Using Finite-State Automata to Model and Solve Planning Problems

Daniel Toropila*[†], Roman Barták*

*Charles University in Prague, Faculty of Mathematics and Physics
Department of Theoretical Computer Science and Mathematical Logic
Malostranské nám. 2/25, 118 00 Praha 1, Czech Republic
roman.bartak@mff.cuni.cz

[†]Charles University in Prague, Computer Science Center
Ovocný trh 5, 116 36 Praha 1, Czech Republic
daniel.toropila@mff.cuni.cz

Abstract

In the recent years, the representation using the state variables, i.e., SAS+ planning, has become one of the most popular formalisms for specifying the planning problems (and domains). Still, the great majority of the planning tasks, especially from the past planning competitions, are defined using the classical STRIPS-like formalism, and so an efficient transformation technique is required in order to allow performance comparison of individual planners. In the paper we present the new formalism that is based on finite-state automata (FSA), and that represents an alternative to the SAS$^+$ (state variables), providing thus a novel platform for development of the modern planners. Finally, we also propose conversion procedures between the individual planning problem representations, together with the basics of automated and semi-automated techniques that employ the new formalism and aim towards the extraction of the state variables representation from the classical representation.

Introduction

It is a well-known fact that the careful choice of the problem representation is one of the key aspects in the endeavor to find its solution. Planning problems are traditionally represented using predicate logic where the world is described by a set of propositions that are either true or false in individual world states. Actions are then changing the validity of certain propositions. This is a very general representation that has some disadvantages such as that it is not possible to directly express that some facts cannot be true together in a single world state. For example, the location of robot is described by a set of propositions claiming that the robot is in a particular location. Obviously, it is not possible that the robot is at two different locations at the same time, but the classical logical model does not take this in account (unless some mutual exclusion relations describe the fact explicitly). Multi-valued state variables were therefore proposed to naturally model these mutually exclusive relations. The major advantage is that this representation is more compact and implicitly covers some mutex relations. Planning problem can be then seen as finding the synchronized evolution of state variables as described in (Pralet and Verfaillie, 2009). It is also interesting to see that the multi-valued state representation found its way to the Boolean satisfiability solvers that are traditionally based on logical representation and helps there in further improvement of efficiency (Huang, Chen, and Zhang, 2010).

The evolution of multi-valued state variables can naturally be represented using a finite state automaton where the states describe the values of the state variable and the transitions describe how the actions are changing the values. We propose using finite state automata as an alternative way for representing classical planning problems and we shall describe how this representation is related to existing representation and sketch some transformation procedures. The ultimate goal of this work is, however, not to provide yet another representation framework for planning problems, but rather to introduce the theoretical foundation for a technique that would allow straightforward modeling of the planning domains, and which could also serve as an intermediate form for the conversion from the classical logical representation into the SAS$^+$ encoding.

The paper is organized as follows. We will first introduce the necessary concepts from the theory of finite-state automata. Then we will continue with the description of the two most popular representations of planning problems: classical STRIPS-like representation and SAS$^+$ encoding based on the multi-valued variables, followed by the introduction of the novel encoding, the basic building block of which is a deterministic finite automaton (acceptor). After that we will provide the conversion procedures that will allow us to generate automata-based encodings from the existing representations. Finally, we will present the algorithm for translating an arbitrary set of deterministic finite automata into the SAS$^+$-based encoding of an equivalent planning problem.

Finite-State Automata

A *finite-state automaton* (FSA), sometimes also called a *finite-state machine* (FSM), is a popular mathematical abstraction used widely in computer science, mostly to design digital logic or computer programs. It is a behavioral model that consists of a finite number or states, transitions between those states, and actions, which execute the state transitions. FSA's can also produce some output, using the defined output alphabet, but for the purpose of this paper we shall consider only the automata that produce binary output, *true* or *false*, based on whether the state of the FSA after processing the input is from the set of *accepting* states. The following formal definition is due to (Carrol and Long 1989).

Deterministic Finite Automaton. A *deterministic finite automaton* or *deterministic finite acceptor* (DFA) is a quintuple $<\Sigma, S, s_0, \delta, F>$, where

- Σ is the *input alphabet* (a finite nonempty set of symbols)
- S is a finite nonempty set of *states*,
- s_0 is the *start* (or *initial*) state, an element of S,
- δ is the state transition function; δ: $S \times \Sigma \rightarrow S$, and
- F is the set of *final* (or *accepting*) states, a (possible empty) subset of S. ■

The input alphabet Σ for any deterministic finite automaton A is the set of symbols that can appear on the input tape. Each successive symbol in a word will cause a transition from the present state to another state in the machine. As specified by the δ function, there is exactly one such state transition for each combination of the symbol $a \in \Sigma$ and the state $s \in S$. This is the origin of the word "deterministic" in the phrase "deterministic finite automaton".

Each FSA A defines a set of accepted inputs $L \subseteq \Sigma^*$, a *language* over the alphabet Σ, denoted also as $\Lambda(A)$. For two automata A_1 and A_2, recognizing the languages L_1 and L_2, respectively, it is sometimes necessary to find their intersection $L = L_1 \cap L_2$, i.e., the set of inputs that is accepted by both automata. This can be done by running the two automata in parallel and accepting the input only when both automata terminate in an accepting state. As one might guess, it is also possible to construct a new automaton $A = A_1 \cap A_2$ that, in fact, performs the parallel computation of A_1 and A_2 (Sipser 2006):

Construction of the DFA Intersection. Let A_1 recognize L_1, where $A_1 = <\Sigma, S_1, s_1, \delta_1, F_1>$, and A_2 recognize L_2, where $A_2 = <\Sigma, S_2, s_2, \delta_2, F_2>$. The automaton $A = <\Sigma, S, s_0, \delta, F>$ recognizing $L_1 \cap L_2$ looks as follows:

- $S = \{(r_1,r_2) \mid r_1 \in S_1 \text{ and } r_2 \in S_2\}$,
- $\forall (r_1,r_2) \in S, \forall a \in \Sigma$: $\delta((r_1,r_2),a) = (\delta_1(r_1,a),\delta_2(r_2,a))$,
- s_0 is the pair (s_1,s_2), and
- $F = F_1 \times F_2$. ■

Naturally, after performing the intersection operation we can filter out the states of the resulting automaton that are not reachable from its initial state.

Representations for Classical Planning

Classical AI planning deals with finding a sequence of actions that transfer the world from some initial state to a desired state (Ghallab, Nau and Traverso 2004). The state space is large but finite. It is also *fully observable* (we know precisely the state of the world), *deterministic* (the state after performing the action is known), and *static* (only the entity for which we plan changes the world). Moreover, we assume actions to be instantaneous so we only deal with action sequencing.

Classical Representation

Throughout the years, probably the most popular and widely used formalism for representing the (not only classical) planning problems has been the first-order-logic-based *classical representation*, where the world *state* is described as a set of predicates that hold in the state, such as location(robot,loc1) saying that robot is located at loc1. In other words, for each predicate and for each state we describe whether the predicate holds in the state or not. *Actions* are described using a triple ($Prec$, Eff^+, Eff^-), where $Prec$ is a set of predicates that must hold for the action to be applicable (preconditions), Eff^+ is a set of predicates that will hold after performing the action (positive effects), and finally Eff^- is a set of predicates that will not hold after performing the action (negative effects). For example, action move(robot,loc1,loc2) describing that robot moves from loc1 to loc2 is specified as triple ({location(robot,loc1)}, {location(robot,loc2)}, {location(robot,loc1)}). Formally, action a is applicable to state s if $Prec(a) \subseteq s$. The result of applying action a to state s is a new state $\gamma(s,a) = (s - Eff^-(a)) \cup Eff^+(a)$. Notice that this description assumes the *frame axiom*, that is, other predicates than those mentioned among the effects of the action are not changed by applying the action.

The set of predicates together with the set of actions is called a *planning domain*. We assume both sets of predicates and actions to be finite. The *goal* is specified as a set of predicates that must hold in the goal state, that is, if g is a goal then any state s such that $g \subseteq s$ is a goal state. The *classical planning problem* is defined by the planning domain, the initial state s_0 and the goal g, and the task of planning is to find a sequence of actions $\langle a_1, a_2, \ldots, a_n \rangle$ called a *plan* such that a_1 is applicable to the initial state s_0, a_2 is applicable to state $\gamma(s_0, a_1)$ etc., and $g \subseteq \gamma(\ldots\gamma(\gamma(s_0,a_1),a_2)\ldots,a_n)$.

Multi-Valued State Variables (SAS$^+$)

There exists, however, an alternative to the above logical formalism that is based on so called *multi-valued state* *variables*, as mentioned in (Bäckström and Nebel 1995) or (Helmert 2006), often also referred to as SAS^+. For each feature of the world, there is a variable describing this feature, for example rloc(robot,*s*) describes the position of robot at state *s*. Now, instead of specifying validity of the predicate in some state *s*, say location(robot,loc1), we can specify the value of the state variable in a given state, in our example rloc(robot, *s*) = loc1. Hence the evolution of the world can be described as a set of state-variable functions where each function specifies evolution of values of certain state variable. Now, the actions are described as entities changing the values of state variables. We can still use preconditions specifying required values of certain state variables, but the positive and negative effects are merged to the effects of setting the values of certain state variables. Figure 1 depicts an example of such encoding, together with the equivalent classical representation.

Notice that this multi-valued formulation is more compact than the logical formulation, where, for example, one needs to express explicitly that if robot is at loc1 then it is not present at another location. For example, the action of moving robot from loc1 to loc2 needs to explicitly describe (in negative effects) that after performing the action, the predicate location(robot,loc1) is no more valid. In the multi-valued representation assigning value loc2 to state variable rloc(robot,*s*) implicitly means that robot is not at a different location at state *s*.

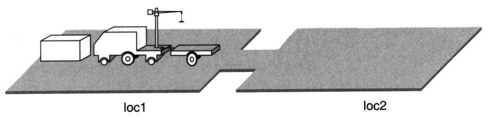
Representation Using FSAs

If we take a look at the SAS^+ representation from a different perspective, we can observe that each state variable can be viewed as a representation of a smaller independent state-transition sub-system that is an integral part of the "world" that we are planning in. The whole planning problem can be then viewed as a set of such independent state-transition sub-systems (properties of the world) that are all modified, in parallel, by the same sequence of actions, where the actions that do not affect the given state variable leave its value intact. As an example, consider the Dock-Worker Robots (DWR) domain where the location of a robot is completely independent from the position of a container (see an example in Figure 1; the domain illustration is taken from (Ghallab, Nau and Traverso 2004)).

This brings us to the idea of representing the planning problem using a model that integrates both the state variables and the definitions of available actions – a set of FSAs, where each automaton corresponds to a state variable in a following manner:

Domain

DWR domain with two locations (loc1,loc2), a robot capable of loading and unloading containers by itself (r), and one container (c)



CLASSICAL (STRIPS-like)

Available Facts

location(r, loc1), location(r, loc2),
position(c, loc1), position(c, loc2), position(c, r)

Actions

1 : move(r, loc1, loc2)
;; robot r at location loc1 moves to location loc2
Precond: location(r, loc1)
Effects: ¬location(r, loc1), location(r, loc2)

2 : move(r, loc2, loc1)
;; robot r at location loc2 moves to location loc1
Precond: location(r, loc2)
Effects: ¬location(r, loc2), location(r, loc1)

3 : load(r, c, loc1)
;; robot r loads container c at location loc1
Precond: location(r, loc1), position(c, loc1)
Effects: ¬position(c, loc1), position(c, r)

4 : load(r, c, loc2)
;; robot r loads container c at location loc2
Precond: location(r, loc2), position(c, loc2)
Effects: ¬position(c, loc2), position(c, r)

5 : unload(r, c, loc1)
;; robot r unloads container c at location loc1
Precond: location(r, loc1), position(c, r)
Effects: ¬position(c, r), position(c, loc1)

6 : unload(r, c, loc2)
;; robot r unloads container c at location loc2
Precond: location(r, loc2), position(c, r)
Effects: ¬position(c, r), position(c, loc2)

SAS^+

State Variables

rloc ∈ {loc1,loc2} ;; robot's location
cpos ∈ {loc1,loc2,r} ;; container's position

Actions

1 : move(r, loc1, loc2)
;; robot r at location loc1 moves to location loc2
Precond: *rloc* = loc1
Effects: *rloc* ← loc2

2 : move(r, loc2, loc1)
;; robot r at location loc2 moves to location loc1
Precond: *rloc* = loc2
Effects: *rloc* ← loc1

3 : load(r, c, loc1)
;; robot r loads container c at location loc1
Precond: *rloc* = loc1, *cpos* = loc1
Effects: *cpos* ← r

4 : load(r, c, loc2)
;; robot r loads container c at location loc2
Precond: *rloc* = loc2, *cpos* = loc2
Effects: *cpos* ← r

5 : unload(r, c, loc1)
;; robot r unloads container c at location loc1
Precond: *rloc* = loc1, *cpos* = r
Effects: *cpos* ← loc1

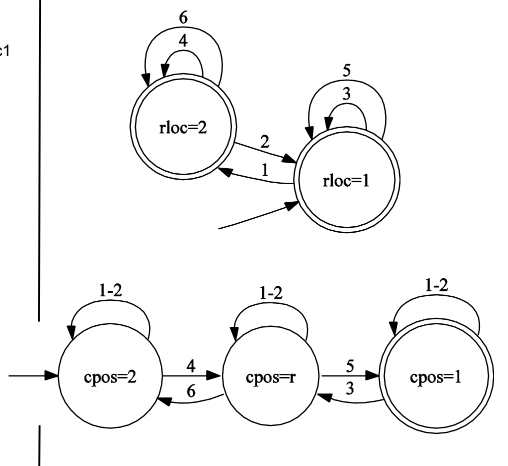
6 : unload(r, c, loc2)
;; robot r unloads container c at location loc2
Precond: *rloc* = loc2, *cpos* = r
Effects: *cpos* ← loc2

FSA-based

Automata

Following two automata correspond to the state variables of SAS^+ encoding: location of robot (r), and position of container (c)

The initial state of the automaton is marked with an incoming empty arrow. The accepting states are marked with the doubled circle.



$$\begin{tikzcd}
\text{rloc=2} \arrow[loop above, "4"] \arrow[loop above, "6"] \arrow[r, bend left, "2"] & \text{rloc=1} \arrow[l, bend left, "1"] \arrow[loop above, "3"] \arrow[loop above, "5"] & \arrow[l]
\end{tikzcd}$$

$$\begin{tikzcd}
\arrow[r] & \text{cpos=2} \arrow[loop above, "1-2"] \arrow[r, bend left, "4"] & \text{cpos=r} \arrow[l, bend left, "6"] \arrow[loop above, "1-2"] \arrow[r, bend left, "5"] & \text{cpos=1} \arrow[l, bend left, "3"] \arrow[loop above, "1-2"]
\end{tikzcd}$$

Fig. 1. Comparison of three different encodings of a simple instance of DWR domain.

Converting SAS$^+$ to FSA. Let V be a state variable and Act a set of ground actions from the SAS$^+$ planning task P. Then we can construct the corresponding DFA $A = \langle\Sigma, S, s_0, \delta, F\rangle$, where

- $\Sigma = Act$,
- $S = \mathrm{Dom}(V)$, i.e., the set of available values of V,
- s_0 is the evaluation of V in the initial state (note that by definitions we have full knowledge about the initial state),
- δ: $S \times \Sigma \rightarrow S$ such that $\forall s,r \in S, \forall a \in \Sigma$: $((s,a),r) \in \delta$ if and only if some of the following conditions holds:
    - i) $s = r$ and action a does not use the state variable V in its preconditions and effects (i.e., a is unrelated to V);
    - ii) $s = r$, the assignment $(V = s)$ is among the preconditions of action a, and a does not use the state variable V in its effects;
    - iii) the assignment $(V = r)$ is among the effects of action a, and a does not use the state variable V in its precondition (i.e., a transition is created from each state of V to the state r using action a);
    - iv) $s \neq r$, the assignment $(V = s)$ is among the preconditions of action a, and the assignment $(V = r)$ is among the effects of action a (i.e., only a single transition from s to r is created using the action a);
- $F = \{s \mid s \in S, (V = s) \in Goal(P)\}$ for the case when state variable V is constrained by the goal definition, otherwise $F = S$ (note that putting $F = S$ means precisely that we are not concerned with the final state of the automaton corresponding to V). ■

Using the algorithm described above we can transform each state variable into the corresponding automaton, and then run the resulting set of automata, $\Phi = \{A_1,\dots,A_k\}$, in parallel using as the input a sequence of actions $p = \langle a_1,\dots,a_n\rangle$ in order to determine whether p is a valid plan for a given planning problem P: p is a (not necessarily optimal) solution of P iff p is accepted by every automaton $A_i \in \Phi$. Hence, the search for a shortest plan for P corresponds to finding a shortest $p \in L$, $L = \Lambda(A_1) \cap \dots \cap \Lambda(A_k) = \Lambda(A_1 \cap \dots \cap A_k)$. Though it is not technically correct, shortly we will also denote L from the previous expression as $\Lambda(\Phi)$.

Thanks to the fact that modern constraint solvers, such as CLPFD library of SICStus Prolog (Carlsson, Ottosson and Carlsson 1997), are equipped with the constraints that model FSAs, the automata-based representation introduced above provides us with an easy-to-implement technique for solving planning problems. Our preliminary experiments show that automata-based constraint models exhibit comparable performance to the other existing constraint models for sequential optimal planning (Barták and Toropila, 2010).

The careful reader will notice that the automata-based representation is very related to the domain transition graphs (DTG) described in (Helmert 2006) or (Chen, Huang and Zhang 2008), however in this work we define the automata-based formulation in order to allow regular operations on automata that will be used in the following section.

FSA Construction

In the previous section we have introduced the FSA-based representation of a planning problem that is equivalent to given SAS$^+$ representation. However, the classical representation is still prevalent in a vast majority of available planning domains, and so it would be useful to have the technique that would convert a classical representation into an automata-based representation.

Generating Binary FSAs

Recall that the (ground) classical representation consists of a set of all possible facts (ground predicates) F, a set of facts I that hold in the initial state, a set of facts G that must hold in the goal state, and, finally, a set of actions A, where each $a \in A$ is a triple $(Prec, Eff^+, Eff^-)$ as described earlier.

Now, in the effort of converting the classical representation into the automata-based representation one can observe, that each fact $f \in F$ can be viewed as a special multi-valued variable V_f with the domain $\{0, 1\}$ corresponding to the false/true validity of the fact f. Intuitively, each action $a \in A$ can be transformed to operate with the set of state variables $\{V_f \mid f \in F\}$ in the following manner:

- each $p \in Prec$ is translated into the new precondition $(V_p = 1)$;
- each $e \in Eff^+$ is translated into the new effect assignment $(V_e = 1)$; and finally,
- each $e \in Eff^-$ is translated into the new effect assignment $(V_e = 0)$.

This way we obtain a primitive SAS$^+$ representation where all state variables have binary domains. Still, it is a valid SAS$^+$ representation, encoding of which can be used as the input for the SAS$^+$ to FSA conversion algorithm described in the earlier sub-section Representation Using FSAs. Figure 2 depicts the example of binary automata generated from facts describing the position of a container.

Let us now mention a few properties of such binary automata. Each binary automaton consists of two states $\{0, 1\}$ with the exception of automata corresponding to static facts (i.e., facts that are not influenced by any action; for example, adjacent(loc1,loc2)) which are degraded to an automaton consisting of a single state $\{0\}$ or $\{1\}$, depending on the validity of the given fact in the initial state. There are four possible transitions in a binary automaton: 0-0, 1-1, 0-1 and 1-0. The "loop" transitions 0-0 and 1-1 support the actions that either do not affect the given state variable V, or that have the assignment $(V = 0)$ or $(V = 1)$, respectively, among their preconditions. The

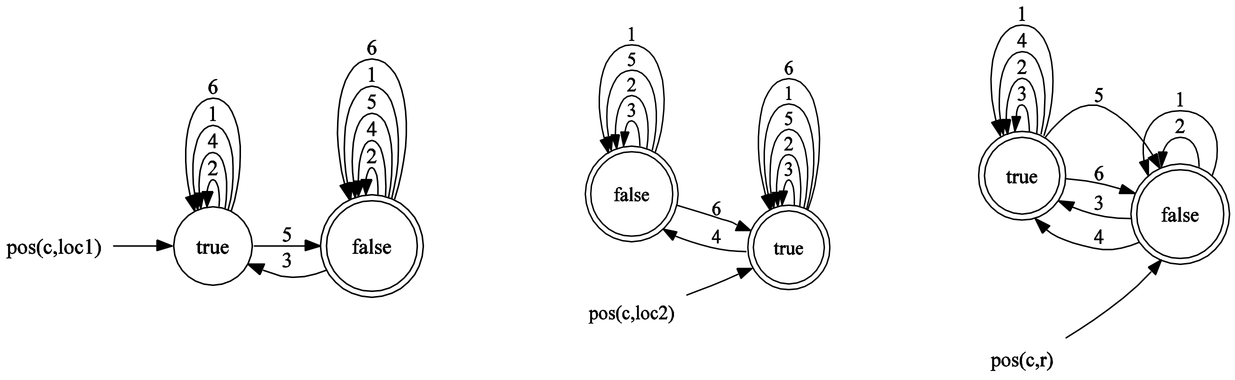


Fig. 2. Example of the binary automaton representation of the facts describing the position of a container.

"active" transitions 0-1 and 1-0, on the other hand, support only the actions that assign the value of the given state variable V to 0 or 1, respectively.

In general, the set F of all possible facts can contain many of them that are not reachable, i.e., there is no sequence of actions that would make a given fact valid. Hence, it might be a good idea to filter the unreachable facts out before translating them to binary automata, in order to minimize the size of the complete representation. One of the ways to implement the filtering is to build a planning graph (Blum and Furst 1997) all to way to the fixed point and consider only the set of facts from the last fact-layer of constructed graph.

Multi-Valued FSAs

Having the representation consisting only of binary FSAs is by itself, obviously, not of too much use. Not only there is large amount of automata in the representation, moreover this way the encoding does not capture the mutual exclusion between individual facts, which was one of the main motivations for introducing the SAS$^+$ formalism. However, we can employ the regular operation of automata intersection! Of course, whenever we create an intersection of a pair of automata that have n states each, the set of the states of the resulting FSA can have, potentially, the size of n^2. The worst case for merging k automata is then n^k states. Nevertheless, for the case when the two original automata encode correlated information that could be encoded using a single variable in SAS$^+$ encoding (such as the two automata created from facts location(robot,loc1) and location(robot,loc2)), the size of the state set of the resulting automaton becomes even smaller than the sum of the number of states in the original automata. That is the reason why we want to create the intersection of correlated automata, while keeping the independent ones separated. The example of "good" and "bad" intersection of binary FSA is depicted in Figure 3. Naturally, intersecting all the automata into one large FSA is intractable, since the resulting automaton would be, in fact, the explicit encoding of all possible states of the given planning problem.

For a given set of binary automata there is an exponential number of options of how to select the clusters of FSAs for intersection in order to obtain the compact encoding similar to SAS$^+$. There are, however, several heuristics that can help us with performing this task. First, in most cases we know the first-order predicates that were used for generating the ground facts. For example, it is a good idea to intersect the automata that correspond to facts location(robot,*l*) for each available location *l*. Another useful hint could be the number of actions on the "active" transitions that have two automata in common. As this is the work in progress, the performance of the possible automated approaches for generating compact multi-valued FSA-based encodings needs to be further examined in the near future.

Nonetheless, the situations also arise when the human-assisted generation of multi-valued automata-based representations can be well-suited for the task. Example of such need may be the modeling of a new planning domain, where the designer has some insight into which facts/properties could be merged together while, on the other hand, the system might help the designer to detect the correlations between the individual facts/properties.

Extracting SAS$^+$ from FSA

So far we have not mentioned the main motivation for generating automata-based encodings, which is, ultimately, a technique that would allow the generation of compact SAS$^+$ encodings of high quality. In fact, such a technique has been employed in Fast Downward planning system presented in (Helmert 2006) and has been, by now, the only available technique for transforming the existing classical logical representation into SAS$^+$ encoding. Nevertheless, in this paper we propose a different approach, which could serve as an alternative to the existing technique and would provide us with some advantages such as better control of the scaling of the transformation process details and straightforward communication with the user (for the case of human-assisted transformation). We consider these features to be useful especially for the planning domain modeling. Thus,

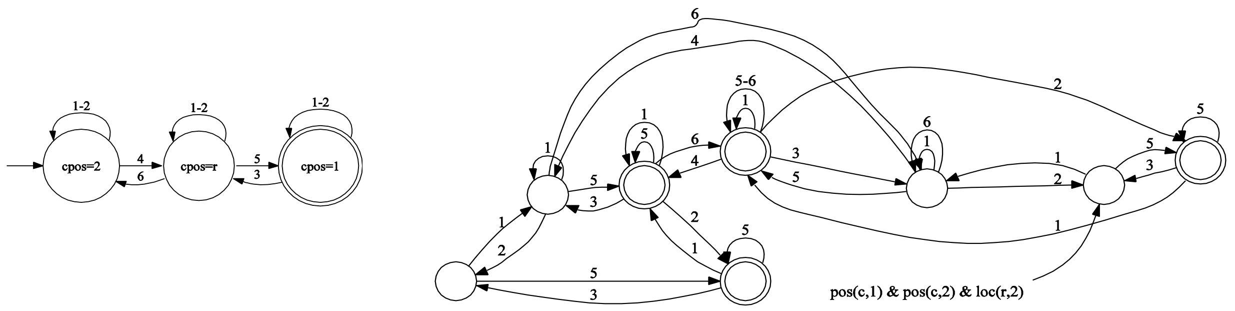


Fig. 3. Example of the good and bad intersection of binary FSAs. The automaton on the left was creating by intersecting pos(c,1) ∩ pos(c,2) ∩ pos(c,r), while the FSA on the right represents the intersection pos(c,1) ∩ pos(c,2) ∩ loc(r,2).

let us now describe how to derive a SAS$^+$ encoding from an existing multi-valued FSA-based representation.

In general, if we take an arbitrary set Φ of FSAs, it is not obvious how to convert it to the equivalent SAS$^+$ encoding of the planning problem (by equivalence we mean that $p \in \Lambda(\Phi)$ iff p is a valid plan for the SAS$^+$-encoded planning problem). It is probably not surprising that each automaton A_i will be converted to a state variable V_i, domain of which will be exactly the set of all states of A_i. More problematic part is the extraction of actions. In FSA, an action can be supported by multiple transitions, however in SAS+ we must explicitly specify the set of all variable evaluations that must hold prior to action execution, and a set of variable assignments that will hold after the action was executed.

Still, if we are lucky (or skillful) enough, we might obtain a set of FSAs, for which it is straightforward to extract an equivalent SAS$^+$ encoding. We will call such set of automata *SAS$^+$-compatible form*.

SAS$^+$-Compatible Form. We say that a set $\Phi = \{A_1,\dots,A_k\}$ of FSAs is in SAS$^+$-compatible form if for each automaton $A_i = <\Sigma, S_i, s_i, \delta_i, F_i>$ and each symbol (action) $a \in \Sigma$ one and only one of the following conditions holds:

i) $\forall s \in S_i$: $((s,a),s) \in \delta_i \wedge \forall s,r \in S_i, s \neq r$: $((s,a),r) \notin \delta_i$, i.e., action (symbol) a is supported only by all loop transitions, which means that a does not affect the state of A_i;

ii) $\exists s \in S_i$: $(((s,a),s) \in \delta_i \;\wedge\; \forall u,v \in S_i, ((u,a),v) \in \delta_i$: $u=v=s)$, i.e., action a is supported by one and only one loop transition, which means that a has state s among its preconditions, but otherwise does not affect the state of A_i;

iii) $\exists s \in S_i \;\; \forall r \in S_i$: $(((r,a),s) \in \delta_i \;\wedge\; \forall u,v \in S_i$, $((u,a),v) \in \delta_i$: $v=s)$, i.e., from each state $r \in S_i$ there is an transition to state s that supports action a and there are no other transitions supporting action a; which means that a has state s among its effects, but does not have any state of A_i among its preconditions;

iv) $\exists s,r \in S_i$: $(((s,a),r) \in \delta_i \;\wedge\; \forall u,v \in S_i, ((u,a),v) \in \delta_i$: $u=s \wedge v=r)$, i.e., there is one and only one transition that supports action a, which means that a has state s among its preconditions and state r among its effects. ■

Unfortunately, in many cases we might not be lucky enough to get a SAS$^+$-compatible set of automata. There is however a transformation procedure that converts an arbitrary set of FSAs into a SAS$^+$-compatible set. We need to ensure that exactly one of the conditions i) to iv) holds for each action a and each automaton A_i. In case none (or more) of the conditions holds, we will ensure the exclusive validity of condition iv) by renaming multiple occurrences of action a in the state transitions. Each transition $((s,a),r)$ in the rest of the automata will then be substituted with a set of transitions $\{((s,a_i),r) \mid a_i$ is a renamed version of $a\}$ Naturally, this will increase the total number of available actions exponentially in the worst case. Let us know describe the process of action renaming formally.

Constructing SAS$^+$-Compatible Form. Given a set $\Phi = \{A_i = <\Sigma, S_i, s_i, \delta_i, F_i> \mid 1 \leq i \leq k\}$ of FSAs we construct a set $\Phi' = \{A_i' = <\Sigma', S_i, s_i, \delta_i', F_i> \mid 1 \leq i \leq k\}$ in SAS$^+$-compatible form by performing the following steps:

- For each action (symbol) $a \in \Sigma$ and each A_i, if a breaks in A_i the definition of SAS$^+$-compatibility we define $\tau(a,i)$ as a set of state transitions that support action a in an automaton A_i, otherwise we define $\tau(a,i) = \{1\}$.
- For each $a \in \Sigma$ we create a set Θ_a of renamed versions of a, where $\Theta_a = \{a_{n1,\dots,nk} \mid \forall i, 1 \leq i \leq k\text{: } ni \in \tau(a,i)\}$.
- For each automaton A_i we create its copy A_i', that we will further modify.
- For each $a \in \Sigma$ and each A_i',
  i) if a breaks in A_i the definition of SAS$^+$-compatibility, then replace every transition $t = ((u,a),v) \in \delta_i'$ with transitions in a set of renamed transitions $\{((u,a_{\dots,ni=t,\dots}),v) \mid a_{\dots,ni=t,\dots} \in \Theta_a\}$;
  ii) otherwise replace every transition $t = ((u,a),v) \in \delta_i'$ with transitions in a set of renamed transitions $\{((u,a_{\dots,ni=1,\dots}),v) \mid a_{\dots,ni=1,\dots} \in \Theta_a\}$. ■

Note that if an action a conforms to the definition of SAS$^+$-compatibility, then the set Θ_a contains only a single element, i.e., there is no need for renaming this action.

Now we can finally describe the procedure of converting an arbitrary set of FSAs into an equivalent SAS$^+$-encoded planning problem.

Converting FSA to SAS$^+$. Given a set of deterministic finite automata $\Phi = \{A_1,\ldots,A_k\}$ over an input alphabet Σ we first construct a set $\Phi' = \{A_1',\ldots,A_k'\}$ in SAS$^+$-compatible form over an extended alphabet Σ'. Then we can construct a SAS$^+$ encoding $\Xi = <Var, Act>$ of an equivalent planning problem, where

- $Var = \{V_i \mid 1 \leq i \leq k\}$ is a set of state variables, with V_i corresponding to the automaton A_i',
- $\forall V_i \in Var$: $\mathrm{Dom}(V_i) = \mathrm{States}(A_i')$,
- for each $a \in \Sigma'$ we define $Prec(a) = \{(V_i = s) \mid a$ satisfies the condition ii) or iv) of the SAS$^+$-compatibility definition, and $((s,a),r) \in \delta_i'$ for an arbitrary $r\}$,
- for each $a \in \Sigma'$ we define $Eff(a) = \{(V_i = r) \mid a$ satisfies the condition iii) or iv) of the SAS$^+$-compatibility definition, and $((s,a),r) \in \delta_i'$ for an arbitrary $s\}$,
- $Act = \{<Name(a), Prec(a), Eff(a)> \mid a \in \Sigma'\}$. ■

Summary

The paper proposed the novel encoding for classical planning task using a set of deterministic finite automata that might provide a new platform for development of modern planners. The transformation methods have been shown for converting both SAS$^+$ and classical representations into the automata-based representation. The conversion from the classical representation that leverages from the regular operation of automata intersection will be the subject of further research in order to obtain fully automated technique for generating the multi-valued automata-based encodings. Last, but by no means least, the paper presented the algorithm for translating an arbitrary set of deterministic finite automata into the SAS$^+$-based encoding of the equivalent planning problem, providing thus the theoretical foundations for the alternative translation technique from the classical representation into the SAS$^+$ encoding.

Acknowledgements

The research is supported by the Czech Science Foundation under the projects no. 201/09/H057 and P103/10/1287 and by the Charles University Grant Agency under the contract no. 9710/2010.

References

Bäckström, Ch., Nebel, B. 1995. Complexity results for SAS$^+$ planning. *Computational Intelligence* 11(4), 625-655.

Barták, R., Toropila, D. 2010. Solving Sequential Planning Problems via Constraint Satisfaction. *Fundamenta Informaticae*, Volume 99, Number 2, IOS Press, 125-145.

Blum, A. and Furst, M. 1997. Fast planning through planning graph analysis. *Artificial Intelligence* 90, 281-300.

Carlsson, M., Ottosson, G., Carlson B. 1997. An Open-Ended Finite Domain Constraint Solver. *Programming Languages: Implementations, Logics, and Programs*.

Carrol, J., Long, D. 1989. *Theory of finite automata: with an introduction to formal languages*. Prentice Hall.

Chen, Y., Huang, R., Zhang, W. 2008. Fast Planning by Search in Domain Transition Graphs. *Proceedings of the AAAI Conference on Artificial Intelligence* (AAAI-08).

Huang, R., Chen, Y., Zhang, W. 2010. A Novel Transition Based Encoding Scheme for Planning as Satisfiability. *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligenc*e (AAAI-10), AAAI Press, 89-94.

Ghallab, M., Nau, D., Traverso P. 2004. *Automated Planning: Theory and Practice*. Morgan Kaufmann.

Helmert, M. 2006. The Fast Downward Planning System. *Journal of Artificial Intelligence Research 26*, 191-246.

Pralet, C., Verfaillie, G. 2009. Forward Constraint-Based Algorithms for Anytime Planning. *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling* (ICAPS), AAAI Press, 265-272.

Sipser, M. 2006. *Introduction to the Theory of Computation, Second Edition*, Thomson Course Technology.