to provide CPlan's core constraints.

Therefore, the three main challenges in automating CPlan are: choosing an appropriate core constraint encoding, automating the creation of CPlan's heuristic constraints, and making effective representation choices (variable identities and domains) in the final CSP. In the following sections, I focus on these issues and how they can be automated and exploited in CPlan.

3 TIM

TIM, the Type Inference and analysis Module, performs a powerful analysis on STRIPS problems [Fox and Long, 1998]. This analysis produces invariants which can be used to help automate the discovery of distance constraints and to improve the choice of variable identities and domains in automated CPlan's CSP encoding.

TIM works by examining a simplified version of an input planning domain. Its simplification process begins by splitting actions into fragments, each of which describes the action from the point of view of one of its parameters. In the process, TIM replaces full predicates with predicate fragments. For example, a predicate which expresses that a package is in a truck, **in**(*package*, *truck*), would be replaced by two predicates: the first denotes that the package has the property of being in something, and the second denotes that the truck has the property of containing something. In general, an n-ary predicate $P(x_1, \ldots, x_n)$ will be split into n predicates $P_1(x), \ldots, P_n(x)$.

Figure 2 shows how TIM fragments the `Load` action from the logistics domain. The first fragment constructed from this action states that an object which is **small** and has the property of being **at** somewhere

[3]There are other issues which must be addressed in a CPlan encoding which are not relevant to the focus of this paper: CPlan's variables must be set as either visible or hidden (derivable from the visible variables), and CPlan's constraints have weightings that assist in choosing an order of evaluation for the constraints. I will briefly describe related work which bears on these issues in Section 6.

```
Load (package, site, vehicle)
   pre: small(package), location(site), vehicle(vehicle)
        at(vehicle, site), at(package, site)
   add: in(package, vehicle)
   del: at(package, site)
```

```
Load_1
   pre: $\mathbf{small}_1$,
        $\mathbf{at}_1$
   add: $\mathbf{in}_1$
   del: $\mathbf{at}_1$
```

```
Load_2
   pre: $\mathbf{location}_1$,
        $\mathbf{at}_2$, $\mathbf{at}_2$
   add:
   del: $\mathbf{at}_2$
```

```
Load_3
   pre: $\mathbf{vehicle}_1$,
        $\mathbf{at}_1$
   add: $\mathbf{in}_2$
   del:
```

Figure 2: *Top*: the `Load` action from the logistics domain. *Bottom*: the fragments of `Load` constructed by TIM. Fragment $\texttt{Load}_1$ refers only to the first argument to `Load`, fragment $\texttt{Load}_2$ refers only to the second, and fragment $\texttt{Load}_3$ only to the third.

can give up its **at** property to gain the property of being **in** something. The other two are similarly object-centered fragments of the original `Load` action focusing on the site and vehicle.

Action fragments can be used to knit together predicate fragments into groups which represent types, sets of functionally similar objects[4]. This process is based on a a motivating idea which is unstated in Fox and Long's work: each predicate induces a type on each of its fields. This is clear for the unary predicates **small**, **location**, and **vehicle**, but less clear for predicates like **at**. However, $\mathbf{at}(x, y)$ also intuitively induces a type on each of its arguments: x is of a type that can have a location and y is a location. An object belongs to the type defined by all the predicate fragments in which it can take part.

Since actions define the transition from one predicate to another, it is the structure of the actions which defines the type hierarchy. For example, $\texttt{Load}_1$ allows objects which have the properties $\mathbf{at}_1$ and $\mathbf{small}_1$ to exchange these for the properties $\mathbf{in}_1$ and $\mathbf{small}_1$. Thus, every object which can be $\mathbf{at}_1$ and $\mathbf{small}_1$ can also be $\mathbf{in}_1$ and $\mathbf{small}_1$, and the set of objects which can have the type $\{\mathbf{small}_1, \mathbf{at}_1\}$ is a subset of the set of objects which can have the type $\{\mathbf{small}_1, \mathbf{in}_1\}$. Therefore the type $\{\mathbf{small}_1, \mathbf{at}_1\}$ is itself a subtype of the type $\{\mathbf{small}_1, \mathbf{in}_1\}$.

In fact, an action fragment is more than just a type relationship; it is a schema of type relationships. For example, by the same logic as above, $\texttt{Load}_1$ also shows that the set of objects that can be $\mathbf{small}_1$, $\mathbf{at}_1$, and $\mathbf{blue}_1$ is a subtype of the set of objects that can be $\mathbf{small}_1$, $\mathbf{in}_1$, and $\mathbf{blue}_1$. In general, the type defined by the precondition predicate fragments is a subtype of the type defined by the effected preconditions — the preconditions plus the add list minus the delete list. Moreover, this relationship holds for any subtype of the precondition type and the *analogous* subtype of the effected precondition type.

Each of the other `Load` fragments also creates type constraints. Indeed, each fragment of every action imposes constraints on the types of objects. These constraints together define a type hierarchy which, in Fox and Long's terms, is neither *over-discriminating* — distinguishing functionally identical objects — nor *under-discriminating* — grouping functionally distinct objects. However, TIM does not reason using this rule; instead, it uses a simplified version of this rule which is under-discriminating. TIM makes two simplifications to the type hierarchy.

First, TIM assumes that action fragments define an equivalence rather than subtype relationship[5]. Although not justified by Fox and Long, this seems a reasonable assumption for planning domains since they usually allow the effects of a sequence of actions to be undone by further actions. If an action's effects *are* reversible, then the action's preconditions are both subtypes of and supertypes of its effected preconditions. In this case, the preconditions and the effected preconditions do define equivalent types.

TIM's assumption of equivalence will certainly never cause it to over-discriminate, but it will cause under-discrimination. For example, in a logistics domain with **fuelled** and **unfuelled** predicates for the vehicles

[4]For this description, types are sets of objects. One type is a subtype of another if and only if the first type's set of objects is a subset of the second type's.

[5]TIM provides special treatment for action fragments which, like $\texttt{Load}_2$, add or remove predicate fragments without exchanging them for new fragments; for more details see [Fox and Long, 1998].

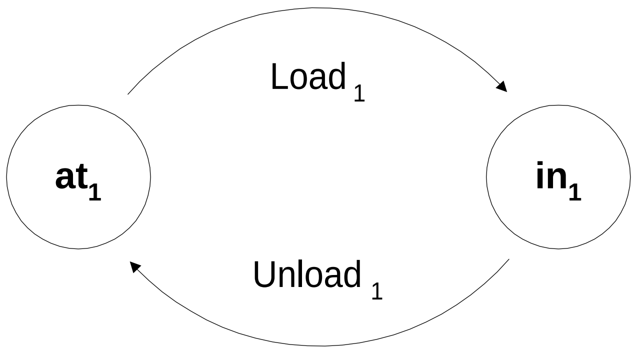


Figure 3: An FSM for the $\mathbf{at}_1 + \mathbf{in}_1$ type space. Each state is a bag of predicate fragments which objects in this space can achieve. The arcs represent action fragments which allow objects to move between the states.

but no action for refueling, vehicles which are initially **fuelled**, and thus capable of moving, are clearly functionally distinct from **unfuelled** vehicles. However, TIM will replace the actual subtype relationship between $\mathbf{fuelled}_1$ and $\mathbf{unfuelled}_1$ with an equivalence, putting both **fuelled** and **unfuelled** vehicles into the same type[6].

TIM's second simplification is to define types only in terms of single predicate fragments. For example, the fragment $\texttt{Load}_1$ from Figure 2 would cause TIM to make the types $\mathbf{in}_1$, $\mathbf{at}_1$, and $\mathbf{small}_1$ equivalent rather than making the two types $\{\mathbf{small}_1, \mathbf{at}_1\}$ and $\{\mathbf{small}_1, \mathbf{in}_1\}$ equivalent.

TIM redresses these simplifications somewhat by ignoring predicate fragments which appear in both the preconditions and the effected preconditions; these "unchanged" predicate fragments include those which appear only in the precondition list and those which appear in both the add and delete lists. Such predicate fragments can be safely ignored because they add no extra type information: the type defined by a single predicate fragment is already equivalent to itself.

With these simplifications, any predicate fragment which appears in an `add` or `delete` list of an action fragment is grouped into the same type as every other predicate fragment appearing in the action fragment's `add` or `delete` list. So, types are defined by the transitive closure of predicate fragments which share `add` and `del` lists of an action fragment.

TIM now creates a new structure called a type space for each type. The type spaces describe the behavior of all the objects of a given type. TIM fills out the type spaces by inserting each object into the spaces representing each of its types. An object participates in each of the type spaces of its initial predicate fragments' types (the fragments true for the object in the initial conditions). An object's initial state in each type space is defined by the combination of predicate fragments from that type space which are true for the object in the initial state. Each starting state is a bag of predicate fragments. Thus, the starting state of an object in a given type space may include several predicate fragments from the initial conditions which were all grouped into that type or even several copies of the *same* predicate fragment. Objects' full types are defined by the set of type spaces in which they take part.

In the final stage of its setup, TIM derives all states reachable in each type space from the states induced by the initial conditions. Anywhere a state contains all the predicate fragments in the delete list of an action fragment, TIM applies that action fragment to the state to derive a new state; the deleted predicate fragments are replaced by the predicate fragments in the add list. This process continues until quiescence[7]. The resulting graph of states together with the transitions which created them define a finite state machine (FSM).

Each type has an associated FSM. Figure 3, for example, is an FSM for the type defined by $\mathbf{at}_1$ and $\mathbf{in}_1$. These type spaces and their FSMs are the primary data structures in TIM. Using these spaces and the

[6]Even if the domain does have a reversing action (as with the `Refuel` action), there is still a subtle difference between TIM's assumption and the true type hierarchy; for example, it is not the types $\mathbf{fuelled}_1$ and $\mathbf{unfuelled}_1$ which are equivalent with a `Refuel` action, but the types $\{\mathbf{at}_1, \mathbf{fuelled}_1\}$ and $\{\mathbf{at}_1, \mathbf{unfuelled}_1\}$.

[7]Again, TIM provides some special treatment. This time, it is for spaces with transitions that allow creating copies of a predicate fragment without changing the state otherwise. Again, I will overlook this special treatment since it has little bearing on invariants, but it is interesting to note that the type information derived by this process is over-discriminating.

original STRIPS planning problem, TIM discovers the following invariants:

Identity invariants — Identity invariants derive from spaces in which no state contains more than one instance of a given predicate fragment. For example, a package can only be **in** one place at a given time. This can be discovered in the type space of Figure 3 because no state in the space contains more than one copy of the $\mathbf{in}_1$ fragment. From this fact, TIM derives the invariant that if a package has the **in** relationship with two objects, those objects must be equal. TIM also derives a similar invariant for $\mathbf{at}_1$ in this space.

In general, an identity invariant for a space defining type T in which predicate fragment P_k appears at most once in any given state would read that if an object of type T appears in predicate P in place k with two different sets of other arguments to P, those other arguments' values must be equal.

Fox and Long also describe an extension to identity invariants which applies them to spaces in which an object can attain at most k instances of a given fragment.

State membership invariants — Since type spaces are closed — all transitions lead to another state inside the type space — an object which starts in some state in a type space must always inhabit a state in that type space. For example, each state in the type space from Figure 3 contains either the $\mathbf{at}_1$ fragment or the $\mathbf{in}_1$ fragment; so, packages must always be either **in** something or **at** somewhere.

State membership invariants generalize this principle. TIM can construct a state membership invariant for every object which is a member of a type space T containing predicate fragments P_1 through P_n to the effect that at any time, one of the predicates represented by the fragments $P_1, \ldots, P_k$ holds for that object; in other words, for one of those fragments, there exist other objects to fill out the predicate's arguments such that the predicate holds in the current state.

Uniqueness invariants — If, in a given type space, two predicate fragments never occur in the same state, TIM can assert that an object of that type will never take part in both predicate fragments at once. In the type space from Figure 3, for example, $\mathbf{at}_1$ and $\mathbf{in}_1$ never appear in the same state; therefore, TIM can specify that a package can never be both **at** someplace and **in** something.

Uniqueness invariants implement this idea. TIM can construct a uniqueness invariant for any two predicate fragments P_j and Q_k which inhabit the same type space but never the same state. The invariant states that no object can simultaneously take on both properties, P_j and Q_k.

Fixed resource invariants — Consider a logistics domain with no packages, no `Load` action, and no `Unload` action. In this simplified domain, an object can change the location it is **at** only by using the `Move` action. Whenever an object `Move`s, it gives up the property of being **at** some location and takes on the property of being **at** some other location. Therefore, an object is always **at** the same number of locations.

Fixed resource invariants describe predicates which are always exchanged in balance such as `at` above. The invariants state that the same number of true instances of a given fixed predicate occurs at every step of the plan. TIM discovers fixed resource invariants at the level of the original STRIPS problem description. For a given predicate, if, in every action which deletes it, it is also added in the same quantity, that resource is fixed. In the original logistics problem, this means that TIM discovers that **small**, **location**, **truck**, and all the other unary type predicates are fixed resources, but **at** is not fixed because packages can exchange being **at** a location for being **in** a vehicle.

These four invariants provide a substantial amount of domain specific information, but they could all also be improved. Identity, state membership, and uniqueness invariants could be strengthened by using the precise construction of the type hierarchy described above. This would make TIM's type distinctions finer. For example, in a logistics domain with fuel but no refueling action, the type $\mathbf{fuelled}_1$ would be a subtype of — rather than the same type as — the type $\mathbf{unfuelled}_1$. Finer typing would allow TIM to discover more invariants and produce more restrictive type classifications.

Improving fixed resource invariants requires a new viewpoint. Although they are described separately by Fox and Long, fixed resource invariants can be viewed as a combination of identity and state membership invariants for predicates. A fixed resource invariant (of size one) states that there exists some set of arguments for which a predicate is true (state membership) and that if two different sets of arguments make a predicate true, those two sets must be equivalent (identity).

This connection to identity and state membership invariants suggests an extension to fixed resource invariants using reasoning similar to that used for uniqueness invariants. If two predicates do not co-exist — in the sense that when one is added, the other is always deleted in equal number — and they are both otherwise fixed resources, the two predicates together can be considered a fixed resource. These conditions hold for the **at** and **in** predicates, for example. Neither one alone is a fixed resource, but the total number of true **at** and **in** predicates is fixed.

Even as TIM's invariants are, however, an automated version of CPlan can use them to discover a great deal of domain knowledge and apply that knowledge in its CSP encoding. The identity, uniqueness, and state membership invariants can improve the efficacy of CPlan's encoding; a new analysis of the type spaces can provide information for lower bound distance constraints; and fixed resource invariants can be used to create capacity constraints.

Together, identity, uniqueness, and state membership invariants can help improve the effectiveness of an automated CPlan encoding. These invariants capture exactly the information needed to decide to encode a predicate in a CSP as a variable/value pair rather than a large set of Boolean variables. For example, an identity invariant from Figure 3 states that a package in the logistics domain has at most one location. Therefore, rather than creating one Boolean variable for each instance of $\mathbf{at}(x, y)$ (over packages and locations), the encoding can have one variable for each package. That variable's value will be the package's location if it participates in an **at** predicate; otherwise, the variable will have a dummy value indicating that no **at** predicate is true for that package. A separate variable for each package would track the **in** predicate. Rewriting predicates in this manner can drastically reduce the size of the final encoding.

In general, if a predicate fragment $\mathbf{P}_k$ participates in an identity relationship (of size one), the k^{th} argument of $\mathbf{P}$ can be transformed into a variable with one value for each possible set of values of the other arguments. If the predicate is binary, the values will be each instance of the type induced on the other argument; otherwise, the values will be each instance of the cross-product of the types induced on all other arguments.

State membership invariants and uniqueness invariants can be used to refine the CSP encodings. For example, since $\mathbf{in}_1$ and $\mathbf{at}_1$ from Figure 3 are unique predicate fragments (with respect to each other), it is unnecessary to use two variables to represent them; a single variable can stand for both. If a packages variable's value comes from the **at** predicate, no **in** predicate is true for that package and vice versa. In general, a special dummy value would represent the case when none of the predicates is true. In this case, however, the state membership invariant for packages indicates that this dummy value is unnecessary. The invariant states that a package must be either **in** or **at** something; so, the dummy value will never be used.

In general, if two CSP variables represent predicates which are unique with respect to each other, they can be merged into a single variable with the combined domains of the original two variables (but only one dummy value). If the predicate fragments represented by a CSP variable represent a whole state membership group, the dummy value in the variable's domain can be eliminated.

It would even be possible to split a variable in order to merge it with two other variables with which it is unique but which are *not* unique with respect to each other. In order to accomplish this, automated CPlan would have to constrain the two variables' values to be equal whenever either of their values falls into the domain of the split variable.

While it is not clear that making all possible merges of variables would be profitable, it is likely that merging entire type spaces into a single variable is valuable when it is possible. As in the case of the $\mathbf{at}_1 + \mathbf{in}_1$ space from Figure 3, such a merge will produce a very compact representation of a type space. Thus, identity, uniqueness, and state membership invariants offer a powerful method for capturing regularities in the domain.

The structure of the state spaces can also be used to help automate CPlan. The state transitions in each space can be used to help define distance constraints. The transitions produce a lower bound for the number of steps to achieve some new property for an object from the current state. Because the transitions ignore preconditions and object interactions, the number of steps to a goal predicate in a TIM property space will be at most the minimum number of steps necessary to actually achieve that predicate. In fact, for its lower bound to achieve any particular goal predicate from a current state, TIM can use the maximum lower bound in any state space in which a predicate fragment of the goal predicate participates.

These lower bounds can be added at each step of the CSP encoding conditionally limiting a variable's value at succeeding steps based on its current value. In a state-based encoding, this amounts to actually reducing the domain of the variable instantiated at different steps. If the variable cannot achieve some value y if its current value is x in fewer than k steps, then the implement lower bound constraint would disallow the value y for the variable at the next k steps if its current value is x. In a causal encoding, the same effect can be accomplished by constraining steps that add the new value y not to occur until after k steps have passed.

3.1 Related Work

There are a variety of other reformulation and analysis systems which also act at the precompilation level. The DISCOPLAN system acts at the precompilation level and discovers some of the same invariants as TIM [Gerevini and Schubert, 1998]. Moreover, DISCOPLAN's method of discovering reachable tuples of predicate arguments might augment TIM's type inference mechanism and CPlan's exploitation of TIM.

For predicates with a small number of values for one argument, TIM could relax its single object centered view to include both the predicate name and the identity of that argument. This might help in discovering domain constraints; for example, in the logistics domain, TIM might deduce that trucks that are **at** Phoenix are actually of a different type than those that are **at** Atlanta. As a result, TIM would not need to instantiate any clauses describing movement between cities for trucks.

Also, TIM can limit the domain of the variables it creates because of identity invariants using DISCOPLAN. Rather than making the domain equal to the cross-product of the types of all the other arguments in a predicate, TIM can set the domain to just that set of argument values which is reachable according to DISCOPLAN.

Fox and Long describe a technique for symmetry detection based on TIM's type system which also applies to the precompilation phase [Fox and Long, 1999]. The symmetries detected by this technique can be broken by the symmetry-breaking techniques described in Section 5.

Finally, the PABLO abstraction system [Christensen, 1990] (described in the next section) and some other abstraction systems can produce a new, abstracted version of the planning domain during compilation. The abstract domains lack some unimportant details that clutter the original. These abstract domains can be fed back into the precompilation stage where they may reveal regularities to TIM that are masked in the original domain.

4 PABLO

PABLO is an abstraction system which focuses on ignoring "details" in planning domains in order to expend initial effort on their more complex facets [Christensen, 1990]. PABLO offers a new way to compile planning problems into CSPs by moving between abstraction levels rather than just by changing plan lengths. Moreover, the analysis PABLO performs to construct its abstraction hierarchy suggests a method for automating lower bounds on distance constraints.

Korf formalizes abstraction as the process of moving between different levels of detail (information quantity) and types of representation (information structure) [Korf, 1980]. This paper concentrates on systems which change the level of detail, particularly on abstraction systems in the style of the ABSTRIPS planner. ABSTRIPS moves through a hierarchy of abstraction spaces [Sacerdoti, 1974]. "Higher" levels in the hierarchy are less detailed; "lower" levels are more detailed. The lowest level is the original or "ground" planning domain.

An abstraction planner searches down through the abstraction hierarchy, creating partial solutions at high levels which are expanded to more and more complete solutions at lower levels. If a solution is found at the ground level, it is an actual solution plan. On the other hand, if at some level no solution can be found, the planner must backtrack up one or more levels in the hierarchy until it can change the partial plan that led to the dead end.

PABLO acts on a STRIPS problem specification; it relaxes the conditions under which all goal predicates — actual goals and action preconditions — hold. At abstraction level k, each predicate holds anytime there is a plan to make the predicate true in k steps or less which could be inserted at that point. Note, however, that these plans might disturb other predicates' truth values; so, PABLO's relaxation ignores some interactions between subgoals. The result of the relaxation process is that goals which appear not take much planning to establish are assumed to be true, and the planner concentrates on establishing those goals which require greater effort.

Formally, a predicate relaxed to level 0 P^0 is equal to the ground predicate P, and a predicate relaxed to level 1 P^1 is equal to the disjunction of the ground predicate and the preconditions of all actions which add the ground predicate. In general, $P^n = P^{n-1} \vee \bigvee_i^m Reg(\mathrm{Op}_i, P^{n-1})$ where $Reg(Op, P)$ is the regression of P through action Op — the set of all preconditions of the action if it adds P and *false* otherwise.

Using these formulas, PABLO can determine whether a relaxed predicate P at abstraction level k holds initially based on whether P^k holds in the initial conditions. PABLO reasons about whether a relaxed predicate holds at intermediate stages of a plan using modal truth as described in [Chapman, 1987] and [Kambhampati and Nau, 1994]. This reasoning is unnecessary in standard (non-lifted) plan encodings. It can be replaced in causal encodings by constraints describing the relaxed preconditions of an action and in state-space encodings by modified frame axioms which account for changes in predicate truth values based on their relaxed formulas.

PABLO relaxes all the goal predicates until only one is initially false, plans for this predicate using actions with relaxed preconditions, and then successively lowers the abstraction level, planning to resupport predicates which become untrue at lower relaxation levels and patch threats which occur in the replanning process.

In the terminology defined in [Allen *et al.*, 1991], PABLO is a *theorem increasing* abstraction system with the *upwards solution property.* That PABLO is *theorem increasing* means that PABLO's abstraction process can only increase the number of solutions to a problem. PABLO is theorem increasing because the relaxed

```
Action A              Action B
   add:P                 add:Q
   del:Q                 del:P
```

Figure 4: An example STRIPS domain. The one step ground plan $< A >$ makes predicate P true; similarly, the ground plan $< B >$ makes Q true. Therefore, P's and Q's relaxed predicates are both true at abstraction level 1. Also, the null plan satisfies the goal $P \wedge Q$ at abstraction level 1. However, no ground plan can ever make P and Q simultaneously true (unless both are true in the initial state).

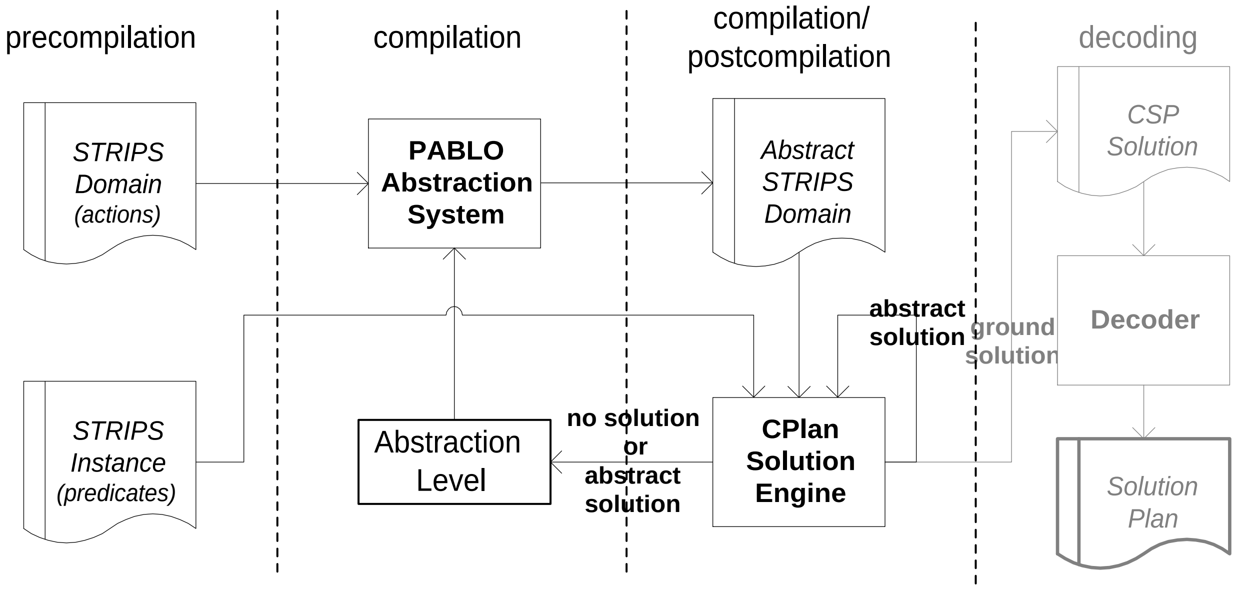


Figure 5: Planning as constraint satisfaction with abstraction. The box labeled **CPlan Solution Engine** represents the **Compiler** and **Solver** from Figure 1 including the plan length loop. Abstraction adds a second loop moving between abstraction levels. If an abstract solution is found, the abstraction level is lowered and the new level's encoding is changed to reflect the abstract solution. If no solution is found, the abstraction level is raised (and previous solutions from the raised level are disallowed). If a ground solution is found, it is decoded into a plan and returned.

predicates it creates are always true when a lower level (tighter) predicate is true. So, there will be at least as many plans at higher abstraction levels as there are at the ground level. Indeed, it is possible for a plan to appear to exist at a higher abstraction level when no such plan exists at the ground level as in Figure 4.

That PABLO has the *upwards solution property* means that if a solution plan exists at lower level in the abstraction hierarchy, a solution which can be refined to that lower level plan exists in each higher abstraction level. This property follows directly from the fact that PABLO's abstraction process is theorem increasing. The upwards solution property guarantees that it is *not* possible for a plan to exist at the ground level unless a corresponding plan exists at each higher level. With this property, PABLO can search from the top of the abstraction hierarchy down and guarantee completeness, reasoning in PABLO can always start in higher (and hopefully simpler) abstraction spaces before proceeding to lower spaces.

PABLO offers two techniques to assist in automating CPlan, the first acts on the compilation process and the second acts on the compiled representation. First, restructuring the standard compilation process used in CPlan (compiling to successively longer plan lengths) to work with an abstraction hierarchy offers a way to break up the usual compilation process into a series of more focused planning stages. Second, Christensen describes PABLO's relaxed predicates as providing "upper bounds" for the number of steps to achieve the strict predicate. These "upper bounds" are actually of use in developing lower bounds for CPlan's distance constraints.

PABLO, along with other ABSTRIPS-style abstraction systems, introduces a new way to sequence compilation of planning domains: compiling into successively more detailed abstraction spaces as shown in Figure 5. Within each space, it is still necessary to compile to successively longer plans[8], but much of the plan may already be fixed, committed at previous levels of the abstraction hierarchy.

The potential benefits of this style of compilation are great. Planning in the abstract levels can proceed quickly and produce quality partial plans. These plans feed back into the next lower level of abstraction, and lower compilation levels in the hierarchy can begin planning with the abstract plans (whereas currently planning begins from scratch at each new plan length).

Ideally, an abstraction system for CPlan would have the downward solution property [Allen *et al.*, 1991] and the ordered monotonicity property [Knoblock *et al.*, 1991]. Having the downward solution property means that if a solution is found in an abstract space, that solution can be refined to a solution in each lower level space. An abstraction system with this property would avoid backtracking through abstraction spaces; so, if a planning problem had no solution, that would be guaranteed to be found at the highest abstraction level. Having the ordered monotonicity property ensures that the refinements necessary to change a high level plan into a lower level plan will not change the truth value of predicates at already committed steps. In other words, the refinement process can proceed completely independently between each step of the high level plan. Such an abstraction system would allow the equivalent of divide-and-conquer planning.

Unfortunately, the PABLO system has neither the *downward* solution property nor the ordered monotonicity property[9]. The result is that in order to use the unaltered PABLO abstraction system, automated CPlan will need to be able to report unsolveability of planning problems and non-locally repair partially instantiated plans.

Because abstract plans in PABLO are not guaranteed to refine into successful lower level plans, the PABLO system may need to backtrack out of dead-end partial plans to higher abstraction levels. Unfortunately, planning as constraint satisfaction systems cannot report failure without the advantage of upper bound distance constraints. At best, CSP planners can report that no solution exists up to some fixed plan length (and local-search based systems cannot even reliably report this much [Kautz and Selman, 1996]). Therefore, in order to use a backtracking abstraction system, CPlan either needs to detect upper bounds on plan length or arbitrarily select such bounds.

On the other hand, CPlan *can* avoid backtracking altogether. In order to do this, it must avoid absolutely committing to high level plans at lower abstraction levels. Instead of committing to these plans, CPlan can use them as heuristics to speed solution.

In a depth-first search constraint solver such as the backtracking solver CPlan currently uses, the high level plan can be used to determine the initial variable and value choices in the search tree, directing search toward the subtree which previous levels indicate as most valuable; however, if that subtree proves to contain no solution, search can backtrack into regions of the space which alter the high level plan.

Adding dynamic backtracking techniques [Ginsberg and McAllester, 1994] to CPlan's constraint solver would allow the system to maintain as much of the partial plan as possible during backtracks. Dynamic backtracking allows a systematic search engine to change the value of the variables causing an inconsistency without altering unrelated values, even if the responsible variables are higher in the search tree than some of the unchanged variables.

In a local search constraint solver, the partial plan produced by a higher abstraction level can be used to define the initial neighborhood of the search. If the partial plan is useful, this will seed the search in the appropriate area; otherwise, the search can eventually work its way out of that neighborhood and find a solution elsewhere.

CPlan can also tolerate PABLO's lack of the ordered monotonicity property. The standard state-space encodings of planning as constraint satisfaction are inefficient for the non-local plan repair necessitated by this lack, but causal encodings can be quite effective for planning from a partially instantiated plan [Mali and Kambhampati, 1999]. The problem with state-space encodings stems from the fact that non-local repair requires examining an entire plan and inserting steps between existing steps. Unfortunately, state-space encodings require that steps be contiguously ordered; so, a partial plan must commit to both an ordering of steps and the number (though not the identity) of steps intervening between the committed steps. Mali and Kambhampati point out that causal encodings provide a solution to this problem. Although causal

[8]It seems unlikely that step-by-step compilation can be avoided. This compilation process is forced by the fact that the general planning problem is PSPACE-complete [Erol *et al.*, 1992]. In order to compile to the NP-complete problem of constraint satisfaction, a planning problem must be constrained somehow to become an NP-complete problem [Kautz *et al.*, 1996]. Restricting solution length is the only technique currently used to achieve this reduction and the only one of which the author is aware.

[9]There *are* practical abstraction systems that have the ordered monotonicity property. There are also practical systems with the downward solution property. Systems of both sorts will be discussed as related work in Section 4.1.

encodings are, in general, less efficient than state-space based encodings, causal encodings allow a partial plan to assert a non-contiguous ordering and allow unfettered insertion of new steps between existing ones.

Using causal encodings, CPlan can search using the partial plans produced at each level of PABLO's abstraction hierarchy. It can encode an appropriate length plan and then build the higher level partial plan into the encoding. However, CPlan will still need to compile to successive lengths at each level in order to determine how many extra steps are needed to fill out the partial plan.

PABLO's relaxed predicates can also be used to produce lower bound distance constraints. Although Christensen describes PABLO's relaxed predicates as providing upper bounds on subgoal plan lengths, for the purposes of CPlan, they actually provide lower bounds. The reason is that the CPlan upper bounds are global and firm: if an upper bound states that a package in Boston will be in Pittsburgh in at most k steps, it means that the plan must put the package in Pittsburgh before k steps pass. PABLO's relaxed predicates describe whether the planner could get the package to its destination in k steps or less. Therefore the lowest level relaxed predicate which is true in a given state provides a lower bound on the number of steps needed to make that predicate true from the given state.

The lower bounds can be implemented as described in Section 3. If a lower bound is activated, it would either restrict the possible values of its variable over the succeeding steps (in a state-space encoding) or apply a restriction on the number of steps before an action can assert a new value for the variable (in a causal encoding).

4.1 Related Work

There are a variety of other systems for creating and exploiting abstraction hierarchies [Sacerdoti, 1974; Knoblock, 1990; Allen *et al.*, 1991]. ALPINE in particular has the ordered monotonicity property [Knoblock, 1990]. So, ALPINE's lower level plans will not disturb any predicates at steps committed at higher abstraction levels. Although this property restricts the abstraction steps ALPINE can take, using a system with this property in CPlan would have a substantial benefit. With the ordered monotonicity property, CPlan could establish a partial plan and then treat each separate pair of contiguous steps in the partial plan as a subproblem. Since CPlan could plan from scratch for each subproblem, causal encodings would not be necessary and CPlan could use state-space encodings freely.

Tenenberg's abstraction system [Allen *et al.*, 1991] has the downward solution property. So, using this system, CPlan could fully commit to partial plans at each abstraction level. However, in order to achieve the downward solution property, Tenenberg's system accepts severe restrictions which limit the type of abstractions it can perform.

[Mali and Kambhampati, 1998] formulates HTN planning as constraint satisfaction. Work in case-based planning also suggests novel compilation methods for planning as constraint satisfaction systems [Hanks and Weld, 1995].

5 Symmetry-breaking Predicates

Crawford et al.'s symmetry-breaking predicates system [Crawford *et al.*, 1996] detects and exploits symmetries in Boolean satisfiability (SAT) problems, a restricted class of constraint satisfaction problems. Their system finds symmetric (interchangeable) variables in a problem and then inserts new constraints which remove the symmetries by invalidating all but one of the symmetric variable arrangements. The equivalent process in a CSP would be to find interchangeable variables (or values) and insert new constraints to disallow all but one of the rearrangements of the equivalent variables (or values). Symmetry detection can be used to find CPlan symmetry constraints and can also find some CPlan action choice constraints.

The CSP variant Crawford et al. study, Boolean satisfiability, is a particularly simple one. A Boolean satisfiability problem is a CSP in which the variables' domains are restricted to the two values *true* and *false*; each variable has a negated version which is constrained to have the opposite value of the variable. Constraints are sets of variables, either negated or non-negated, of which at least one must be *true*. Listed below is a small SAT problem from [Crawford *et al.*, 1996]. The variables are a, b, and c and their negations $\overline{a}$, $\overline{b}$, and $\overline{c}$. The "at-least-one" constraints are represented as sets; at least one variable in each set must have the value *true*.

$$\{a, \overline{c}\}, \{b, \overline{c}\}, \{a, b, c\}, \{\overline{a}, \overline{b}\}$$

There are two solutions to this problem: $\{a = \text{true}, b = \text{false}, c = \text{false}\}$ and $\{a = \text{false}, b = \text{true}, c = \text{false}\}$. In each solution, the value of $\overline{a}$ is the opposite of a's value and similarly for $b/\overline{b}$ and $c/\overline{c}$. Both solutions make one variable true in each constraint. Moreover, a and b are symmetric in this problem. Rewriting the problem by exchanging a for b and $\overline{a}$ for $\overline{b}$ shows the symmetry:

$$\{b, \overline{c}\}, \{a, \overline{c}\}, \{b, a, c\}, \{\overline{b}, \overline{a}\}$$

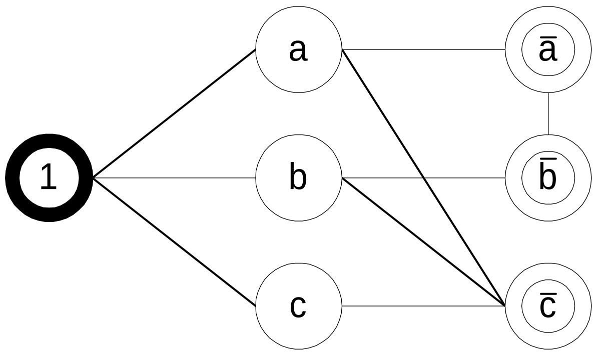


Figure 6: A graph representation of the SAT problem $\{a, \overline{c}\}, \{b, \overline{c}\}, \{a, b, c\}, \{\overline{a}, \overline{b}\}$. Each positive and negative variable is represented by a vertex. Corresponding positive and negative variables are linked. Constraints with more than two variables are also represented by a vertex; in this case, vertex 1 represents the only such constraint $\{a, b, c\}$. Constraints with two variables are represented by an edge between the variables' vertices. Each style of circle (single ring, double ring, and wide ring) represents a different graph color. Notice that swapping the labels of vertices a and b as well as vertices $\overline{a}$ and $\overline{b}$ produces an identical graph.

Although the order of the constraints and the order of the variables within the constraints has changed, this new problem nonetheless contains precisely the same constraints as the previous one. Therefore, a and b are symmetric variables. In general, a symmetry in SAT is a relabelling of the variables in a problem which leaves the set of constraints unaltered except for order.

Symmetries are detected by constructing a graph from the SAT problem such that each *automorphism* of the graph — a relabelling of the nodes in the graph which produces an equivalent graph — represents a symmetry in the SAT problem. While the problem of detecting automorphisms is not known to be easy (*i.e.*, not known to take only polynomial time), there are several packages for detecting automorphisms in graphs which are quite efficient.

Each variable in the SAT problem is represented in the symmetry finding system's graph, $\mathcal{G}$, by a vertex. Negated variables are given one color while non-negated variables receive a second color. Each pair of variable vertices is connected by an edge. Each constraint in the SAT problem is a vertex of a third color. The different colors ensure that constraint and variable vertices are not swapped in an automorphism since no automorphism is allowed to swap two variables of different colors. Initially, there is an edge between variable vertex x and constraint vertex y in $\mathcal{G}$ if and only if the variable x appears in the constraint y in the original SAT problem. For efficiency however, binary constraint vertices are removed, and the two vertices of the variables participating in the constraint are connected directly. In the resulting graph, any automorphism represents a symmetry in the SAT problem, and existing algorithms can be used to detect automorphisms. Finally, the relabellings of variable vertices in an automorphism can be translated directly into a relabelling of variables in the SAT problem. Figure 6 shows the graph for the example SAT problem described above.

Having detected a set of symmetries, Crawford et al.'s system eliminates them by breaking each symmetry. It accomplishes this by viewing solutions as numbers and insisting that only the numerically least of a set of symmetric solutions be investigated.

If the domain elements *false* and *true* had numerical values (0 and 1, respectively), an ordering of the variables in a problem along with the value assignments in a solution would define a number. For example, the variable ordering $< a, b, c >$ and solution $\{a = \text{true}, b = \text{false}, c = \text{false}\}$ from above defines the number 100.

Using this numerical interpretation of solutions, Crawford et al.'s system constructs an inequality constraint for each symmetry which breaks it. The inequality states that every acceptable solution value for the canonical ordering $< x_1, \ldots, x_n >$ is less than or equal to the solution value for the symmetrical reordering $< x_{s_1}, \ldots, x_{s_n} >$. For any two different symmetric solutions, this inequality will be true of only one of the solutions (the numerically least).

For example, the symmetry in the problem from Figure 6 can be described by reordering a canonical variable ordering $< a, b, c >$ to $< b, a, c >$ (in the first position, b replaces a; in the second position, a replaces b; in the third, c stays the same). The predicate breaking this symmetry states that acceptable

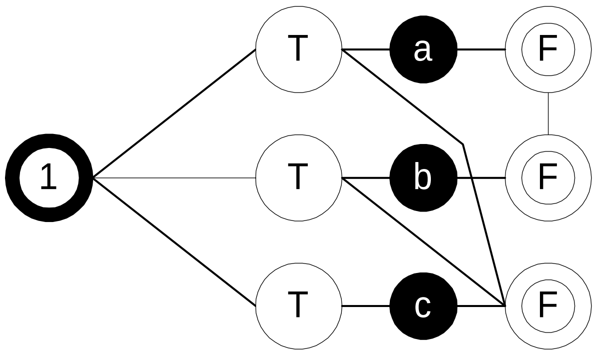


Figure 7: A graph representation of the SAT problem $\{a, \overline{c}\}, \{b, \overline{c}\}, \{a, b, c\}, \{\overline{a}, \overline{b}\}$, the same problem as in Figure 6. However, in this problem, the "exactly-one" constraint between each variable's values is explicitly represented by the small, black nodes.

solution numbers with digits ordered $< a, b, c >$ are less than or equal to solution numbers with digits ordered $< b, a, c >$. This inequality can be expressed by comparing the numbers digit by digit and insisting that at the first discrepancy, the number ordered by $< a, b, c >$ is less than the number ordered by $< b, a, c >$. In other words, the first digit of $< a, b, c >$ must be less than or equal to the first digit of $< b, a, c >$ ($a \leq b$), if the first digits are the same, the second digit of $< a, b, c >$ must be less than or equal to the second digit of $< b, a, c >$ (if $a = b$, then $b \leq a$), and so forth. Translated into SAT and optimized to remove vacuously consistent constraints, this predicate takes the form $\{\overline{a}, b\}$ ($a \leq b$ or, equivalently, $a \rightarrow b$). Of the two symmetric solutions mentioned above, $\{a = \text{true}, b = \text{false}, c = \text{false}\}$ (100) and $\{a = \text{false}, b = \text{true}, c = \text{false}\}$ (010), only the lesser of the two $\{a = \text{false}, b = \text{true}, c = \text{false}\}$ is consistent with the symmetry-breaking constraint $\{\overline{a}, b\}$.

With a few changes to the graph transformation, Crawford et al.'s symmetry detection techniques can be extended to deal with more general CSPs such as those used by CPlan. The key change is to represent explicitly the relationship which binds variables to their negations.

Crawford et al.'s graph form makes variables and their negations two different colors of vertices in which certain pairs (variables and their negations) bear a special relationship. This relationship can be thought of as a special constraint, one that ensures that *exactly one* of its variables has the value *true* rather than at least one as with normal SAT constraints. This new constraint type, "exactly-one" constraints, can be represented in the graph in the same way that at-least-one constraints are represented but with a new color for the constraint vertex[10]. The example from Figure 6 is rewritten in this style in Figure 7. In this new style of graph, the exactly-one constraint can be thought of as representing the variable and the positive and negated vertices as representing the values *true* and *false* for the variable. The constraint then states that the variable can take on exactly one of these values.

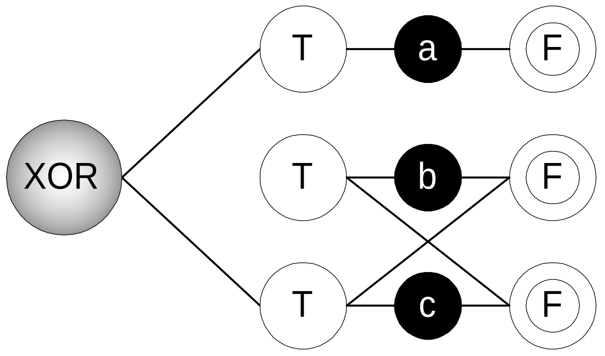
This explicit representation of the exactly-one constraint has an important advantage. There is no reason such a constraint need hold between only two values for a given variable. As many values as needed — each value with its own vertex color — can be connected to the constraint/variable vertex, and all will be forced to participate together in an exactly-one constraint in all automorphisms of the graph. This means that the graph reformulation can be used on variables with domains of any finite size rather than simply *true* or *false*.

In fact, the same technique of introducing a new color for new constraint types can be used for many different types of constraints[11]. As long as each of the new constraint types uses a distinct color, automorphisms of the graph will be forced to maintain the distinctions between constraint types. It is not even necessary to know how a constraint works to encode it in the graph. As long as each different type of constraint is encoded with a different color, automorphisms in the graph will represent symmetries in the CSP.

These typed constraints can be used to encode CPlan's implicit constraints (as long as the implicit constraints are grouped into distinct types). Unfortunately, the different types can mask symmetries that would be found with a CSP containing just a single constraint type. Figure 8 shows an example of this problem using SAT-style constraints along with an exclusive-or constraint.

[10]However, the optimizing technique of removing binary constraints can be used for at most one type of constraint.

[11]"Types" of constraints are defined in terms of functionality. If two constraints always allow the same set of value assignments when applied to a particular set of variables (or variable values), they are of the same type.



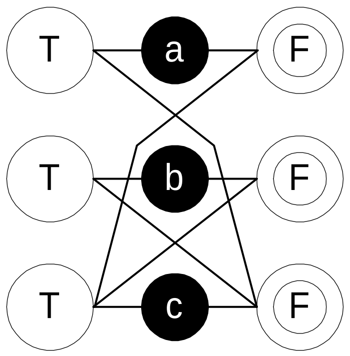


Figure 8: *Left*: a mixed constraint CSP in graphical form. *Right*: the SAT equivalent.

The left-hand graph represents the CSP $(a \text{ XOR } c), (b \text{ OR } \overline{c}), (\overline{b} \text{ OR } c)$. The two OR vertices have been optimized away and replaced with edges. The XOR constraint is represented with a new color node (the large node labelled XOR). Note that the graph has no nontrivial automorphisms. The right-hand graph is an equivalent CSP as a Boolean satisfiability problem, $(a \text{ OR } \overline{c}), (\overline{a} \text{ OR } c), (b \text{ OR } \overline{c}), (\overline{b} \text{ OR } c)$ or equivalently $\{a, \overline{c}\}, \{\overline{a}, c\}, \{b, \overline{c}\}, \{\overline{b}, c\}$. However, this graph has an automorphism which swaps a and b (and their values). The left graph hides an underlying symmetry.

Once a symmetry is detected in a CPlan-style CSP, the same technique used in Crawford et al.'s system can be used to break it. CSP solutions can be represented as numbers by ordering both the variables and the domains of each variable (keeping the orderings of identical domains consistent). The result is a number in which each digit may have a different base depending on the domain size of the variable it represents, but there is nonetheless a total ordering on the numbers. CSP solution numbers are compared from left to right and the first discrepancy defines their ordering, just as with the binary numbers resulting from SAT solutions. The symmetry-breaking predicates for CSPs also impose an ordering on symmetrical variable arrangements just as in the SAT problem.

Finally, by making all domain values in the graph representation of a CSP the same color, automorphisms which map values into other values can be discovered rather than just those that map variables into other variables. Because of the structure of the exactly-one vertex which connects each variable's values together, an automorphism which relabels a variable into some other variable must still relabel all the variable's values. However, two values can now be exchanged in a relabelling. This type of symmetry can capture symmetric variable values which would otherwise be disallowed by the color distinctions between objects. For example, two different trucks in the logistics domain with exactly the same start locations both of which are the value (location) of a package node are symmetric objects, but their symmetry appears in the graph as two value nodes which are interchangeable. If all value nodes have the same color, this exchange of value nodes will be allowed, and the symmetry detected. Such symmetries can be broken by considering each value of each variable to be a different Boolean "digit" in the solution. A digit is 1 if the variable it represents takes on the value it represents and 0 otherwise. At this point, the same sort of inequality predicates described above can be applied to the symmetries.

Symmetry detection and symmetry breaking help primarily to automate CPlan's symmetric object constraints. These constraints keep the system from investigating plans which differ only in the choice between a set of symmetric objects. Symmetry detection will find these symmetric objects, as either variables or values which participate in the same set of constraints, and symmetry breaking will force the planner to consider only one alternative at any point in a plan where both of the two symmetric objects can play the same role.

Symmetry breaking at the CSP level can also prevent more complex symmetries which are classified as symmetrical action choices in CPlan. These are alternate action sequences all of which have the same end result. Joslin and Roy describe an example of a logistics problem with an action choice symmetry which the symmetry breaking system can detect and remove from a planning CSP [Joslin and Roy, 1997]. The problem is from the logistics domain and has two packages, three cities, and one airplane. Initially, *package*1 is at *city*1, *package*2 is at *city*2, and *airplane*1 is at *city*3; all cities are connected. The goal is to have *package*1 reach *city*2 and *package*2 reach *city*1. Although there are two different action sequences which are reasonable to try (flying first to *city*1 or first to *city*2), the action sequences are symmetrical. If a plan cannot be constructed which first routes the airplane to *city*1, neither can any plan be constructed which first routes the airplane to *city*2. As described in [Joslin and Roy, 1997], this and similar action choice symmetries will show up as symmetries in the CSP and can be broken by the techniques described above.

5.1 Related Work

There are a variety of systems for detecting symmetries in CSPs. Joslin and Roy describe a system based on Crawford et al.'s symmetry-breaking system for analyzing CSPs produced from planning problems [Joslin and Roy, 1997]. Their system improves performance by analyzing only the initial conditions and goal description in the CSP. Freuder [Freuder, 1991] describes a system which discovers symmetric object constraints in the strict sense that CPlan describes them. That is, Freuder's technique discovers symmetries between values in a variable's domain. Wiegel and Faltings describe several CSP simplification techniques based on eliminating local symmetries [Wiegel and Faltings, 1997].

Some other systems also act at the level of constraint satisfaction problems to discover regularities in planning (and other) problems. Wiegel and Bliek describe a method of compiling CSPs to Boolean satisfiability and a number of reformulation techniques which can then be applied to the satisfiability problems [Wiegel and Bliek, 1998]. Most planning as satisfiability systems take advantage of the "unit propagation" and "pure literal constraints" from the DPLL algorithm [Davis *et al.*, 1962]. Brafman takes these further, using resolution techniques and higher level versions of unit propagation to perform greater simplification on Boolean satisfiability problems [Brafman, 1999]. Although these techniques apply to satisfiability, they can also be useful in more general CSPs. Either the CSPs can be compiled down to Boolean satisfiability problems — as in Wiegel and Bliek's work — or similar polynomial time simplification techniques can be used in the CSPs themselves. All of these techniques require explicit constraints.

Levy's abstraction by relevance reasoning system abstracts a knowledge base to fit a specific query or set of queries [Levy, 1994]. The system creates an abstraction with both the downward and upward solution properties and which is guaranteed to be at least as efficient, computationally, as the original knowledge base. Abstraction according to relevance might be applicable to a compiled planning problem with the action and axiom schemas acting as the knowledge base and the initial conditions and goal as the query.

Selman and Kautz's technique for approximating propositional logic theories with Horn theories [Selman and Kautz, 1991] is also, in a sense, an abstraction system which works at the encoded CSP level. It provides two levels of abstraction, one of which has the downward solution property and the other the upward solution property. Moreover, the abstracted problems have a form that is guaranteed to be easy to solve compared to the general problem of finding a model for a propositional logic theory. This technique provides quick but incomplete abstraction spaces that can be checked for satisfiability and inconsistency, respectively. Although this technique applies to propositional logic and not more general CSPs, there are similar variations in the tractability of different types of CSPs to those exhibited by certain types of logical theories (like Horn clauses). However, Selman and Kautz's technique requires explicit constraints, and it is not designed to produce partial solutions which can be refined into total solutions.

6 Related Work

Much of the work relating to the reformulation and analysis tools has been discussed in the Related Work sections of each system (Sections 3.1, 4.1, and 5.1). However, there are several systems which resemble CPlan. ILPPlan [Kautz and Walser, 1999] and Vossen et al.'s planning system [Vossen *et al.*, 1999] are CSP planners both of which use hand crafted encodings in Integer Linear Programming (ILP), their constraint satisfaction variant. Ernst et al. automatically construct core constraints, but then incorporate expert knowledge as extra clauses [Ernst *et al.*, 1997]. Kautz and Selman follow this work with a thorough analysis of hand encoded domain specific knowledge in planning as satisfiability systems [Kautz and Selman, 1998]. A careful analysis of the advantages, disadvantages, and logical foundations of the expert knowledge used in these systems would provide more domain specialization targets for automation.

The Heuristic Search Planner [Bonet and Geffner, 1999] (HSP) is not a planning as constraint satisfaction system. Rather, HSP plans using state-space search. However, it benefits from a (surprisingly simple) heuristic in its search. The heuristic somewhat resembles PABLO's relaxation formula and might be useful in defining a new type of abstraction hierarchy. It would also be interesting to see how domain specialization techniques can benefit HSP's state-space search.

LPSAT [Wolfman and Weld, 1999] is the only fully automated planning as constraint satisfaction system which handles metric values. Along with metric IPP [Koehler, 1998], upon which its compilation process is based, it might provide insight in how to exploit capacity constraints.

Long and Fox examine TIM's applicability in the specific domain of transportation planning [Long and Fox, 1999]. Their work is interesting in that it reverses the direction of this paper's analysis, taking a domain independent analysis tool and studying specializations of the tool to particular types of domains. Conversely, the results of their work supply new insight which can be used to improve domain independent analysis techniques.

Alongside the CPlan expert knowledge discussed in this paper were a few topics this paper did not address. The two most important in CPlan are selecting variables as hidden or visible and choosing weights for heuristic constraints. Kautz and Selman's work in state-based and action-based encodings suggests some plausible choices of hidden variables [Kautz *et al.*, 1996]. Machine learning methods can help with tuning weights for heuristic constraints. Dietterich provides an excellent overview of recent work in machine learning [Dietterich, 1997]. Machine learning techniques might also be helpful in paring down the huge number of potential heuristic constraints which automated techniques can provide to only the most useful.

7 Conclusions

The technology to automatically discover many of the invariants and heuristics used in CPlan exists. This paper describes methods to use and extend existing reformulation and analysis techniques in order to automate CPlan. The techniques are used to discover several of CPlan's most important heuristic constraints, to streamline CPlan's CSP encoding, and even to allow domain specialization of the planning as constraint satisfaction compilation process through abstraction techniques.

However, merging all of these techniques into a single, automated system whose effectiveness rivals the hand-coded CPlan methodology seems a daunting task. Beyond the difficulties of balancing resource allocation to various analyses and exploiting their results in a single encoding, simply using the full expressive power of constraint satisfaction poses a substantial challenge. Moreover, some of these reformulation systems have contradictory demands on the representational scheme used by CPlan. For example, PABLO demands highly compact implicit constraints to represent its relaxed predicates, but this type of constraint degrades the performance of the symmetry detection system.

Fortunately, it may not be necessary to match CPlan's power with a fully automated system immediately in order to produce a powerful new planning tool. The TIM system provides enough information to automatically construct efficiently encoded core constraints. Along with these core constraints, the other techniques described in this paper can be used to produce a fully automated CSP planning system with a substantial amount of domain specialization already built in. The resulting planning system can be improved further by hand addition of expert human knowledge which cannot yet be automated. Over time, just as the compilers community has slowly automated and finally made mundane the process of compilation, the AI community can automate these heuristics in an orderly fashion.

8 Future Directions

The primary future direction of this work is actually implementing an automated CPlan system. This process can be broken down into a series of steps, each of which will produce a more effective, completely automated CSP planner. The vital first step is to construct a system which can automatically transform STRIPS planning domains and instances into CSP core constraints and make effective use of variables and their domains. This will require choosing a core constraint style and automatically deciding which identity invariants to use in the process of streamlining the encoding.

Once a system exists for the core constraints, it can be extended without altering its internals to support a STRIPS level abstraction system (*i.e.*, an abstraction system which produces new, simplified, STRIPS planning domains). This step also poses representational and algorithmic challenges. The system can be used to explore the tradeoffs between different abstraction styles and encoding styles (*e.g.*, causal versus state-space). Finally, the various heuristic constraint construction techniques and manipulations of the compiled CSP can be applied to the system.

In the process of constructing and optimizing these intermediate systems, it will be necessary to address one aspect of these analysis and reformulation systems which was not carefully discussed in this paper: time and space requirements. Each of [Fox and Long, 1998], [Christensen, 1990], and [Crawford *et al.*, 1996] provides details relating to the asymptotic time and space requirements of the main techniques considered in this paper, but two issues remain for future research. First, expressing the results of these techniques in a CSP framework may allow more efficient representations or enforce less efficient representations. Asymptotic analyses of the space requirements of applying these techniques in an expressive CPlan-style CSP will resolve this issue. Second, regardless of the asymptotic performance of these techniques, a practical planning system could benefit from managing the amount of resources allocated to these analyses on a problem by problem basis. This issue can be studied by experimenting with manual or automatic adjustments to resource allocations in automated CPlan-style planners.

This paper also presents various extensions to the existing reformulation and analysis systems. The most extensive involve TIM: strengthening its type hierarchy, unifying its analysis of fixed resource invariants with the other invariant analyses, and relaxing its object centered approach by using reachability analyses [Gerevini and Schubert, 1998].

References

[Allen *et al.*, 1991] J. Allen, H. Kautz, R. Pelavin, and J. Tenenberg. *Reasoning about Plans*. Morgan Kaufmann, San Mateo, CA, 1991.

[Blum and Furst, 1995] A. Blum and M. Furst. Fast planning through planning graph analysis. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1636–1642. San Francisco, Calif.: Morgan Kaufmann, 1995.

[Bonet and Geffner, 1999] B. Bonet and H. Geffner. Planning as heuristic search: New results. In *Proceedings of the Fifth European Conference on Planning*, Sept 1999.

[Brafman, 1999] Ronen I. Brafman. Reachability, relevance, resolution and the planning as satisfiability approach. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, Stockholm, Sweden, Aug 1999. San Francisco, Calif.: Morgan Kaufmann.

[Chapman, 1987] D. Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32(3):333–377, 1987.

[Christensen, 1990] J. Christensen. A hierarchical planner that generates its own hierarchies. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 1004–1009, August 1990.

[Crawford *et al.*, 1996] James Crawford, Matthew Ginsberg, Eugene Luks, and Amtabha Roy. Symmetry-breaking predicates for search problems. In *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning*, pages 148–159. San Francisco, Calif.: Morgan Kaufmann, 1996.

[Davis *et al.*, 1962] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *C. ACM*, 5:394–397, 1962.

[Dietterich, 1997] T.G. Dietterich. Machine learning research: Four current directions. *AI Magazine*, 18(4):97–136, 1997.

[Ernst *et al.*, 1997] M. Ernst, T. Millstein, and D. Weld. Automatic SAT-compilation of planning problems. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 1169–1176. San Francisco, Calif.: Morgan Kaufmann, 1997.

[Erol *et al.*, 1992] K. Erol, D. Nau, and V. Subrahmanian. On the complexity of domain-independent planning. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, July 1992.

[Fikes and Nilsson, 1971] R. Fikes and N. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3/4):189–208, 1971.

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

[Fox and Long, 1999] Maria Fox and Derek Long. The detection and exploitation of symmetry in planning problems. Technical Report 1/99, Department of Computer Science, University of Durham, UK, 1999.

[Freuder, 1991] Eugene G. Freuder. Eliminating interchangeable values in constraint satisfaction problems. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 227–233. Menlo Park, Calif.: AAAI Press, July 1991.

[Gerevini and Schubert, 1998] A. Gerevini and L. Schubert. Inferring state constraints for domain-independent planning. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, pages 905–912, Madison, WI, July 1998. Menlo Park, Calif.: AAAI Press.

[Ginsberg and McAllester, 1994] Matthew L. Ginsberg and David A. McAllester. Gsat and dynamic backtracking. In *Proceedings of the Fourth International Conference on Principles of Knowledge Representation and Reasoning*. San Francisco, Calif.: Morgan Kaufmann, 1994.

[Hanks and Weld, 1995] Steven Hanks and Daniel S. Weld. A domain-independent algorithm for plan adaptation. *J. Artificial Intelligence Research*, pages 319–360, January 1995. Available at `ftp://ftp.cs.washington.edu/pub/-ai/`.

[Joslin and Roy, 1997] D. Joslin and A. Roy. Exploiting symmetry in lifted csps. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*. Menlo Park, Calif.: AAAI Press, 1997.

[Kambhampati and Nau, 1994] S. Kambhampati and D.S. Nau. On the nature of modal truth criteria in planning. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*. Menlo Park, Calif.: AAAI Press, July 1994.

[Kautz and Selman, 1996] H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 1194–1201. Menlo Park, Calif.: AAAI Press, 1996.

[Kautz and Selman, 1998] H. Kautz and B. Selman. The role of domain-specific knowledge in the planning as satisfiability framework. In *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems*, pages 181–189. Menlo Park, Calif.: AAAI Press, June 1998.

[Kautz and Walser, 1999] H. Kautz and J.P. Walser. State-space planning by integer optimization. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, Orlando, Florida, July 1999. Menlo Park, Calif.: AAAI Press.

[Kautz *et al.*, 1996] H. Kautz, D. McAllester, and B. Selman. Encoding plans in propositional logic. In *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning*, pages 374–384. San Francisco, Calif.: Morgan Kaufmann, 1996.

[Knoblock *et al.*, 1991] Craig Knoblock, Steve Minton, and Oren Etzioni. Integrating abstraction and explanation-based learning in prodigy. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 541–546, 1991.

[Knoblock, 1990] C. Knoblock. Learning abstraction hierarchies for problem solving. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 923–928. Menlo Park, Calif.: AAAI Press, August 1990.

[Koehler, 1998] J. Koehler. Planning under resource constraints. In *Proceedings of the Thirteenth European Conference on Artificial Intelligence*, pages 489–493. Chichester, UK: John Wiley & Sons, 1998.

[Korf, 1980] R. Korf. Toward a model of representation changes. *Artificial Intelligence*, 14:41–78, September 1980.

[Levy, 1994] Alon Y. Levy. Creating abstractions using relevance reasoning. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*. Menlo Park, Calif.: AAAI Press, July 1994.

[Long and Fox, 1999] D. Long and M. Fox. Automatic synthesis and use of generic types in planning. Technical Report 3/99, Durham University, UK, 1999.

[Mali and Kambhampati, 1998] A. D. Mali and S. Kambhampati. Encoding HTN planning in propositional logic. In *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems*, pages 190–198. Menlo Park, Calif.: AAAI Press, June 1998.

[Mali and Kambhampati, 1999] A. D. Mali and S. Kambhampati. On the utility of plan-space (causal) encodings. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence*. Menlo Park, Calif.: AAAI Press, July 1999.

[Meseguer, 1989] P. Meseguer. Constraint satisfaction problems: an overview. *AI Commun.*, pages 213–217, 1989.

[Penberthy and Weld, 1992] J.S. Penberthy and D. Weld. UCPOP: A sound, complete, partial order planner for ADL. In *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning*, pages 103–114, October 1992.

[Prosser, 1993] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computer Intelligence*, pages 268–299, 1993.

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

[Schubert, 1994] L. Schubert. Explanation closure, action closure, and the sandewall test suite for reasoning about change. *Journal of Logic and Computation*, 4:679–700, 1994.

[Selman and Kautz, 1991] B. Selman and H. Kautz. Knowledge compilation using horn approximations. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 904–909, 1991.

[van Beek and Chen, 1999] Peter van Beek and Xinguang Chen. Cplan: A constraint programming approach to planning. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, Orlando, Florida, July 1999. Menlo Park, Calif.: AAAI Press.

[Vossen *et al.*, 1999] T. Vossen, M. Ball, A. Lotem, and D. Nau. On the use of integer programming models in ai planning. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, Stockholm, Sweden, Aug 1999. San Francisco, Calif.: Morgan Kaufmann.

[Wiegel and Bliek, 1998] Rainer Wiegel and Christian Bliek. On reformulation of constraint satisfaction problems. In *Proceedings of the Thirteenth European Conference on Artificial Intelligence*. Chichester, UK: John Wiley & Sons, 1998.

[Wiegel and Faltings, 1997] Rainer Wiegel and Boi V. Faltings. Structuring techniques for constraint satisfaction problems. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, Nagoya, Japan, August 1997. San Francisco, Calif.: Morgan Kaufmann.

[Wolfman and Weld, 1999] S. Wolfman and D. Weld. The lpsat system and its application to resource planning. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, 1999. To Appear.