Efficient Planning with State Trajectory Constraints

Stefan Edelkamp

Baroper Straße 301
University of Dortmund
eMail: stefan.edelkamp@cs.uni-dortmund.de

Abstract. This paper introduces a general planning approach with state trajectory constraints, a language extension that has recently been introduced to PDDL for running the Fifth International Planning Competition in 2006. In our approach PDDL inputs with state trajectory constraints are translated into ordinary (fully instantiated) PDDL. It exploits the automata-based interpretation of the constraints, and refers to planning via explicit-state model checking. We explain how trajectory constraints are translated into LTL and non-deterministic Büchi automata and how these automata are compiled back to ordinary PDDL predicates and actions. We provide promising initial experimental results with heuristic search planning.

1 Introduction

State trajectory constraints [8] are one of two rather disjoint language features proposed for describing the inputs for the next International Planning Competition. They provide a important step of the agreed fragment of PDDL toward the description of *temporal control knowledge* [1, 13] and to *temporally extended goals* [15, 16]. State trajectory constraints assert conditions that must be met during the execution of a plan and are expressed through temporal model operators with bounded quantification over domain objects. Through the decomposition of a temporal/parallel plans into happenings, plan constraints feature also higher levels of PDDL, namely metric and temporal planning [6].

Temporally extended specifications are frequently met in *model checking* [3], where a property specification given in some temporal formalism is checked against the model. In *explicit state model checking* approaches properties are often specified in *linear temporal logic* (LTL). Here, the problem is to detect whether the model satisfies the specification on all (possibly infinite) transition sequences (runs) in the system. A counterexample is a run, which *falsifies* the property. In contrast, in action planning, a plan has to *satisfy* the constraints that are denoted in the goal. While there is continuously rising success in the guided falsification of models [5], elaborated exploration heuristics that exploit general temporal properties for improved error detection are rare. On the other hand, in AI planning, improved heuristics [11, 9, 4] have shown a dramatic impact on enhanced solving the reachability problem to establish a plan. Unfortunately, the success has not yet been enlarged to temporal annotations of goals. Existing planners for search control knowledge like TAL- and TLPlan [14, 1] and for temporally extended goals like MBP [15, 16] do not incorporate guided search.

In expressing state trajectory constraints and their semantics the authors in [8] already refer to LTL. Their extensions do not allow nested applications of the operators and thus take only a fraction of LTL into account. Nonetheless, all formulas can be transformed to an equivalent Büchi automaton. With some respect, the operators come quite close to the control knowledge in planners like TL- and TALPlan, where formula progression and bounded object quantification is used. In our proposal, however, we will use automata, as once built, they are more efficient to be traversed.

In this paper we make efficient solver technology applicable to planning with state trajectory constraints. As the approach relies on a translation of PDDL into LTL formula, we can deal even with larger constructs than the ones that are currently under considerations. The text is structured as follows. First we turn to temporally extended goals in the form of state trajectory constraints and discuss equivalent LTL and Büchi automata representations for most of the examples in [8]. Afterwards we study how to perform exploration in goals with trajectory constraints. Starting from a model checking perspective, we discuss the need of nested search. At the end we will present, how the language features can be compiled away to ordinary PDDL by simulating the automata with synchronized actions. Finally, we reflect related work and draw conclusion.

2 Automata-based Model Checking

We briefly recall automata-based model checking, a common approach to the model checking problem. It is not per se LTL specific. The idea is to represent both the model to be analyzed and the specification to be checked as *Büchi automata*. Syntactically, Büchi automata are the same as finite state automata, but designed for the acceptance of infinite words. Therefore, they have a different acceptance condition. Let ρ be a run and $\mathit{inf}(\rho)$ be the set of states reached infinitely often in ρ, then a Büchi automaton accepts, if the intersection between $\mathit{inf}(\rho)$ and the set of final states F is not empty.

Transforming the model and the specification into Büchi automata assumes that systems can be modeled by finite automata, that the automaton for the model considers all states as accepting, and that the LTL formula can be transformed into an equivalent Büchi automaton. The contrary not always possible, since Büchi automata are clearly more expressive than LTL expressions [20].

Checking correctness is reduced to checking emptiness of this automaton. More formally, the model checking procedure validates that a model represented by an automaton $\mathcal{M}$ satisfies its specification represented by an automaton $\mathcal{S}$. The task is to verify if $\mathcal{L}(\mathcal{M}) \subseteq \mathcal{L}(\mathcal{S})$. In words: the *language accepted by the model is included in that of the specification.* We have that $\overline{\mathcal{L}(\mathcal{S})} = \Sigma^\omega - \mathcal{L}(\mathcal{S})$ so that $\mathcal{L}(\mathcal{M}) \subseteq \mathcal{L}(\mathcal{S})$ holds if and only if $\mathcal{L}(\mathcal{M}) \cap \overline{\mathcal{L}(\mathcal{S})} = \emptyset$ is given. In practice, checking emptiness is more efficient than checking inclusion.

Büchi automata are closed under intersection and complementation [2], so that there exists an automaton that accepts $\overline{\mathcal{L}(\mathcal{S})}$ and an automata that accepts $\mathcal{L}(\mathcal{M}) \cap \overline{\mathcal{L}(\mathcal{S})}$. It is possible to complement Büchi automaton equivalent to LTL formula, *but* the worst-case running time of such a construction is double-exponential in the size of the formula. Therefore, in practice one constructs the automaton for negation of the LTL formula, avoiding complementation.

Given a Büchi automaton $\mathcal{M}$ for the model $\mathcal{M} = \langle \Sigma, S_{\mathcal{M}}, \Delta_{\mathcal{M}}, S_0^{\mathcal{M}}, F_{\mathcal{M}} \rangle$ with $S_{\mathcal{M}} = F_{\mathcal{M}}$ and a Büchi automaton $\mathcal{N}$ for the negation of the specification $\mathcal{N} = \langle \Sigma, S_{\mathcal{N}}, \Delta_{\mathcal{N}}, S_0^{\mathcal{N}}, F_{\mathcal{N}} \rangle$ the intersection $\mathcal{M} \cap \mathcal{N}$ of $\mathcal{M}$ and $\mathcal{N}$ is a Büchi automaton $\langle S, \Sigma, \Delta, S_0, F \rangle$, where:

- $S = S_{\mathcal{M}} \times S_{\mathcal{N}}$; $S_0 = S_0^{\mathcal{M}} \times S_0^{\mathcal{N}}$;
- $F = F_{\mathcal{M}} \times F_{\mathcal{N}} = S_{\mathcal{M}} \times F_{\mathcal{N}}$;
- $\Delta = \{((s_{\mathcal{M}}, s_{\mathcal{N}}), a, (s'_{\mathcal{M}}, s'_{\mathcal{N}})) \mid s_{\mathcal{M}}, s'_{\mathcal{M}} \in S_{\mathcal{M}} \wedge s_{\mathcal{N}}, s'_{\mathcal{N}} \in S_{\mathcal{N}} \wedge (s_{\mathcal{M}}, a, s'_{\mathcal{M}}) \in \Delta_{\mathcal{M}} \ \wedge \ (s_{\mathcal{N}}, a, s'_{\mathcal{N}}) \in \Delta_{\mathcal{N}}\}$

The automata product is *synchronous*, that is each transition in one automata matches one in the other. We also see that the property automata is non-deterministic, such that both the successor generation and the temporal formula representation may introduce branching to the overall exploration module. As said, the construction assumes that all states in the model itself are accepting. If arbitrary Büchi automata are intersected, one has to include an extended acceptance condition [3].

3 Application to Planning

In the proposed extension to planning we do not have to negate the formula as planning goals $\mathcal{G}$ already correspond to the negations of properties in model checking. In other words, we search for witnesses of $L(\mathcal{M}) \cap L(\mathcal{G}) \neq \emptyset$, where $\mathcal{M}$ is the original plan space. If ordinary goals ϕ without any temporal modalities are used, we may either add their restrictions to the acceptance condition in the model or the specification.

For a proper exploration we need a Büchi automaton representation of the formula and some explorat¡ion algorithm that can figure out if the language intersection is not empty. By the semantics of [8] it is clear that all sequences are finite, so that we can interpret the Büchi automaton as a (non-deterministic) finite state automaton (NFA), which accepts a word if it ends up in a final state. The labels of the automaton are conditions over the proposition and fluents in a given state. It is well known that a NFA can be transformed in a equivalent determinisitic finite automaton (DFA) using a power set construction. This DFA, however, can become exponential in the size of the NFA, so in most cases a on-the-fly simulation will be preferable.

It is also important to note that most temporal expressions are universally quantified. These quantifications can be eliminated through grounding predicates. This is possible due to the finite scope of the quantified variables with respect to the referenced object types. Therefore, universal quantification resolves

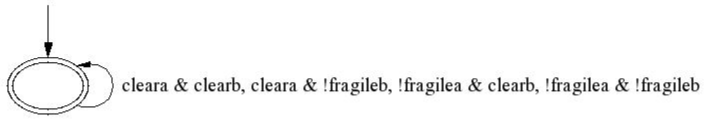
cleara & clearb, cleara & !fragileb, !fragilea & clearb, !fragilea & !fragileb

Fig. 1. Automaton for the *fragile-nothing* Property.

to large conjuncts. The conjunct of two LTL expressions itself corresponds to the intersection of the two languages.

4 Automata for the Example Instances

Next we look at the examples in the original text of Gerevini and Long. Most example refer to Blocksworld and Logistics problems. In PDDL with state trajectory constraints, the expression *A fragile block can never have something above it* is expressed as

```
(:goal (and (always (forall (?b - block) (implies (fragile ?b) (clear ?b))))
```

We call this condition the *fragile-noting* property. The LTL formula for two selected blocks a and b is

```
G ((fragilea -> cleara) & (fragileb -> clearb))
```

The corresponding automaton is shown in Figure 1.

The automaton consists of only one state. Commas are interpreted as disjunctions. The label of the transition is equivalent to the expression ϕ in $G\phi$ as $a \to b \wedge c \to d$ is equivalent to $(\neg a \wedge \neg c) \vee (\neg a \wedge d) \vee (b \wedge \neg c) \vee (b \wedge d)$.

The effect of synchronizing the automaton with the planning state space will result in a pruning of the search space. Therefore, the automata representation subsumes the control rule pruning that is inherent to TL and TALplan. For this case, control rules have to be defined as state trajectory constraints and then transferred to Büchi automata.

The expression *A fragile block can have at most one block on it* is denoted as

```
(:goal (and (always (forall (?b1 ?b2 - block)
            (implies (and (fragile ?b1) (on ?b2 ?b1)) (clear ?b))))
```

The LTL formula with two blocks is given by

```
G (((fragilea & onba) -> clearb) & ((fragileb & onab) -> cleara))
```

The according Büchi-Automaton is shown in Figure 2. As the pattern is of type $G\phi$ the automaton also consists of only one state.

In the proposed extension of PDDL the assertion *each block should be picked up at least once* it read as:

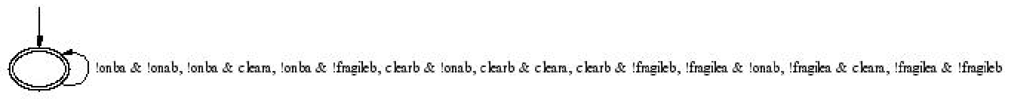
!onba & !onab, !onba & cleara, !onba & !fragileb, clearb & !onab, clearb & cleara, clearb & !fragileb, !fragilea & !onab, !fragilea & cleara, !fragilea & !fragileb

Fig. 2. Automaton for the *fragile-clear* property

```
(:goal (and (forall (?b - block) (sometime (holding ?b)))
```

We call the specification the *every-block/some-state* property. We chose two blocks and constructed a Büchi automaton with respect to the LTL formula

```
(F holdinga) & (F holdingb)
```

It is shown in Figure 3 (left).

The statement *in some state visited by the plans all blocks are on the table* is declared with the PDDL expression

```
(:goal (sometime (forall (?b - block) (ontable ?b)))
```

We call the specification the *sometime/every-block* property. The LTL formula we chose for illustration is

```
F (ontablea & ontableb)
```

with a Büchi automata shown in Figure 3 (middle). It is much simpler than the previous one.

The expression *Each truck should visit each city exactly once* is given by

```
(:goal (and (forall (?t - truck ?c - city) (at-most-once (at ?t ?c)))
            (forall (?t - truck ?c - city) (sometime (at ?t ?c))))
```

We call the requirement a *exactly-once* property. We used a simple instantiation with one truck and one city, yielding the LTL formula

```
F attruckacitya & G (attruckacitya -> (attruckacitya U (G !attruckacitya)))
```

The corresponding Büchi-Automaton is displayed in Figure 3 (right). Note that the second part of the formula shows the *at-most-once* property. The according automaton is the same as in Figure 3, with simply the first state being accepting.

The expression *Each city is visited by at most one truck at the same time* can be expressed by

```
(:goal (forall (?t1 ?t2 - truck ?c1 - city)
       (always (implies (and (at ?t1 ?c1) (at ?t2 ?c1)) (= ?t1 ?t2))))
```

We used a simple instance with two trucks trucka and truckb in citya.

```
G ((attruckacitya & attruckbcitya) -> equalstruckatruckb)
```

The corresponding Büchi-Automaton is a simple one-state automata as in the beginning of the section.

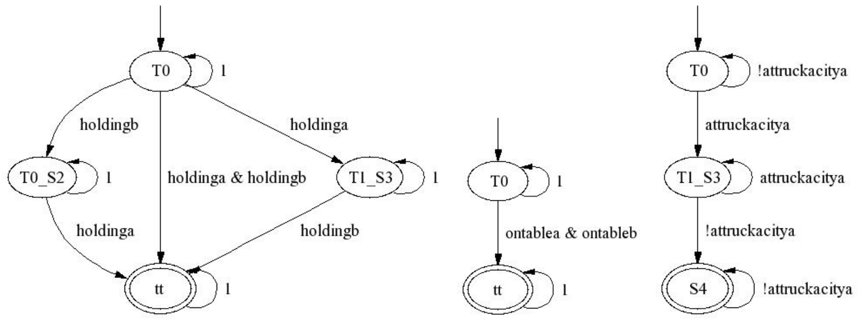

Fig. 3. Büchi automata for the *every-block/some-state* property (left), for the *sometime/every-block* property (middle) and for the *exactly-once* property (right).

5 Exploration

On-the-fly model checking is an efficient way to perform model checking. It computes the global state transition graph during the construction of the intersection. The advantage is only that part of the state space is constructed that is needed in order to check the desired property. In case an error/goal is detected it is not necessary to continue generating states. To check language emptiness, we have to validate that the automaton accepts no word. By the semantics of Büchi automata this is equivalent to examine that the strongly connected components reachable from initial state and contain at least one accepting state. In other words if there is no reachable cycle containing at least one accepting state the language is empty and no counterexample/plan exists.

In on-the-fly model checking accepting cycles are efficiently detected by *nested depth-first search* [12] The first DFS procedure explores the state space in depth-first manner, stores visited states in *Closed* list, marks states which are on the current search stack and invokes a second DFS for accepting states in postorder. The second DFS procedure explores states already visited by the first one but not by any secondary search. All states visited by the second search are *flagged*, and if a state is found on the stack of first search, an accepting cycle is found. Typical implementations use 2 bits per state, one for each DFS. The worst-case time complexity is linear in the size of the global state transition graph.

The structure of the goal property automata is important to decide, which exploration algorithm to take. Fortunately, we have that all cycles in the example cases are either fully accepting or non-accepting, so that we for the LTL language fragment that is proposed, no nested search is required. Moreover, as we have only have finite runs to judge, so we can drop the Büchi acceptance condition and work with the corresponding condition for a non-deterministic finite state automaton (NFA). Moreover, accepting states have self-loops, which makes it possible to easily focus the search and to detect termination. We will see that it is possible to include any state traversal politics together with such automata that runs in parallel to the search.

Probably the easiest way to implement a planning algorithm is to extend any existing forward chaining state planner in a such a way that the state vector s for current planning state is extended to (s, n), with n being the corresponding automata state. That is the approach usually featured in explicit-state model model checkers like SPIN. As the automata is non-deterministic, ordinary successors s' of s in original plan space may imply more than one successor $(s', n_1), \ldots, (s', n_k)$ in the extended one.

6 Language Compilation

Have the language extensions enriched the PDDL language or is it possible to compile the language extensions to already existing fragments? Fortunately, for most cases is possible to give an efficient language compilation.

To encode the working of the automata, we devise a predicate `(at-state ?n - state)` instantiated for each state in the automata. The connection between two states is realized by a static predicate `(transition ?n1 ?n2 - state)`. To detect accpeting states, we require a tag `(is-accepting ?n - state)`. Next, we have to specify which transition is allowed. This can be done by declaring one grounded operator for each transition, with the current automata state and the transition conditions as a precondition and the successor state as the add effect. Since goals are instantiated, adding the transition labels to operator preconditions can only be done in the grounded representation.

As accepting states match the proposed language extensions, we simply have to include the proposition `(is-accepting ?s)` to the set of goals. This substitutes the state trajectory constraint. We also require a synchronization mechanism between the goal property automaton and the planning space space, which is available by having a `synchronize` flag that is put on and of when a ordinary operator and a property automata transition is chosen. The compilation, however, is not *simple*, as the blow-up can be exponential, given that the size of the Büchi automata for a given formula can be exponential in the length of the formula. Therefore, we judge this language extension as essential[1]. However, the size of the Büchi automaton is often small compared to the size of the model.

The above setting extends to more than one state trajectory constraint through introducing an additional parameter `?a - automata` to the the propositions `at-state`, `is-accepting`, `transition`, and `synchronize`. This conjunctive split into several concurrent automata can also be an apparent option when constructing one Büchi automata is involved.

[1] Such a notation of *essentiality* goes back to Nebel, who provides a theory on domain compilations. Similar essential compilations have been proposed by Gazen and Knoblock for ADL to STRIPS and by Hoffmann, Nebel and Thiebaux for derived predicates/axioms.

7 Experimental Results

We wrote a compiler that takes the PDDL input that includes the state trajectory constraints and prompts the textual changes needed to have the constraints compiled away. We kept all the implementation planner-independent, although it is not difficult to internalize the methods. The tool divides into two parts. The first one, called *PDDL2LTL*, parses the problem description, resolves the bounded quantification and writes the corresponding LTL-formula to disk. The second one, called *LTL2PDDL* derives the Büchi-automata representation from the LTL formula and generates the corresponding PDDL code for modifying the domain description. This conversion is derived from the tool *LTL2BA* by Oddoux and Gastin[2]. We draw initial experiments on an Intel Pentium M, 2 GHz, 512 MByte computer. In *Blocks World* we selected a problem with 10 blocks a,... ,i. In the initial state we have the two towers $(c, e, j, b, g, h, a, d, i)(f)$, while in the goal we require one tower $(d, c, f, j, e, h, b, a, g, i)$. We use

```
F ontable-a & F ontable-b & F ontable-c & F ontable-d & F ontable-e
```

as a state trajectory constraint that generates an Büchi automaton with 64 states, one of which is accepting.

Without a state trajectory constraint the MetricFF planner [10] evaluated 283 states in 0.62 CPU seconds to find a 42 step long sequential plan (cf. Figure 4). It has 131 facts and 220 actions. With state trajectory based on six blocks to be placed *ontable*, the Büchi automata consists of 64 states. The resulting grounded description contains of 188 facts and 929 actions. The CPU time for evaluating 1,024 states to find a plan of 2×49 states was 2.19s.

8 Related Work

Applying temporal logic to constrain the exploration is a wide-spread field. There are two concurrent approaches: formula progression as applied in AI planning [1] and Büchi automata as applied in model checking [20]. In principle these methods are equivalent, but it is not clarified what method should be preferred for which problems. Constructing the Büchi automata prior to the search is a time-critical task. However, the savings during the search are considerable, as for each state in the search space we need only to maintain the position in the automaton.

To take care of both extended goals and control knowledge, [13] apply a hyprid algorithm, formula progression for the LTL control knowledge and Büchi automata for analyzing the temporally extended goal. The methods are applied concurrently for an extended state vector that includes currently progressed formula and the current state in the Büchi automaton.

There is already some related work on compiling temporal logic control into PDDL. Rintanen [17] compared formula progression as in [1] with a direct translation into plan operators. His translation applies all changes to the operators

[2] Similar results are obtained with the tool LTL$\rightarrow$NBA and the never-claim converter that is inherent to SPIN

so that produced plans remain unchanged. Instead of PDDL input, he considers set of clauses with *next-time* and *until*. For each of the clauses, the operators in his translation contain at most 7 plus the number of literals many effects. For TLPlan he obtained a time improvement within a factor of 2-4 with respect to the standard approach of formula progression. In difference to Rintanen's work, we choose the Büchi automata interpretation adopted of model checking to translate the modalities and a simple synchronisation mechanism to organize the search. In standard STRIPS notation only two flags are needed to realize synchronous execution. Operator instantiation and temporal formula conversion are independent. The worst-case exponential number of operators in the size of the unrolled formula (its literals) is not a burden as in most cases it is smaller than the number of ordinary planning operators. Our translation features heuristic search planning and applies to more expressive planning formalisms.

Fox, Long and Hasley [7] considered the transformation of *maintenance* and *deadline goals* in the context of PDDL2.1 planning. The setting is slightly different as they consider formulas of the type $U\ p\ c$ (and $F\ p\ c$), where p is the condition until which (or from which condition) c must hold. Similar to here they used a synchronized additional condition-checking operator, which runs in parallel to the others operators to govern the exploration. A technique called, $\epsilon/3$-slicing is used to preserve temporal plan-execution based on the ϵ-transition that are neccesary in between every two dependent happenings. The advantage of the method is that it is very much concerned on parallel/temporal plan execution and operator mutexes. Unfortunately, this approach has not been implemented.

9 Conclusion

In this paper, we gave a Büchi automata interpretation for state-trajectory constraints. We provided an effective compilation of the new language fragment to allow already existing planners to deal with the new requirements. The plan length increases by factor 2, while the number of actions can rise exponentially in the unrolled size of the state trajectory constraint. As the translation of the nondeterministic Büchi automaton in PDDL is canonical, we expect not much gain in including the transformation process into a planner.

We observed that the automata-based approach from explicit-state LTL model checking applies to the framework, with the set of goals as the negation of the property specification. A plan is a witness of the language containment problem $L(\mathcal{G}) \cap L(\mathcal{M}) \neq \emptyset$. There are Büchi automata for most temporal properties, probably large by quantification, but in most cases simple in structure. Nested search is not needed. The semantics of trajectory constraints are based on finite runs, such that the Büchi accepting condition is re-interpreted as one for an NFA. It is possible to compile the language extension to ordinary PDDL by simulating the synchronous working of the automata. We have not yet included the state trajectory constraints *(always-)within*. They refer to plan deadlines and execution time windows. We conjecture that these constraints can be dealt with the existing ones and timed initial literals as introduced to PDDL in 2004.

References

1. F. Bacchus and F. Kabanza. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 116:123–191, 2000.
2. J. R. Buchi. On a decision method in restricted second order arithmetic. In *Conference on Logic, Methodology, and Philosophy of Science*, LNCS, pages 1–11. Stanfort University Press, 1962.
3. E. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 2000.
4. S. Edelkamp. Planning with pattern databases. In *European Conference on Planning (ECP)*, pages 13–24, 2001.
5. S. Edelkamp, S. Leue, and A. Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *International Journal on Software Tools for Technology (STTT)*, 5(2-3):247–267, 2004.
6. M. Fox and D. Long. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Research (JAIR)*, 20:61–124, 2003.
7. M. Fox, D. Long, and K. Hasley. An investigation on the expressive power of PDDL2.1. In *European Conference on Artificial Intelligence (ECAI)*, 2004.
8. A. Gerevini and D. Long. Plan constraints and preferences in pddl. A Proposal for the Language of the Fifth International Planning Competition, 2005.
9. M. Helmert. A planning heuristic based on causal graph analysis. In *International Conference on Automated Planning and Scheduling (ICAPS)*, pages 161–170, 2004.
10. J. Hoffmann. The Metric FF planning system: Translating "Ignoring the delete list" to numerical state variables. *Journal of Artificial Intelligence Research*, 20:291–341, 2003.
11. J. Hoffmann and B. Nebel. Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.
12. G. J. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. *The SPIN Verification System*, pages 23–32, 1972.
13. F. Kabanza and S. Thiebaux. Search control in planing for termporally extended goals. In *International Conference on Automated Planning and Scheduling (ICAPS)*, pages 130–139, 2005.
14. J. Kvarnström, P. Doherty, and P. Haslum. Extending TALplanner with concurrency and ressources. In *European Conference on Artificial Intelligence (ECAI)*, pages 501–505, 2000.
15. U. D. Lago, M. Pistore, and P. Traverso. Planning with a language for extended goals. In *National Conference on Artificial Intelligence (AAAI)*, pages 447–454, 2002.
16. M. Pistore and P. Traverso. Planning as model checking for extended goals in non-deterministic domains. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 479–486, 2001.
17. J. Rintanen. Incorporation of temporal logic control into plan operators. In *European Conference on Artificial Intelligence (ECAI)*, pages 526–530, 2000.
18. S. Safra. On the complexity of omega-automata. In *Annual Symposium on Foundations of Computer Science*, pages 319–237. IEEE Computer Society, 1998.
19. A. P. Sistla, M. Y. Vardi, and P. Wolper. The complementation problem for Buchi automata with applications to temporal logic. *Theoretical Computer Science*, 49(2–3):217–237, 1983.
20. P. Wolper. Temporal logic can be more expressive. *Information and Control*, 56:72–99, 1983.

```
 0: UNSTACK C E
 1: STACK C F
 2: UNSTACK E J
 3: STACK E C
 4: UNSTACK J B
 5: STACK J E
 6: UNSTACK B G
 7: STACK B J
 8: UNSTACK G H
 9: PUT-DOWN G
10: UNSTACK H A
11: STACK H B
12: UNSTACK A D
13: STACK A H
14: UNSTACK D I
15: STACK D A
16: PICK-UP G
17: STACK G I
18: UNSTACK D A
19: PUT-DOWN D
20: UNSTACK A H
21: STACK A G
22: UNSTACK H B
23: STACK H D
24: UNSTACK B J
25: STACK B A
26: UNSTACK H D
27: STACK H B
28: UNSTACK J E
29: STACK J D
30: UNSTACK E C
31: STACK E H
32: UNSTACK J D
33: STACK J E
34: UNSTACK C F
35: STACK C D
36: PICK-UP F
37: STACK F J
38: UNSTACK C D
39: STACK C F
40: PICK-UP D
41: STACK D C
```

```
 0: UNSTACK C E
 2: PUT-DOWN C
 4: PICK-UP C
 6: STACK C F
 8: UNSTACK E J
10: PUT-DOWN E
12: UNSTACK J B
14: STACK J E
16: UNSTACK B G
18: PUT-DOWN B
20: UNSTACK G H
22: STACK G J
24: UNSTACK H A
26: STACK H B
28: UNSTACK A D
30: STACK A G
32: UNSTACK D I
34: PUT-DOWN D
36: PICK-UP D
38: STACK D C
40: UNSTACK A G
42: PUT-DOWN A
44: PICK-UP A
46: STACK A D
48: UNSTACK G J
50: STACK G I
52: UNSTACK A D
54: STACK A G
56: UNSTACK H B
58: STACK H D
60: PICK-UP B
62: STACK B A
64: UNSTACK H D
66: STACK H B
68: UNSTACK J E
70: STACK J D
72: PICK-UP E
74: STACK E H
76: UNSTACK J D
78: STACK J E
80: UNSTACK D C
82: PUT-DOWN D
84: UNSTACK C F
86: STACK C D
88: PICK-UP F
90: STACK F J
92: UNSTACK C D
94: STACK C F
96: PICK-UP D
98: STACK D C
```

```
 1: SYNC-INIT-T0-2
 3: SYNC-T0-2-T0-10
 5: SYNC-T0-10-T0-10
 7: SYNC-T0-10-T0-10
 9: SYNC-T0-10-T0-10
11: SYNC-T0-10-T0-12
13: SYNC-T0-12-T0-12
15: SYNC-T0-12-T0-12
17: SYNC-T0-12-T0-12
19: SYNC-T0-12-T0-28
21: SYNC-T0-28-T0-28
23: SYNC-T0-28-T0-28
25: SYNC-T0-28-T0-28
27: SYNC-T0-28-T0-28
29: SYNC-T0-28-T0-28
31: SYNC-T0-28-T0-28
33: SYNC-T0-28-T0-28
35: SYNC-T0-28-T0-32
37: SYNC-T0-32-T0-32
39: SYNC-T0-32-T0-32
41: SYNC-T0-32-T0-32
43: SYNC-T0-32-ACCEPT-0
45: SYNC-ACCEPT-0-ACCEPT-0
47: SYNC-ACCEPT-0-ACCEPT-0
49: SYNC-ACCEPT-0-ACCEPT-0
51: SYNC-ACCEPT-0-ACCEPT-0
53: SYNC-ACCEPT-0-ACCEPT-0
55: SYNC-ACCEPT-0-ACCEPT-0
57: SYNC-ACCEPT-0-ACCEPT-0
59: SYNC-ACCEPT-0-ACCEPT-0
61: SYNC-ACCEPT-0-ACCEPT-0
63: SYNC-ACCEPT-0-ACCEPT-0
65: SYNC-ACCEPT-0-ACCEPT-0
67: SYNC-ACCEPT-0-ACCEPT-0
69: SYNC-ACCEPT-0-ACCEPT-0
71: SYNC-ACCEPT-0-ACCEPT-0
73: SYNC-ACCEPT-0-ACCEPT-0
75: SYNC-ACCEPT-0-ACCEPT-0
77: SYNC-ACCEPT-0-ACCEPT-0
79: SYNC-ACCEPT-0-ACCEPT-0
81: SYNC-ACCEPT-0-ACCEPT-0
83: SYNC-ACCEPT-0-ACCEPT-0
85: SYNC-ACCEPT-0-ACCEPT-0
87: SYNC-ACCEPT-0-ACCEPT-0
89: SYNC-ACCEPT-0-ACCEPT-0
91: SYNC-ACCEPT-0-ACCEPT-0
93: SYNC-ACCEPT-0-ACCEPT-0
95: SYNC-ACCEPT-0-ACCEPT-0
97: SYNC-ACCEPT-0-ACCEPT-0
```

Fig. 4. Generated plans in the original and in the constraint state space for the Blocksworld problem.