Constraint-Based Planning using the Process Specification Language

Stuart Aitken
Artificial Intelligence Applications Institute
Division of Informatics
University of Edinburgh
Edinburgh EH1 1HN, Scotland
Email: stuart@aiai.ed.ac.uk
Phone: +44 131 650 2738
Fax: +44 131 650 6513

May 2, 2001

Abstract

A full account of the semantics of constraint-based planning in the Cyc KBS is presented. The constraint representation is generic as it makes use of a process representation ontology, the Process Specification Language, which is being developed as a standard by NIST. We also describe a planning graph algorithm that makes use of these constraints. These techniques allow modern, efficient planning techniques to be embedded into a large-scale KBS architecture, thereby re-combining two technologies that have diverged in recent research programmes. The relation between plan and process representation is also explored.

keywords: knowledge representation techniques for planning; constraint reasoning for planning; process representation

1 Introduction

This paper presents a constraint-based approach to planning [4, 5] where constraints are represented in the standard process representation ontology of the Process Specification Language (PSL) [9]. The NIST-PSL ontology is embedded into the Cyc Upper-Ontology giving an explicit account of plan representation within a large-scale KBS; Cyc [6, 7]. For efficient reasoning the Graphplan [1, 3] approach is adopted and modified by a planning module external to Cyc.

In recent years the planning community has focussed on algorithms, at the expense of knowledge representation. This emphasis has led to many new results [11]. In contrast,

knowledge-based systems research has concentrated on large-scale systems and knowledge authoring in RKF (Rapid Knowledge Formation). But problems requiring deep reasoning such as planning have been avoided. An intermediate ground of standard plan and process ontology has developed [9, 10] but has not yet gained acceptance in the traditional AI planning research community - in part due to its abstract nature and the lack of any accompanying reasoning procedures. There are a number of examples of plan ontology reuse in KBS projects, but these typically address tasks such as plan critiquing rather than plan generation. This paper attempts to use a proposed standard process representation, PSL, for plan generation tasks. These tasks are too hard to be solved deductively by a KBS, and hence they require efficient algorithms such as those making use of planning graphs [1, 3]. We propose a simple modular system architecture to account for the different reasoning tasks, and use of PSL to express plan constraints in each module (i.e. KBS and planner). The constraints are expressed in an implementation-neutral way. In fact, we embed PSL into the Cyc Upper-Ontology to illustrate how this can be achieved.

In this paper we describe: the embedding of PSL into Cyc; the use of PSL to express constraints; and we explain the novel aspect of our approach to encoding the semantics of actions as constraints. We then describe a planning-graph planner. Finally, we consider the specification of a process whose subactivity structure is known and compare the resulting representation with the equivalent plan representation.

2 Embedding PSL in the Cyc Upper-Ontology

For the purposes of process and plan representation the proposed standard Core PSL theory and the activity and states extensions [9] have been implemented in CycL. The core theory defines activity types and activity occurrences as top-level collections, in CycL we add *PSL* as a prefix: *PSLActivity*, *PSLActivityOccurrence*. The predicate *occurrenceOf* relates activity occurrences to activity types. PSL also defines objects and time points as top-level collections. We introduce *PSLObject* and reuse the existing CycL constant *TimePoint*. The four top-level collections are mutually exclusive in PSL.

Important relations in the Core theory include: *isOccurringAt* which holds between an activity occurrence and the timepoints between or equal to its begin and end; *participatesIn* which relates an object to an activity occurrence at a timepoint; *existsAt* which relates an object to a timepoint; *before*, *beforeEq*,*betweenInTime*, and *betweenEqInTime* are used to order timepoints; *beginOf* and *endOf* are functions returning a timepoint for a TemporalThing (an activity or an object); *infMinus* and *infPlus* are time points.

The activity occurrence extension of PSL adds a number of predicates representing order and containment between activity occurrences and activities. *PSLPrimitiveActivity* is a subset of activity containing activity types which have no *subactivity*. *subactivity* relates a subactivity to its super-activity. *occurrenceContains*, *occurrenceEarlier*, *occurrenceOverlap*, *successor*, and *subactivityOccurrence* plus the associated axioms provide the vocabulary for activity occurrence ordering.

The states extension of PSL provides the concepts for relating states of the world to activity occurrences. *Fluent* is the collection used for state descriptions, i.e. properties

of the world that can change as a result of an activity occurrence. We define fluent as a collection of *CycFormula*.

state and *postState* relate fluents to activity occurrences. Therefore, the states prior to and following activity occurrences can be specified.

A number of types of activity are identified and new sub-collections of activity and fluent are introduced: *AchievementActivity*, *RepairableFluent*, *NonrepairableFluent*, *ReversibleFluent*, and *IrreversibleFluent*. Predicates which relate activity occurrences to fluents include: *changes*, *possiblyChanges*, *achieved*, *falsified*, *fluentInterval*, *negFluentInterval*, These relate activities to fluents: *preconditionFluent*, *negPreconditionFluent*; and to timepoints: *possibleFluent*, *requiredFluent*, *temporalFluentInterval*.

3 Constraints for Planning in PSL

The *state* and *postState* predicates are used to represent preconditions and postconditions of activity occurrences. They could be used to express deductive planning rules, for example: the rule that *act* has precondition *p* and when performed produces state *q* can be expressed in the situation calculus as:
1. $(holds(p, s) \wedge state(p, act) \wedge postState(q, act)) \rightarrow\ holds(q, do(act, s))$.
In this formulation, actions are related to fluents, not to a state index (or time point), as is sometimes proposed. A KBS could perform planning using such rules but reasoning would be intractable without sophisticated heuristics for guiding search. Similarly, STRIPS representations of actions do not take a state index or time point as a parameter. However, in the STRIPS approach the action description may contain the state modification operators **add** and **delete**. If such a representation were to be used for actions in a KBS then the corresponding theory of state modification would also have to be formulated. We show that this is not necessary.

As stated above, it is our aim to fully characterise the semantics of actions in the ontology and the KBS, and to find a common semantics for ontology/KBS and planner. Therefore we can either build the theory of STRIPS operators into the ontology/KBS or express the full set of action constraints in the action definition itself. We explore the latter.

The *planning as satisfiability* view of action specification of Kautz and Selman allows this possibility when we require that the representation of actions in the KBS fully specifies all changes to all fluents that may be modified. This is the direct approach to constraint representation mentioned in [4]. Activity occurrences must be explicitly parameterised, and the scope must include all variables which change through the occurrence of the activity. Activity specifications have the following schematic form (expressed in standard predicate calculus syntax):
2. $\forall p_1 ... \forall p_n \exists ao\ occurrenceOf(ao, ActivityType) \wedge\ state(fluent_1(p_1, ..., p_n), ao) \wedge\ ...$
$postState(fluent_2(p_1, ..., p_n), ao)\ \wedge\ ...$
There will typically be a conjunction of *state* and a conjunction of *postState* terms in the formula. Fluents $fluent_i$ are atomic formula, that is, they do not contain connectives or quantifiers.

Situation calculus rules (1. above) can express constraints by turning implications into equivalences. This is necessary as the constraint solver does not operate purely deductively. However, if we are not developing the world state model deductively in the KBS, the *holds* predicates are redundant. Therefore, the proposed PSL constraint formulation only contains the conjunction of *state* and *postState* predicates of the original equivalence, and each conjunct of the expression is true unconditionally. The pattern of quantification is modified appropriately to produce the schematic formulation given in 2.

There is an explicit commitment to the existence of all possible $|D_1| * ... * |D_n|$ actions (where D_i is the domain of parameter i)[1] The model contains all state/postState relations for all possible values of ao in the deductive formulation. Aside from the names of constants used to identify states, the logical model of a plan is the same for equivalent constraints expressed as a conjunction or as an logical equivalence. Matching the fluent in *state* to a specific world state determines the parameters of the activity occurrence (i.e. the parameters of the skolem function), hence the *postState* for that occurrence can be constructed, and vice versa. In the deductive rule the *state* and *postState* would share variables and the same patterns of instantiation would be permitted. The advantage of the conjunctive formulation is that it requires actions to be defined as a complete set of constraints on the plan. The mapping to a graph representation is thereby simplified as there is no other source of plan generation knowledge - explicit or implicit in the action representation.

The example in Figure 1 shows the encoding of step 1 of the citric acid cycle, a process taken from a cellular biology text. The structure of the process is well known, but we formulate it as a planning problem in order to illustrate the dual planning/process representation we obtain through PSL. The *?Event* is an occurrence of a *Hydrolysis* action and is parameterised by the number of Oxaloacetate, AcetylCoA, HSCoA, and Citrate molecules. The amount of Oxaloacetate and AcetylCoA is decreased by one molecule, while one molecule of HSCoA and Citrate is produced. Note that the *numberOfMoleculesInCell* predicate is used for illustrative purposes and is not presented as part of an authoritative theory of cell biology. The expression is presented in CycL syntax, with the simplification of using + and - for the arithmetic operators.

The constraints discussed so far have been restricted to constraints on action preconditions/effects. Satisfying these ensures valid plans (assuming that the planner correctly implements syntactic matching of fluents). Other important classes of constraints include constraints based on the semantics of the predicates used as fluents, and action ordering constraints. These constraints may ease the authoring of constraint specifications or may direct plan generation respectively.

3.1 Ontology constraints

Constraints such as the restriction in the blocks world that the predicate on is irreflexive, $\neg on(x, x)$, may be specified in the ontology. Including an inequality $x \neq y$ when $on(x, y)$ as a pre/postcondition in the action specification is a simple way to enforce this restriction

[1]Naturally, in the skolemized form, ao is represented by a skolem function parameterised over $p_1, ..., p_n$, and no other constants need be created.

```
(forAll ?W (forAll ?X (forAll ?Y (forAll ?Z
   (thereExists ?Event
   (and
     (occurrenceOf   ?Event Hydrolysis)
     (state
       (numberOfMoleculesInCell Oxaloacetate ?W)
       ?Event)
     (postState
       (numberOfMoleculesInCell Oxaloacetate ?W-1)
       ?Event)
     (state
       (numberOfMoleculesInCell AcetylCoA ?X)
       ?Event)
     (postState
       (numberOfMoleculesInCell AcetylCoA ?X-1)
       ?Event)
     (state
       (numberOfMoleculesInCell HSCoA ?Y)
       ?Event)
     (postState
       (numberOfMoleculesInCell HSCoA ?Y+1)
       ?Event)
     (state
       (numberOfMoleculesInCell Citrate ?Z)
       ?Event)
     (postState
       (numberOfMoleculesInCell Citrate ?Z+1)
       ?Event)))))).
```

Figure 1: A constraint expression for step 1 in the citric acid cycle

in the planner also[2]. However, it is more convenient to express these constraints once only, in a more general form. Equality/inequality constraints are formalised as follows:
3. $\forall x \forall y \forall a\ postState(P(x,y),a) \rightarrow postState(x \neq y, a)$
The standard interpretation of $=$ must now be assumed to be implemented in the planner. Equality and inequality constraints are common, and can be more concisely expressed as a relation between a conjunction of fluents and a conjunction of equality/inequalities:
4. $(\forall f_1 ... f_n \forall i_1 ... i_n\ stateConstraint(f_1 \wedge ... f_n, i_1 \wedge ... i_n) \leftrightarrow$
$(\forall a\ postState(f_1,a) \wedge ... postState(f_n,a) \rightarrow\ postState(i_1,a) \wedge ... postState(i_n,a))$
where $f_1 ... f_n$ are fluents and $i_1 ... i_n$ are in the set $\{=\neq\}$. The *stateConstraint* predicate is introduced for convenience, i.e. it is a purely definitional extension of PSL. In the citric acid example, the fact that the predicate for the number of molecules in a cell of a given type is functional in the second argument requires an equality constraint which can now be expressed concisely:

```
(stateConstraint (and (numberOfMoleculesInCell A,B)
                      (numberOfMoleculesInCell A,C)) B=C))
```

[2]This is a simple way to make the planner aware of ontology constraints.

3.2 Action ordering constraints

The *occurrenceOf* predicate is used in the activity specification to relate occurrences to activity type. However, activity type information plays no role in action sequencing as all that matters is the $state \leftrightarrow postState$ interdependency which is encoded via the skolem function. Activity type can be used to express action ordering constraints. For example we can state that Hydrolysis should precede Oxidation in the plan we are creating:

```
(implies (and (occurrenceOf ?X Hydrolysis)
              (occurrenceOf ?Y Oxidation)
              (subactivityOccurrence ?X plan)
              (subactivityOccurrence ?Y plan))
         (occurrenceEarlier ?X ?Y))
```

Unlike ontology constraints, action ordering constraints cannot be expressed using *state, postState* predicates as these constraints are on the plan not the world state. This type of constraint could be used to determine acceptable plans in the solution extraction phase of graph-based planning. This is possible as solutions can be easily expressed in PSL. However, we have not yet implemented this feature.

In general, there is a great potential for using ontology constraints such as symmetry, reflexivity and range types in the planner. We shall explore this further in future work.

4 Graph-Based Planning with Constraints

The direct encoding of plan constraints in PSL within a CycL-based KBS has been described. We now outline the plan graph approach to plan synthesis. The approach implements the additional equality constraints described above. The planner is implemented as a module which communicates with the Cyc KBS, but is external to it. The planner simply extracts *state* and *postState* assertions from the KB prior to planning. The plan graph is constructed directly from these assertions without any reformulation as is now described.

The algorithm for solving Dynamic Constraint Satisfaction Problems (DCSP) was first applied to configuration tasks. Solution components are represented by variables which range over a specified domain. Variables may be assigned values from the domain and may be *in* or *out* of the solution. The DCSP algorithm [8] makes use of ATMS to keep track of dependencies between propositions. DCSP can be used in plan synthesis after a plan graph is constructed. Propositions in the plan graph are represented as DCSP variables and their domain is the set of actions that cause the proposition to be true. Variables must be indexed by plan graph level, and boolean consistency must be imposed by ARN rules (always require not) in the DCSP problem.

In our approach the plan graph is created directly from the action constraints as follows.

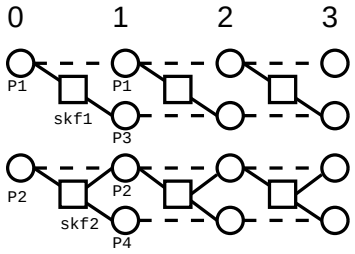
4.1 Plan graph and DCSP

The planning graph is generated forwards from assumptions. Level 0 contains the ground propositions that are true in the initial state of the world. As described in [3], the planning

PSL Constraints

```
state(P1,skf1)
postState(P3,skf1)
state(P2,skf2)
postState(P2,skf2)
postState(P4,skf2)
```

Plan Graph



DCSP

```
domain(<P2,1>,{nop,skf2})
<in(P1),0>
<in(P2),0>
<in(P2),1>
<eq(P2,skf2),1>
<in(P4),1>
<eq(P4,skf2),1>
```

Figure 2: Constraints, planning graph and DCSP

graph differs from the plan states of a state-space planner as the graph contains all proposition true in any state, but does not directly encode state consistency. Figure 2 illustrates the approach. In contrast with [1], the graph has no delete edges as there is no delete construct in the action representation.

A simple procedure is used to construct the *level i* graph from the *level i-1* graph:
1. Find all actions whose preconditions are true at level i-1. (Use state(P,A) and the graph at level i-1 to determine potential Actions, then check that all preconditions are true.)
2. Generate all propositions at level i from the valid actions found in 1. (Use postState(P,A) for all A identified.)

While the state/postState predicates may contain variables, the plan graph at level $i-1$ is always composed of ground propositions. In valid actions, all preconditions match ground states, therefore all variables are bound, and propositions in the level i graph are also ground. The procedure for matching state conditions against the propositions in the graph is simple syntactic pattern matching.

The plan graph is translated into a DCSP problem as follows:
1. The domain of a variable (proposition) is the set of actions that can create it.
2. RV rules are constructed for all propositions at level i-1 on which a proposition at level i depends.
3. RV rules and CON rules are constructed for all propositions at level i which must be true at level i given an assignment at level i.
4. ARN rules are constructed for all variables (propositions) P and not-P at level i.

The dependency between propositions (variables) at successive levels is encoded by the RV rules in 2. To ensure that the assignment of an action to a proposition entails all

```
(occurrenceOf  plan1          PlannedActivity)
(occurrenceOf  (skf1 1 1 0 0) Hydrolysis)
(occurrenceOf  (skf3 1 0 1)   Isomerization)
(occurrenceOf  (skf5 1 0 0)   Oxidation)
(subactivityOccurrence (skf1 1 1 0 0) plan1)
(subactivityOccurrence (skf3 1 0 1)   plan1)
(subactivityOccurrence (skf5 1 0 0)   plan1)
(successor (skf1 1 1 0 0) (skf3 1 0 1))
(successor (skf3 1 0 1)   (skf5 1 0 0))
```

Figure 3: Representation of a plan in PSL

other post states of that action are in and take that value too, RV and CON rules (3) are generated. To ensure boolean consistency between solutions ARN rules in 4. are required. Note that the names of DCSP variables have no semantics in themselves.

Figure 2 shows the plan graph created from the PSL constraints for two actions *skf1* and *skf2*. Note that the arguments of predicates (*numberOfMoleculesInCell* in the citric acid example) and functions are not shown to simplify the diagram. Two new ground predicates, or propositions, are created at level 1: *P3* and *P4*. Assuming that *P2* is *in* at level 1, it can be assigned *skf2* from its domain. As a consequence, an RV rule asserts that *P4* must be *in* and also take the value *A2*. This is necessary as the action has two postconditions as illustrated. A further consequence is that *P2* is *in* at level 0. In fact all level 0 propositions are *in* as shown. The assignment of *skf2* to *P2* does not explicitly depend on the *in(P2)* assumption as the need for variables to be *in* to be relevant to the solution is built into the solution extraction procedure.

4.2 Constraints

Constraints on predicates such as irreflexivity and having functional arguments are expressed as equality constraints and are obtained from *stateConstraint* assertions in the Cyc KB. If a postState of an action violates a constraint, the action and its postStates are not added to the graph. Constraints with multiple fluents as conditions (which could be supported by different actions) and are tested against all propositions in the graph at every level, and ARN rules are generated between propositions as a result.

4.3 Solution extraction and representation

Solution extraction is attempted when there is an assignment to the goal proposition at the current level of graph expansion. This process is initiated by first asserting that the goal proposition is *in* at the current level. Assignments are found for all variables which are *in*, based on the same assumptions as the goal assignment. New assumptions may be found and new *in* variables identified and assigned values as a result. This procedure is repeated from the current graph level to level 0. A consistent set of assumptions must be maintained. If this procedure can be completed, the solution is the sequence of (sets of) actions that support propositions at levels *1 - n*.

The result of planning—the action sequence or plan—can be expressed in the PSL ontology and therefore can simply be asserted in the KBS. There are two possibilities for plan representation: the assignment of activity occurrences to ordered time points using *isOccurringAt* and *before*, and/or the sequencing of activity occurrences using *successor* and *subactivityOccurrence* (activities can be viewed as subactivities of the overall plan activity occurrence). As all activity occurrences are implicitly represented in the KBS no new constants need be created to represent the planned activities (the symbols skf1, skf3 etc are skolem terms which are already defined in the KB). Constants for the overall plan activity and for timepoints may need to be created. The order of activity occurrence is easily extracted from the variable assignments at each level of the DCSP found in solution extraction. The PSL encoding follows directly from this ordering. For example, a three step plan consisting of `skf1`, `skf3`, `skf5`, at levels 1, 2, and 3, would be represented in PSL as shown in Figure 3. In the case of the citric acid cycle example presented above, the *successor* and *subactivityOccurrence* model exactly corresponds to the specification of the process model of the citric acid cycle.

Currently, we have implemented a number of simple domains including the blocks world and citric acid cycle, and successfully tested the planner in these domains. The runtime performance of the current Prolog implementation has not been systematically evaluated as it intended to be a testbed rather than a competitive implementation. Indeed, efficient C implementations already exist for the basic algorithms and these could be modified to improve run times.

The procedures for generating the DCSP problem are simple. In future work we shall further investigate any additional complexity they may introduce into the DCSP through the generation of additional rules. As a preliminary exploration of the effect of constraints on problem size, the following table shows the effect of deleting the irreflexivity constraint on *on* in the blocks world domain. Both the number of nodes in the graph and the number DCSP rules generated on each iteration of the algorithm increases:

Iteration	1	2	3
All Constraints			
Graph Size	18	25	30
No. DCSP Rules	276	794	1241
Without $\neg on(x,x)$			
Graph Size	20	29	34
No. DCSP Rules	381	1160	1609

It can be seen that the small increase in graph nodes (e.g. 2 instances of *on(x,x)* at level 1) results in a disproportionate increase in the number of DCSP rules. If this result generalises, then ontology constraints appear to have a significant impact on the size of the problem representation (nodes and rules). However, the irreflexivity constraint may be particularly critical in this domain.

```
(implies (occurrenceOf ?P CitricAcidCycle)
     (thereExists ?A1 (thereExists ?A2 (thereExists ?A3
        (and (occurrenceOf ?A1 Hydrolysis)
             (occurrenceOf ?A2 Isomerization)
             (occurrenceOf ?A3 Oxidation)
             (subactivityOccurrence ?A1 ?P)
             (subactivityOccurrence ?A2 ?P)
             (subactivityOccurrence ?A3 ?P)
             (successor ?A1 ?A2)
             (successor ?A2 ?A3)))))))
```

Figure 4: Specification of the citric acid cycle (steps 1-3)

5 Process Representation

The citric acid cycle example was formulated as a planning problem in order to show how sequences of actions can be constructed from constraint formula in PSL. In this case, the prototypical model of the process is in fact known - the problem in process representation is to describe the process structure, the participants in the activities and subactivities, and the effects of activities. Clearly, there should be commonality between the process and plan representations, and the aim of this section is to demonstrate this.

The event structure of the citric acid cycle is specified in Figure 4 by a rule which asserts that all occurrences of the cycle have subactivity occurrences of the specified types, occurring in the specified order. The cycle has 9 steps but only three are shown for simplicity as the complete rule has the same form as the version shown. The rule is a specification, and does not introduce any instances of activity until the conditions are satisfied. The full specification of the process contains additional assertions using CycL vocabulary, but these are not directly relevant here. The pre and postconditions of the first subactivity occurrence is as shown in Figure 5.

The assumptions of this rule determine the participants in each subactivity occurrence. In this case, the number of Oxaloacetate molecules is decreased from the initial value after `?A1`. The changes to the world made by a subactivity occurrence are found by examining the (ground) state and postState assertions of that occurrence.

From sets of rules of these types, hypothesising:
`(occurrenceOf P1 CitricAcidCycle)` and by making the necessary assumptions about the context in which the process is executed, an instance of the complete process can be derived by a sequence of inferences in the KBS. The description is in terms of *occurrenceOf, subactivityOccurrence, successor, state* and *postState* as was the case for the instance level model created by the planner. While activity occurrences are represented by different skolem functions in the plan-derived model compared with the process model, the models are structurally identical.

It is interesting to compare the action ordering statement of section 3.2 with the process specification: The ordering statement is a weaker specification of the process. Thus there is a spectrum from no action ordering requirements, through partial ordering, to complete specification. The reasoning tasks range from plan generation to plan/process

```
(implies
 (and
   (occurrenceOf ?P CitricAcidCycle)
   (occurrenceOf ?A1 Hydrolysis)
   (occurrenceOf ?A2 Isomerization)
   (occurrenceOf ?A3 Oxidation)
   (subactivityOccurrence ?A1 ?P)
   (subactivityOccurrence ?A2 ?P)
   (subactivityOccurrence ?A3 ?P)
   (successor ?A1 ?A2)
   (successor ?A2 ?A3)
   (numberOfMoleculesInCell Oxaloacetate ?W)
   (numberOfMoleculesInCell AcetylCoA ?X)
   (numberOfMoleculesInCell HSCoA ?Y)
   (numberOfMoleculesInCell Citrate ?Z) ...)
 (and
   (state (numberOfMoleculesInCell Oxaloacetate ?W) ?A1)
   (state (numberOfMoleculesInCell AcetylCoA ?X) ?A1)
   (state (numberOfMoleculesInCell HSCoA ?Y) ?A1)
   (state (numberOfMoleculesInCell Citrate ?Z) ?A1)
   (poststate (numberOfMoleculesInCell Oxaloacetate ?W-1) ?A1)
   (poststate (numberOfMoleculesInCell AcetylCoA ?X-1) ?A1)
   (postState (numberOfMoleculesInCell HSCoA ?Y+1) ?A1)
   (postState (numberOfMoleculesInCell Citrate ?Z+1) ?A1) ...))
```

Figure 5: Effects of activity occurrences

checking. Hierarchical planning is between these extremes due to the extensive action ordering knowledge encoded in the plan schemas.

6 Conclusions

In the domains we are considering, we may have a model of a process and wish to represent and reason about that. We may also wish to perform reasoning about possible action sequences - planning. Therefore a underlying representation of actions that can be used for both purposes is an important requirement.

We have used a proposed standard representation for processes, PSL, and shown how it can be used for a direct encoding of action constraints for plan synthesis. We have adapted the planning graph and the related DCSP encoding to implement a planner which operates on this representation. The results of planning can also be encoded in PSL and can therefore simply be asserted in a KBS which conforms to the PSL ontology, or to which the PSL ontology can be mapped. The approach we describe is a novel combination of a proposed standard process/plan ontology representation with efficient graph-based planning techniques. Future work will be to investigate how expressions in the ontology can be used efficiently as constraints in the planner. This will ease the problem of writing constraints, and give insights into how ontology constraints can be used to guide graph/DCSP generation.

Acknowledgements

This work is sponsored by the Defense Advanced Research Projects Agency (DARPA) under subcontract 00-C-0160-01 with Cycorp on BAA99-35. The U.S. Government is authorised to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation hereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing official policies or endorsements, either express or implied, of DARPA, Rome Laboratory or the U.S. Government.

References

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

[2] M. D. Ernst, T. D. Millstein, and D. S. Weld. Automatic SAT compilation of planning problems. In *Proceedings of IJCAI '97*, 1997.

[3] S. Kambhampati, E. Parker, and E. Lambrecht. Understanding and extending graphplan. In *Proceedings of the 4th European Conference on Planning*, 1997.

[4] H. Kautz and B. Selman. Planning as satisfiability. In *Proceedings of ECAI '92*, 1992.

[5] H. Kautz and B. Selman. Unifying SAT-based and Graph-based planning. In *Proceedings of*, 1999.

[6] D.B. Lenat. Leveraging cyc for hpkb intermediate-level knowledge and efficient reasoning, 1997. URL: `http://www.cyc.com/hpkb/proposal-summary-hpkb.html`.

[7] D.B. Lenat and R.V. Guha. *Building large knowledge-based systems. Representation and inference in the Cyc project*. Addison-Wesley, 1990.

[8] S. Mittal and B. Falkenhainer. Dynamic constraint satisfaction problems. In *Proceedings of AAAI '90*, 1990.

[9] C. Schlenoff. The process specification language (psl) overview and version 1.0 specification, 2000. NIST Internal Report (NISTIR) 6459, http://www.mel.nist.gov/psl/.

[10] A. Tate. Roots of spar - shared planning and activity representation. In *Engineering Review, Vol 13(1), pp. 121-128*, 1998.

[11] D. Weld. Recent advances in ai planning, 1998. Technical Report UW-CSE-98-10-01, University of Washington, and in AI Magazine.