KMONITOR – A Tool for Monitoring Plan Execution in Action Theories ⋆

Thomas Eiter, Michael Fink, and Ján Senko

Vienna University of Technology, Institute of Information Systems, Vienna, Austria
e-mail: {eiter, michael, jan}@kr.tuwien.ac.at

Abstract. We present a monitoring tool for plan execution in non-deterministic environments, which are described in an action language, based on non-monotonic logic programming. Thanks to it, deviations of concrete executions from expected ones can be detected, and diagnostic explanations in terms of unsuccessful action executions can be obtained. The latter may be exploited for execution recovery, and may help in rectifying an incoherent view of the planning domain.

1 Introduction

In a non-deterministic environment, an agent's plan for achieving a goal by taking a sequence of actions might fail, if some of the actions do not materialize as expected. For this reason, the plan execution might be monitored in order to detect an execution failure or potential problems at an early stage, from which the agent may then recover. Execution monitoring was considered for logical domain descriptions in Golog [7, 8] and Flux [6], and for the action language $\mathcal{AL}$ in the APLAgent Manager [1, 2]. To our knowledge it has not been considered for other KR action languages such as $\mathcal{C}$, or $\mathcal{K}$, and in particular for non-deterministic domains. In [4, 3], a general monitoring approach for logic-based action languages with transition-based semantics is presented, in which it is checked from time to time whether the current state complies with a set $\mathcal{T}$ of trajectories which describe the expected executions of the plan. If a discrepancy is detected, then the execution is not on track and the agent might suitably reconsider it; in order to diagnose discrepancies, points of failure in the execution are computed, which informally explain discrepancies applying Occam's Razor by the latest action execution which might have resulted in a "bad" outcome. Such information is useful for execution recovery, e.g., if actions are undone [3], but also for checking whether the user's understanding of the domain is coherent with the formalization.

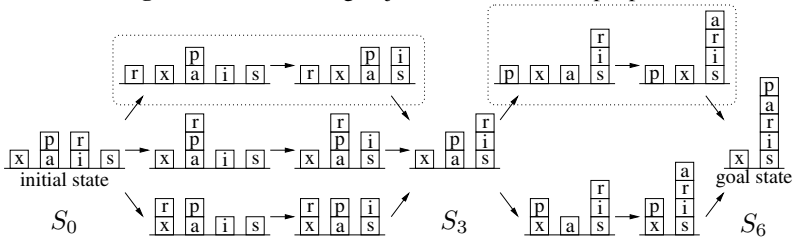
Example 1. As a running example, we consider here a variant of the well-known Blocks World domain, in which a block being moved may end up at a location different from the intended one, because the agent might not grip it properly. Suppose we have the blocks $\mathtt{a}, \mathtt{i}, \mathtt{p}, \mathtt{r}, \mathtt{s}, \mathtt{x}$, and the plan

$$P = \langle \mathtt{move}(\mathtt{r}, \mathtt{x}), \mathtt{move}(\mathtt{i}, \mathtt{s}), \mathtt{move}(\mathtt{r}, \mathtt{i}), \mathtt{move}(\mathtt{p}, \mathtt{x}), \mathtt{move}(\mathtt{a}, \mathtt{r}), \mathtt{move}(\mathtt{p}, \mathtt{a}) \rangle$$

for reaching the goal state S_6 from the initial state S_0 in 6 steps as in Figure 1, which shows all trajectories for P that establish the goal. If now at stage 4 the discrepancy is detected that p is on r, then the execution will fail, since the next action $\mathtt{move}(\mathtt{a}, \mathtt{r})$ can not be taken. If all other blocks are situated as in S_3, an explanation is that the preceding action $\mathtt{move}(\mathtt{p}, \mathtt{x})$ has failed (see [4, Ex. 3] for a formal description).

⋆ This work was supported by the Austrian Science Fund (FWF) under grant P16536-N04.

Fig. 1. Goal-establishing trajectories for the example plan.



In general, not all goal-establishing trajectories might be equally desired, and some preferred over others. To model this, $\mathcal{T}$ contains all preferred trajectories.

Example 2. In our example, let the preferred trajectories $\mathcal{T}$ be those in which no block unintentionally falls on the table during execution. Hence, from the goal-establishing trajectories in Figure 1, those which pass through the dotted area are not preferred.

KMONITOR (*http://www.kr.tuwien.ac.at/research/monitoring*) implements the execution monitoring approach of [4, 3] for the action language $\mathcal{K}$ on top of the DLV$^{\mathcal{K}}$ planning system [5]. Fig. 2 shows the main loop, which is entered when monitoring is issued. To keep the monitoring overhead low, the current state is analyzed only at certain *checkpoints*, which are determined by a respective component from a *checkpoint policy* specified by a non-monotonic logic program (see Section 2). State analysis is done by the tool KDIAGNOSE implementing the diagnosis method from [4]. If a discrepancy is detected, control is returned with this information and any diagnoses found.

Example 3. Suppose that blocks `x`, `p`, `r` and `s` are known to be heavy, and that the policy is to check each time when a heavy block was moved (as such moves bear high likelihood of failure). Then, the checkpoints would be the stages 1, 3, 4, and 6.

2 Checkpointing

Rather than checking for a discrepancy after each step, we may check only at certain stages, e.g., if a stage has higher likelihood of failure, or do a periodic check. Then monitoring can be less intrusive and the execution of the whole plan will be faster.

In order to select stages for a discrepancy check, a "checkpointing policy" is specified in terms of a logic program, which has facts and rules over the sets of fluents and actions from the domain. The policy is evaluated for all stages in the plan, and if a fact $\texttt{checkpoint}(t)$ is true for the current stage, then a check for discrepancy is issued.

We distinguish two types of checkpointing policies – static and dynamic ones. In the static case, checkpoints are calculated one and for all from the policy logic program and the plan. In the dynamic case, a part of the current state is taken into consideration, yielding more expressive power, but also the need for re-evaluating checkpoints. Figure 3 gives an overview of the checkpoint generation utility.

Static checkpoints. When using a static checkpointing policy, we compute the model of a logic program, comprised of the checkpoint definition file and the modified plan. Actions of the plan are rewritten into facts with timestamps. (E.g., an action $\texttt{move}(\texttt{a}, \texttt{r})$ occurring at stage 2 is rewritten into $\texttt{move}(\texttt{a}, \texttt{r}, 2)$.) The checkpoint definition file contains rules over the modified action predicates and the special predicate $\texttt{checkpoint}(t)$, whose truth value defines whether we have to do a check at stage t or not.

Fig. 2. Monitoring Loop in KMONITOR.

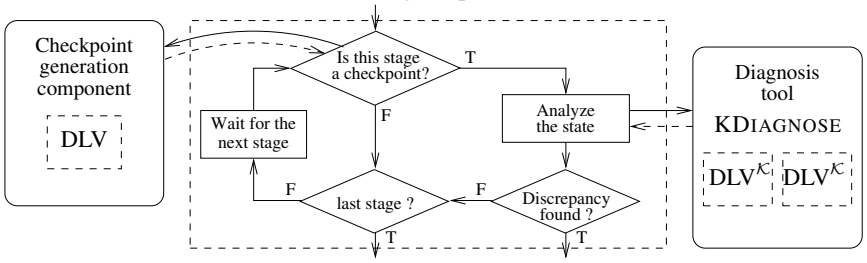
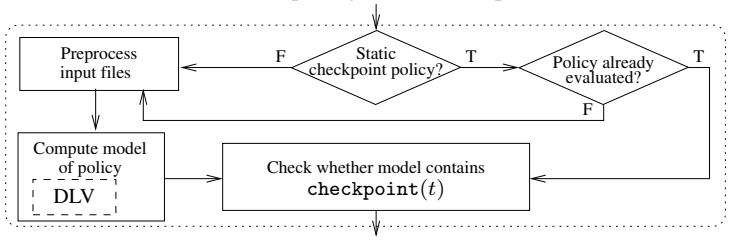


Fig. 3. Checkpoint generation component.



Example 4. To force a check after each move involving a heavy block, we can define the following static checkpointing policy:

$$\texttt{heavy(X)}. \qquad \text{for } \texttt{X}\in\{\texttt{x},\texttt{p},\texttt{r},\texttt{s}\}, \text{ and}$$
$$\texttt{checkpoint(Tj)} \ :- \ \texttt{move(X,Y,Ti)},\ \texttt{heavy(X)},\ \texttt{Tj} = \texttt{Ti} + \texttt{1}.$$

Dynamic checkpoints. A dynamic checkpointing policy also involves information about some fluents of the current state; intuitively, they are sensed to steer the checkpointing. Therefore, we need to find models of the logic program – made up from the checkpoint declaration file, the plan, and partial state information – at each stage. At stage t we compute a model of this program, and if $\texttt{checkpoint}(t)$ is true, a check for discrepancy is executed. The current stage t may be accessed via $\texttt{now}(t)$.

Example 5. For issuing a check after each step in which a block ends on a different location than intended, we can use the following dynamic checkpointing policy:

$$\texttt{checkpoint(Tj)} : -\texttt{move(X,Y,Ti)}, \texttt{not on(X,Y)}, \texttt{Tj} = \texttt{Ti} + \texttt{1}.$$

This policy will be evaluated at each stage and its result depends on the current state.

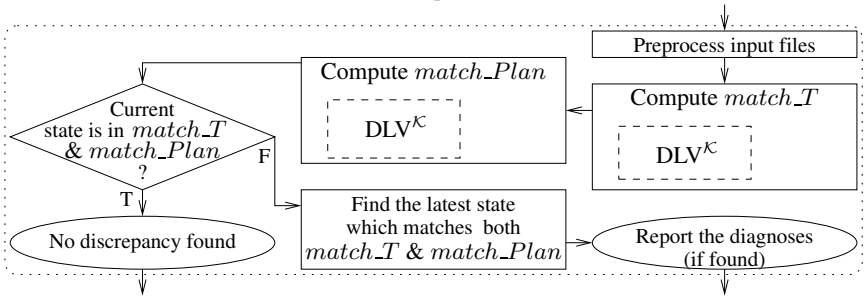
In a "sleep" mode, policy re-evaluation can be suppressed until the next provisional checkpoint according to the last evaluation.

3 Diagnosis of Discrepancies

Before we detail KDIAGNOSE, we informally introduce some terminology (cf. [4]).

There is a *discrepancy* between a state S_i and a set of preferred trajectories $\mathcal{T}$ relative to a plan P, if there is no trajectory $S'_0, A_0, S'_1, \ldots, S'_{n-1}, A_{n-1}, S'_n$i.e., an alternating sequence of states and action occurrences in $\mathcal{T}$, such that $S'_i = S_i$ and S'_n is a goal state. Furthermore, we identify the point of failure for an observed discrepancy by finding the latest time stamp after which every evolution of S_i deviates from every trajectory in $\mathcal{T}$. Thus, a pair (S_k, k) is a *point of failure*, or a *diagnosis*, if:

Fig. 4. KDIAGNOSE components and control flow.



(S1) Some evolution of S_i matches a goal-establishing trajectory in $\mathcal{T}$ at stage k ($0 \leq k < i \leq n$) in state S_k and deviates at stage $k+1$.
(S2) No evolution of S_i matches a goal-establishing trajectory in $\mathcal{T}$ at stage $k' > k$.

Example 6. If, in our running example, (i) at stage 1 block r is on the table, or (ii) $\texttt{on}(\texttt{p},\texttt{r})$ holds at stage 4 while the remaining blocks are as depicted in S_3, then in both cases we cannot find a preferred trajectory with a corresponding state, i.e., we detect a discrepancy. For (i) still a feasible trajectory to the goal exists (stages 1 and 2 along the dotted area), not so for (ii). Thus, discrepancy (i) cannot be detected in, or after, stage 3. Observe also that $(S_3, 3)$ is the only diagnosis for (ii) and no diagnosis exists for (i).

KDIAGNOSE detects discrepancies and, if found, computes all diagnoses as follows (cf. Figure 3). Besides P, $\mathcal{T}$, and S_i it takes a domain description and the planning problem as inputs. In a preprocessing step, the current state and the planning problem are transformed into slightly modified planning problems for calculating

- the set $match\_T$ of all goal-establishing preferred trajectories, and
- the set $match\_Plan$ of all evolutions leading to S_i according to P.

Clearly, if S_i is on a goal-establishing preferred trajectory of $match\_T$ (and thus also in $match\_Plan$), then there is no discrepancy. Otherwise, KDIAGNOSE computes all diagnoses by comparing the above sets and searching maximal states S_j at which possible evolutions and goal-establishing preferred trajectories coincide.

4 Implementation

Our implementation of KMONITOR builds on KDIAGNOSE, which uses the DLV$^{\mathcal{K}}$ system to compute diagnoses for a given state. It invokes the diagnosis tool sequentially at all stages singled out by the checkpoint policy. The inputs to KMONITOR are:

- `plan` - a plan (in DLV$^{\mathcal{K}}$syntax) to monitor for discrepancies;
- `checkpoints.dl` - the checkpoint definition file;
- `background.dl` and `K.plan` - the domain description and planning problem;
- `T.plan` - a list of preferred trajectories, or alternatively a (modified) planning problem defining preferred trajectories (by its solutions);
- `state.*`, `cstate.*` - state & checkpointing information about the plan execution.

The checkpoint generation component is invoked as described in Section 2. Checkpoint computation is accomplished by computing models using the DLV system. For

static policies, facts `checkpoint`(t) are extracted into a temporary file, and at the respective stages KDIAGNOSE is invoked for state analysis. A state t of a concrete partial execution (run) of the plan is fetched from file `state.`t. For simulation, a run may be automatically generated using DLV$^{\mathcal{K}}$(e.g., by randomly generating a trajectory for a modified planning problem). In the dynamic case, checkpointing state information is fetched from file `cstate.`t. If state information is missing, checkpointing is skipped.

Example 7. The static checkpoint policy of Ex. 4 yields `checkpoint`(t) , $t \in \{1, 3, 4, 6\}$. For a plan execution as in Ex. 6 (ii), KMONITOR reports a discrepancy for state S_4:

```
> No error at stage 1.
> No error at stage 3.
> Error at stage 4 - Point of failure at stage 3.
> State info: {on(x,table), on(a, table), on (p,a), ...
```

The dynamic policy of Ex. 5 would yield a run with the same result, but the check point policy would be evaluated at each step, and a check would occur only at stage 4. Note that if at stage 1 block r would unintentionally end on block p, then we would have another check but would not detect any discrepancy. We also remark that by simple refinements of our dynamic policy we could avoid evaluation at each step.

5 Conclusion and Future Work

KMONITOR is an execution monitoring tool for non-deterministic domains utilizing action language $\mathcal{K}$. It detects deviations from intended execution paths and computes explanations for discrepancies. By means of a checkpointing policy, discrepancy checking and diagnosis can be restricted to certain stages of the execution. A detailed comparison with [1] is left for a longer version of the paper. For a comparison and further references to related work on diagnosis in answer-set programming, the reader is referred to [4].

Currently KDIAGNOSE handles no concurrent actions, which however is easy to overcome. Generalizing [4], it shall also support diagnosis w.r.t. partial state information restricted to a focus of interest. Finally, we plan to extend KMONITOR towards a system capable of giving recovery support as well. To this end, implementing and integrating techniques for recovery as described e.g. in [3] is envisioned.

References

1. Balduccini, M.: APLAgent Manager. krlab.cs.ttu.edu/~marcy/APLAgentMgr/index.html.
2. Balduccini, M., Gelfond, M.: Diagnostic reasoning with A-Prolog. Theory and Practice of Logic Programming, 3(4–5), 425–461. 2003.
3. Eiter, T., Erdem, E., Faber, W.: Plan Reversals for Recovery in Execution Monitoring. In *Proc. NMR*, 147–154. 2004.
4. Eiter, T., Erdem, E., Faber, W., Senko, J.: A logic-based approach to finding explanations for plan execution discrepancies. TR INFSYS RR-1843-04-03. 2004.
5. Eiter, T., Faber, W., Leone, N., Pfeifer, G., Polleres, A.: A logic programming approach to knowledge-state planning, II: The DLV$^{\mathcal{K}}$ system. AI, 144(1–2), 157–211. 2002.
6. Fichtner, M., Großmann, A., Thielscher, M.: Intelligent execution monitoring in dynamic environments. Fundamenta Informaticae, 57(2–4), 371–392. 2003.
7. Giacomo, G. D., Reiter, R., Soutchanski, M.: Execution monitoring of high-level robot programs. Proc. of KR'1998, 453–465. 1998.
8. Soutchanski, M.: High-level robot programming and program execution. Proc. of ICAPS Workshop on Plan Execution. 2003.