Expert Systems With Applications 334 (2027) 134509

Contents lists available at ScienceDirect

Expert Systems With Applications

journal homepage: www.elsevier.com/locate/eswa

RVREPLAN: A runtime verification framework for adaptive replanning

Rafael C. Cardoso [a,*], Angelo Ferrando [b]

[a] Department of Computing Science, University of Aberdeen, Aberdeen, AB11 7DF, UK
[b] Department of Physics, Informatics and Mathematics, University of Modena and Reggio Emilia, Modena, 41125, Italy

ARTICLE INFO

Keywords:
Runtime verification
Classical planning
Replanning
Action specification

ABSTRACT

Classical planning systems rely on action models that describe the expected preconditions and effects of actions. In realistic execution settings, these models may become incomplete, outdated, or inconsistent with observed behaviour, causing plans to fail at runtime. This paper presents RVREPLAN, an intelligent framework for runtime adaptation of planning action models. Given a planning domain and problem specifications, RVREPLAN automatically generates an executable simulation environment and synthesises runtime monitors for action preconditions and postconditions. These monitors analyse execution traces to detect missing expected preconditions and effects, while additional observed effects are recorded and used to diagnose subsequent violations when they affect the execution of the current plan. When such a violation is detected, RVREPLAN reasons about its likely cause, updates the relevant action specification when supported by the observations, validates the consequences of the candidate update in a simulated environment, and triggers replanning when necessary. We evaluate the framework on established International Planning Competition domains using multiple planner configurations. The results analyse monitor synthesis time, replanning time, plan size, and plan stability, showing that runtime verification can support explainable and systematic adaptation of planning models during execution. The proposed framework contributes a domain-independent mechanism for improving the robustness of intelligent planning systems at runtime.

1. Introduction

Automated planning employs search algorithms and heuristics to find a sequence of actions that achieves a desired goal from an initial state, given a model of the world and state transition functions (Nau et al., 2004). Classical planning, largely inspired by the STRIPS (STanford Research Institute Problem Solver) formalism (Fikes & Nilsson, 1971), assumes that states are fully observable and that actions are deterministic.

In practice, planning models are abstractions of the real world and may be imprecise due to human error or outdated knowledge. When such discrepancies arise, the plan may no longer be valid and is likely to fail at runtime. Detecting and handling mismatches between the expected and actual effects of actions is crucial for ensuring the reliable execution of plans. This is typically addressed through *replanning*, which creates a new plan with updated information, or through *plan repair*, which modifies an existing plan.

The first challenge is to detect the problem. We use *Runtime Verification* (RV), a formal technique that employs monitors to observe execution traces and check them against temporal properties (Bartocci et al., 2018). RV allows us to identify violations of action specifications during execution and automatically update the domain description to reflect these changes.

We propose RVREPLAN, a general-purpose, domain-independent framework that integrates RV with classical planning. It uses automatically synthesised RV monitors to detect imprecise or outdated action abstractions at runtime. These monitors share global knowledge of the system, initially consistent with the planner's abstract model, and record changes observed during execution in a shared data structure that informs domain updates and replanning.

The central novelty of RVREPLAN is a framework that connects runtime monitoring, failure diagnosis, action-model repair, validation, and replanning. Existing replanning approaches generally revise the current problem state while retaining the original action schemas. Action-model acquisition approaches can learn or refine schemas from execution traces, but are not usually integrated with formal runtime monitoring and immediate replanning decisions. Previous work on monitor synthesis for planning focused on detecting failures and did not revise action specifications or reason about how execution should proceed after a violation (Ferrando & Cardoso, 2022).

RVREPLAN combines these previously separate capabilities. It automatically derives temporal monitors from PDDL action schemas, uses

* Corresponding author.
E-mail addresses: rafael.cardoso@abdn.ac.uk (R.C. Cardoso), angelo.ferrando@unimore.it (A. Ferrando).

https://doi.org/10.1016/j.eswa.2026.134509
Received 10 May 2026; Received in revised form 5 September 2026; Accepted 21 September 2026
Available online 24 September 2026
0957-4174/© 2026 The Authors. Published by Elsevier Ltd. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

their verdicts to identify discrepancies between expected and observed action behaviour, repairs the affected operator effects, evaluates the consequences of the candidate repair in a simulated symbolic environment, and decides whether execution can continue or replanning is required. The contribution is therefore not the individual use of planning, temporal logic, or simulation, but their integration into a reusable methodology for maintaining alignment between a planning model and the system being executed. This avoids the need to implement domain-specific diagnosis and recovery rules for each possible action failure.

More specifically, this work makes the following contributions:

- an automated method for synthesising precondition and postcondition monitors from parameterised action schemas;
- a diagnosis mechanism that detects missing expected preconditions and effects and uses recorded additional effects to determine whether a later violation can be attributed to an earlier action;
- an online action-model repair mechanism that updates operator effects from monitored execution observations;
- a symbolic validation and replanning procedure that evaluates candidate repairs before they are used to guide continued execution; and
- a domain-independent implementation and empirical evaluation across multiple planning domains, monitor variants, and planner configurations.

2. Background

To explain the synthesis of the monitors and the other components of RVREPLAN, we first briefly recall the basic definitions of the STRIPS-like planning fragment used by RVREPLAN and RV. We use the term "STRIPS-like" because this fragment retains deterministic action schemas with conjunctive preconditions and add/delete effects, but extends the original STRIPS formulation by allowing explicit negative preconditions. In practice, RVREPLAN consumes Planning Domain Definition Language (PDDL) domain and problem specifications restricted to this fragment.

2.1. Planning

Definition 1. A STRIPS-like classical planning problem is a tuple $\langle A, S_0, G^+, G^- \rangle$, where A is a set of action schemas whose ground instances define the available actions, S_0 is the initial state with a set of propositions, and G^+ and G^- are disjoint sets of goal propositions describing the propositions required to be true and false (respectively) in the goal state.

Each state is a finite set of ground propositions drawn from a vocabulary of predicate symbols.

The output of a planner is a *plan* containing a sequence of actions $a_1, \ldots, a_n$. The state transition function $\gamma(S_0, a_1)$ applies action a_1 to state S_0. To successfully execute the plan, we require that $S = \gamma(\cdots \gamma(\gamma(S_0, a_1), a_2) \cdots, a_n)$ and $(G^+ \subseteq S) \wedge (G^- \cap S = \emptyset)$; that is, after applying the state transitions associated with each action in the plan, we expect state S to contain all goal propositions required to be true (G^+) and none of the goal propositions required to be false (G^-).

Definition 2. An action schema (*i.e.*, planning operator) specification is a tuple

$$a_i(\vec{x}) = \langle \alpha^+_{a_i}, \alpha^-_{a_i}, \beta^+_{a_i}, \beta^-_{a_i} \rangle,$$

where $\vec{x} = \langle x_1, \ldots, x_n \rangle$ is the tuple of parameters of the action schema. The sets $\alpha^+_{a_i}$ and $\alpha^-_{a_i}$ are the disjoint positive and negative preconditions of $a_i(\vec{x})$, respectively, and $\beta^+_{a_i}$ and $\beta^-_{a_i}$ are the disjoint positive and negative effects of $a_i(\vec{x})$.

In the remainder of the paper, we distinguish between action schemas and ground action occurrences. An action schema, denoted $a_i(\vec{x})$, is a parameterised planning operator with variables $\vec{x}$. A ground action occurrence, denoted a_i, is an instantiated occurrence of an action schema appearing in a plan, obtained by replacing the variables $\vec{x}$ with objects from the problem domain. The set A denotes the set of action schemas in the domain, while a plan is a sequence of ground action occurrences.

For readability, when a ground action occurrence is clear from the context, we may refer to it simply as an *action*. Moreover, when we write preconditions or effects of a ground action occurrence, such as $\alpha^+_{a_i}$, $\alpha^-_{a_i}$, $\beta^+_{a_i}$, or $\beta^-_{a_i}$, we mean the corresponding preconditions or effects of its associated action schema, instantiated with the objects of a_i. Similarly, when an algorithm updates the preconditions or effects of an action, the update is applied to the corresponding action schema in the domain specification.

Action schemas are interpreted under standard deterministic planning semantics. Each action schema defines a parameterised template with preconditions and effects over literals; its grounded instances form the actions that can appear in a plan. Automated planners use as input *domain* and *problem* specifications and output a *plan*. A domain specification contains the description of the action schemas, while the problem specification defines the initial state and the goals of the planning problem.

For illustration, suppose that the ground action `load(pkg1,truck1,loc1)` requires both the package and the truck to be at location `loc1` beforehand. Using propositions, this can be expressed as

$$\texttt{load(pkg1,truck1,loc1)} \rightarrow \ominus(\texttt{at(pkg1,loc1)}^+ \wedge \texttt{at(truck1,loc1)}^+).$$

This states that when the ground action occurs, both positive precondition observations were held in the previous trace event. Algorithm 1 later generalises this idea using the *since* operator to track the most recent observation of each proposition.

2.2. Runtime verification

The conventional formalism for expressing formal properties in RV is propositional Linear-time Temporal Logic (LTL) (Pnueli, 1977). In the context of propositional LTL properties, a monitor can be synthesised as a Finite State Machine (FSM) (Bauer et al., 2011). When given a trace, where each element is a proposition, the FSM provides a verdict of $\top$ or $\bot$ to indicate satisfaction or violation of the LTL property, respectively.[1]

LTL is traditionally limited to future modalities.Extending it with past operators (Manna & Pnueli, 1992) enables the specification of properties that are often more concise and intuitive for runtime verification (Lichtenstein et al., 1985). The syntax of Linear Temporal Logic restricted to past-time operators (Past LTL) is defined as follows Manna and Pnueli (1989):

$$\varphi = \text{true} \mid \text{false} \mid a \mid p^+ \mid p^- \mid (\varphi \wedge \varphi') \mid (\varphi \vee \varphi') \mid \neg\varphi \mid (\varphi \; S \; \varphi') \mid \ominus\varphi$$

Here, a denotes a ground action occurrence, p^+ and p^- are positive and negative ground propositions, φ is a formula, S stands for *since*, and $\ominus$ for *previous-time*. We also use $(\varphi \rightarrow \varphi')$ as shorthand for $(\neg\varphi \vee \varphi')$, and define $H\varphi$ (*history*) as $\neg(\mathit{true}\; S\; \neg\varphi)$. Using these operators, we can

[1] While additional verdicts can be employed, for our focus on safety properties, we are primarily concerned with $\bot$ to identify violations. For more in-depth discussions, refer to Bauer et al. (2010).

express runtime constraints over the preconditions and effects of actions in classical planning.

3. Related work

Temporal logic has been used in the past as a formalism for representing and reasoning about actions and change. In Calvanese et al. (2002), for example, LTL action theories are introduced in which the behaviour of a dynamic system and the effects of actions are specified using LTL. They study reasoning about action effects and executability, as well as the generation of conformant plans for temporally extended goals under incomplete information. In contrast, our work does not use LTL as the primary action-description formalism or as the language in which planning goals are expressed. RVReplan starts from PDDL action specifications and automatically derives temporal properties for runtime verification. The resulting monitors observe an executing system, detect discrepancies between expected and observed action behaviour, and guide the online update of action specifications and subsequent replanning.

The seminal work by Fox et al. (2006) introduced *plan stability* as a metric for evaluating plan repair and replanning, measuring the deviation between the original and new plans. Stability is desirable in many domains as it promotes consistent and safe behaviour. Building on this idea, *plan commitment* was proposed in Babli et al. (2023), a related metric that quantifies the impact of replanning on other agents—particularly relevant in multi-agent settings where one agent's changes can disrupt others.

Other approaches address related challenges from different perspectives. In Aineto et al. (2023) *resilient planning* was introduced, which produces plans that can tolerate up to K execution failures. Their approach assumes that failed actions cannot modify the world state and cannot be re-executed. In contrast, our work explicitly relies on specification violations to modify the state, allowing these changes to be observed and used to update the corresponding action models.

MIST (Fickert et al., 2021) is a technique for replanning over multiple potential initial states using time-aware heuristics. Their replanning is triggered by the addition of new goals, not by execution failures. Similarly, in Fritz and McIlraith (2007), the situation calculus was used to monitor plan validity and optimality in dynamic environments, thereby avoiding unnecessary replanning when the plan remains valid. Unlike these approaches, our work uses formally specified monitors to verify the planner's assumptions about action behaviour and to trigger domain updates when these assumptions are violated.

Action schema repair in RVReplan is closely related to action model acquisition, which seeks to infer action preconditions and effects from observations of plan execution (Arora et al., 2018). Approaches in this area differ in the information available in their traces and in whether learning is performed before or during execution. For example, FAMA learns STRIPS action models from observations of plan executions and can operate when actions or intermediate states are only partially observable (Aineto et al., 2019). Lamanna et al. (2021) instead learn PDDL action models incrementally during plan execution, selecting goals that produce informative traces. Stringer et al. (2023) similarly collect information about action performance at runtime, synthesise revised action descriptions, and use them to reconfigure the plans of BDI agents.

RVReplan shares with these approaches the use of observed state transitions to infer discrepancies in action behaviour. However, its objective is not to acquire a complete action model from a collection of traces. It begins with an existing PDDL domain model and combines automatically synthesised runtime monitors with recorded state observations to identify execution-relevant mismatches between the expected and observed behaviour of actions. Under the assumptions of full observability and deterministic action execution, the framework performs a local repair of the affected action effects by removing an expected effect that was not observed or adding an unexpected effect that was observed. The revised specification is then evaluated in the simulated environment and immediately used to determine whether execution can continue, or replanning is required.

Consequently, action model acquisition and the repair performed by RVReplan are complementary. Acquisition methods can construct or refine models from evidence accumulated across multiple traces, potentially accounting for incomplete observations and repeated executions. In contrast, RVReplan focuses on detecting and responding to an execution-time violation to support continued operation.

We do not claim that runtime repair with RVReplan is generally superior to offline or online action model acquisition. The approaches address different but complementary objectives. Action model acquisition aims to construct or refine a model from evidence accumulated across one or more traces, whereas RVReplan assumes that an initial model is already available and focuses on responding to a specific discrepancy during execution. Acquisition techniques may provide more reliable model revisions by considering repeated observations, while the local repair used by RVReplan can support an immediate replanning decision without first collecting a separate training dataset.

4. RVReplan

RVReplan is designed as a modular system for domain-independent classical planning. At its core, RVReplan is completely agnostic to the execution environment and the planner being used. This is obtained by defining interfaces and abstractions between the planner, the execution environment, and RVReplan.

The framework is composed of three layers: *abstract environment*, which abstracts the real environment and interfaces it with the runtime monitors; *runtime enforcement*, in which automatically synthesised monitors observe the runtime trace populated by propositions obtained from executed actions, check preconditions and postconditions, and stop execution when a violation requires replanning; and *replan*, where violations detected by the runtime monitor may invoke the planner at runtime.

Postcondition violations indicate that the current action did not produce an expected effect. Additional effects are treated differently: their observation does not itself constitute a monitor violation. They are recorded in the runtime trace and become actionable when their consequences cause a later monitored precondition to be violated. In that case, the diagnosis procedure examines the additional effects of previously executed actions to identify a candidate source of the discrepancy. The two types of monitor are therefore kept separate because they support different stages of this diagnosis.

Fig. 1 presents a general overview of RVReplan. The workflow followed by our approach is as follows:

1. An action is selected for execution from the plan and added to the runtime trace.
2. The trace is sent to the precondition monitor for verification.
3. The precondition monitor for that action verifies whether the trace violates the property. If it does not, then it clears the action for execution in the environment.

3a. Otherwise, if the precondition monitor detects a violation, then it triggers the precondition replanning algorithm that reasons about what to do next (continue execution, call replanning, update action specifications and call replanning, or update action specifications and continue execution);

4. After the action is executed in the environment, the results are mapped into the propositions used for planning, which are also stored in a *Snapshot* and translated to a problem specification when replanning is called in order to use the most up-to-date information about the environment;
5. The set of propositions is then added as a new event in the trace.
6. The updated runtime trace is sent to the postcondition monitor for verification.

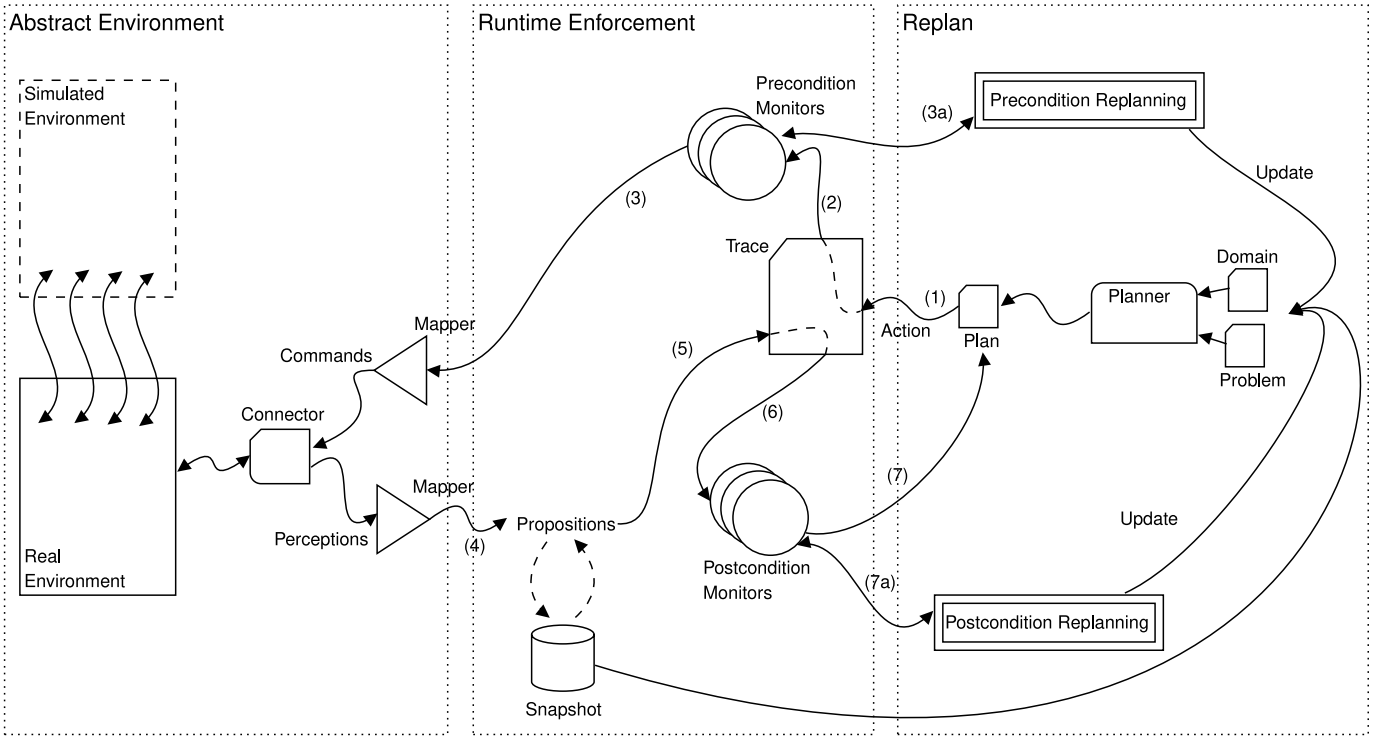


Fig. 1. RVREPLAN overview. Dashed lines represent relation, solid lines represent communication.

7. The postcondition monitor for the action that was just executed checks if the runtime trace violates its property. If it does not, it will proceed to select another action from the plan. This process repeats as long as there are actions to be executed in the plan.

7a. Otherwise, if the postcondition monitor detects a violation, it triggers the postcondition replanning algorithm.

The first event in the trace contains the initial valuation of the planning-relevant propositions, as defined in the problem specification: propositions that hold are recorded as p^+, while propositions that do not hold are recorded as p^-. Subsequent events contain the action being executed and the proposition updates caused by the effects of that action.

To accurately detect changes in actions, we must assume that any changes we observe at runtime result from the action's effects. Therefore, for the purposes of this paper, we do not consider external or probabilistic actions, dynamic events, or environmental/sensor noise. We discuss how the environment interface could be extended to accommodate sensor-derived observations and concurrent or exogenous events at the end of the Abstract Environment section.

4.1. Abstract environment

The *Abstract Environment* abstracts the *commands* that can be performed in the real environment and the *perceptions* that can be gathered. Note that unless the real environment is programmed to be an exact implementation of the domain and problem representations, actions and commands will not be the same. There must be a link between commands and actions, and perceptions and propositions, as found in the domain and problem specifications, with those that can be observed in the *Real Environment*.

The *Connector* links the real environment with the abstract one. It manages the execution interface and delegates the translation from actions to commands and from perceptions to propositions to the Mapper. To do so, it is the only component of RVREPLAN that contains environment-specific information. It can be based on a raw connection, meaning it is part of the real environment and has direct access to it. Or the *Connector* can be based on a socket connection, establishing a network link to the real environment via message passing.

The *Mapper* is responsible for translating between abstract events (*i.e.*, actions and propositions) and their counterparts in the real environment (*i.e.*, commands and perceptions).

The *Connector* manages communication with the environment, while the *Mapper* converts raw perceptions to symbolic propositions. This separation keeps the Connector platform-specific and the Mapper domain-reusable, ensuring modularity in RVREPLAN. In the current implementation, the Connector and Mapper assume that the environment is fully observable and that perceptions can be translated into definite symbolic propositions. Actions are executed sequentially, and changes observed between the start and completion of an action are therefore attributed to that action. This assumption allows the replanning reasoner to infer whether an expected effect is missing or an additional effect has occurred.

Note that if the real environment maps one-to-one to the planning representation, then the connector and mapper can be fully automated and domain-independent. This "automation" enables the connector and mapper to translate sensor data into planning propositions through declarative one-to-one correspondences between environment variables and planning predicates.

The simulated environment is a copy of the real environment, generated using updated domain and problem specifications, and is then used to test new action specifications and observe their impact on future actions (similar to lookahead planning). In this case, we modify the action specifications and apply them in a simulated environment to observe how this affects future actions in the plan and to inform the decision on what to do next. Based on the results, the reasoner may decide to continue execution as normal in the real environment, invoke replanning, invoke replanning after reverting action descriptions to their pre-simulation state, or invoke replanning with the updated action descriptions. These outcomes are described later in Section 4.3.

The *Connector* can also be used to inject wrong behaviours at particular execution steps to simulate failures in the real environment. This is useful in simulated environments and experiments, as it can trigger replanning and plan repair.

The simulated environment is not intended to reproduce the physical dynamics of the real environment or to establish that an updated action model is an exact description of reality. Rather, it acts as a symbolic sandbox over the current *Snapshot* and the revised PDDL action specifications. Candidate updates are derived from observations made by monitors that are automatically synthesised from the action schemas, rather than from domain-specific rules for diagnosing and modifying each action. The simulation then evaluates the downstream logical consequences of these updates, including whether the remaining plan can still be executed and whether the revised specifications introduce further precondition or postcondition violations. Consequently, successful simulation indicates that an updated specification is consistent with the planner's current belief state and suitable for replanning, but it does not prove that the specification is a complete or physically accurate model of the environment. Failures caused by unmodelled physical dynamics, sensor uncertainty, partial observability, or stochastic effects require complementary mechanisms.

For example, in an integrated physical system, the Connector could receive observations from multiple sensors and associate them with action start and completion events before the Mapper converts them into planning propositions. Supporting uncertain observations would additionally require a state-estimation or sensor-fusion component to determine which propositions should be added to the Snapshot. Such a component could also expose observation confidence to the Mapper, allowing uncertain proposition updates to be confirmed before they are committed to the Snapshot or used as evidence for an action-model repair. A concrete persistence-confirmation policy for integrating such information with the current reasoning procedure is discussed in Section 4.3. Similarly, supporting parallel actions or exogenous events would require traces to record the time and source of each observed change, together with a mechanism for distinguishing action effects from changes caused by other processes. These extensions would primarily affect the construction of the Snapshot and the diagnosis of violations, and would mainly require custom Connectors/Mappers. Their integration is left for future work.

4.2. Runtime enforcement

In Ferrando and Cardoso (2022), the authors introduced an approach for automated failure detection using runtime monitors to verify, at execution time, the preconditions of actions in classical planning using both Past LTL and a first-order extension of LTL. Their work was limited to failure detection and did not include any replanning mechanism. From the perspective of failure detection, their work was also limited to synthesising monitors for action preconditions, which we extend to include the automatic synthesis of RV monitors for both pre- and postconditions of actions. The monitors are then deployed to check whether the execution system satisfies these properties.

To generate parameterised monitors, we use a first-order extension of LTL, which we refer to as FO-LTL (Havelund et al., 2020). Let D denote the finite set of objects in the planning problem. Variables $x, y, \ldots$ range over D, and terms are either variables or object constants. We write $\vec{t}$ for a tuple of terms. Atomic formulae are built from action-schema names and predicate symbols. In particular, $a(\vec{t})$ is a parameterised action-occurrence atom associated with the action schema a, while $p^+(\vec{t})$ and $p^-(\vec{t})$ are parameterised proposition atoms associated with predicate symbol p. These atoms may contain variables and are therefore schema-level patterns. Under a valuation of their variables with objects from D, they denote ground action occurrences and ground propositions in the execution trace.

The FO-LTL fragment used in this paper is defined as follows:

$$\begin{aligned}\varphi ::= {} & true \mid false \mid a(\vec{t}) \mid p^+(\vec{t}) \mid p^-(\vec{t}) \\ & \mid (\varphi \wedge \varphi') \mid (\varphi \vee \varphi') \mid \neg\varphi \mid (\varphi \rightarrow \varphi') \\ & \mid (\varphi S \varphi') \mid \ominus\varphi \mid \bigcirc\varphi \\ & \mid \exists_{x\in D}.\varphi \mid \forall_{x\in D}.\varphi.\end{aligned}$$

The temporal operators have their usual meaning. Formula $\varphi\ S\ \varphi'$ states that φ' held at some previous point in the trace and that φ has held continuously since then; $\ominus\varphi$ states that φ held in the previous trace event; and $\bigcirc\varphi$ states that φ holds in the next trace event. The quantifier $\exists_{x\in D}.\varphi$ states that φ holds for at least one object assignment of x in D, while $\forall_{x\in D}.\varphi$ states that φ holds for every object assignment of x in D.

In Algorithm 1, the quantified variables are the parameters of the corresponding action schema. Given an action schema $a_i(\vec{x})$ with parameters $\vec{x} = \langle x_1, \ldots, x_n\rangle$, we use $\forall_{\vec{x}}$ as an abbreviation for $\forall_{x_1\in D} \ldots \forall_{x_n\in D}$. Therefore, the generated FO-LTL formula is a parameterised, schema-level monitoring property that is interpreted over all ground instances of a_i obtained from objects in D.

Algorithm 1: FO-LTL action monitoring.

Function *GenerateFOLTL*(A, D)
- **Input:** A set of action schemas A and the finite object domain D. Each action schema $a_i(\vec{x}) \in A$ is represented as $\langle \alpha^+_{a_i}, \alpha^-_{a_i}, \beta^+_{a_i}, \beta^-_{a_i}\rangle$. Here, $\alpha^+_{a_i}$ and $\alpha^-_{a_i}$ denote positive and negative preconditions, respectively, while $\beta^+_{a_i}$ and $\beta^-_{a_i}$ denote positive and negative effects.
- **Output:** A set *result* of FO-LTL formulae encoding the precondition and postcondition monitoring rules for all action schemas in A.

$result = \emptyset$;
foreach *action schema* $a_i(\vec{x}) \in A$ *with specification* $\langle \alpha^+_{a_i}, \alpha^-_{a_i}, \beta^+_{a_i}, \beta^-_{a_i}\rangle$ **do**
  $\vec{x} = \langle x_1, \ldots, x_n\rangle = getParameters(a_i)$;
  $\varphi_{pre} = true$;
  foreach $pre \in \alpha^+_{a_i}$ **do**
    $\varphi_{pre} = \varphi_{pre} \wedge (\neg pre^-\ S\ pre^+)$;
  foreach $pre \in \alpha^-_{a_i}$ **do**
    $\varphi_{pre} = \varphi_{pre} \wedge (\neg pre^+\ S\ pre^-)$;
  $\varphi^{pre}_{a_i} = \forall_{\vec{x}}(a_i(\vec{x}) \rightarrow \ominus\varphi_{pre})$;
  $\varphi_{post} = true$;
  foreach $post \in \beta^+_{a_i}$ **do**
    $\varphi_{post} = \varphi_{post} \wedge (\neg post^-\ S\ post^+)$;
  foreach $post \in \beta^-_{a_i}$ **do**
    $\varphi_{post} = \varphi_{post} \wedge (\neg post^+\ S\ post^-)$;
  $\varphi^{post}_{a_i} = \forall_{\vec{x}}(a_i(\vec{x}) \rightarrow \bigcirc\varphi_{post})$;
  $result = result \cup \{\varphi^{pre}_{a_i}, \varphi^{post}_{a_i}\}$;
return *result*;

In Ferrando and Cardoso (2022), two types of monitors were introduced, *parameterised* and *instantiated*. The former uses only action specifications, while the latter uses both the actions from the solution plan and action specifications to generate the monitors. For the remainder of this paper, when we mention FO-LTL monitors, we refer to parameterised monitors; when we mention LTL monitors, we refer to instantiated monitors. We present the construction of FO-LTL monitors in the

following paragraphs. Due to space constraints and the straightforward nature of LTL monitor construction, we omit its details.

Two monitors are generated for each action schema: one for its preconditions and one for its postconditions. Algorithm 1 shows the synthesis of FO-LTL runtime monitoring formulae from parameterised action schemas. It returns a set of formulae formalising the properties synthesised from each action schema's preconditions and postconditions. The algorithm for synthesising instantiated monitors is subsumed by Algorithm 1 and is therefore omitted. The input is the set of action schemas extracted from the domain specification, together with the finite object domain D. The result is initialised as an empty set and is later populated with the FO-LTL formulae used for runtime verification. The main loop iterates over the action schemas and extracts, for each schema, its parameters and its positive and negative preconditions and postconditions.

The preconditions and postconditions extracted from an action schema contain variables, and are therefore parameterised. For an action schema $a_i(\vec{x})$, the algorithm extracts the parameter tuple $\vec{x} = \langle x_1, \ldots, x_n \rangle$ and uses the abbreviation $\forall_{\vec{x}}$ to denote universal quantification over all object assignments of these parameters. The formulae φ_{pre} and φ_{post} are then constructed by conjoining the monitoring conditions generated from the positive and negative preconditions and postconditions of a_i.

These conditions use the *since* operator (S). For example, for a positive precondition pre, $(\neg pre^{-}\ S\ pre^{+})$ checks that pre was observed to hold at some previous point in the trace, and that its negative version pre^{-} has not been observed since then. Similarly, for a negative precondition, the formula $(\neg pre^{+}\ S\ pre^{-})$ checks that the proposition was previously observed to be false and has not subsequently been observed to be true. The same construction is used for positive and negative postconditions. For each action schema $a_i(\vec{x})$, the generated precondition formula universally quantifies over the action parameters and states that, whenever the action occurs, its preconditions must have held in the previous trace event: $\forall_{\vec{x}}(a_i(\vec{x}) \rightarrow \ominus\varphi_{pre})$. Similarly, the generated postcondition formula states that, whenever the action occurs, its postconditions must hold in the following trace event: $\forall_{\vec{x}}(a_i(\vec{x}) \rightarrow \bigcirc\varphi_{post})$. The two formulae, $\varphi^{pre}_{a_i}$ and $\varphi^{post}_{a_i}$, are then added to $result$.

Both parameterised and instantiated monitors only check for the absence of expected propositions. That is, both precondition and postcondition monitors report a violation only when something expected by the action specification is missing from the trace. For instance, suppose that the positive postconditions of an action schema a are expected to produce the propositions $\{p^{+}, q^{+}\}$. The corresponding postcondition monitor checks whether both p^{+} and q^{+} hold in the next trace event (*i.e.*, the event after the completion of the action). If either is missing, then the monitor reports a violation. If another proposition r^{+} is also observed in the runtime trace, it is not reported as a violation by the monitor.

However, such additional information can be useful for reasoning about potential changes in actions, since the additional proposition may indicate that the action is producing more than expected. Even though the monitors do not check for additional propositions, RVReplan still logs that information, which is then used during the reasoning about replanning.

Additional effects are therefore not reported as violations at the moment they are observed. Instead, they are recorded in the execution trace. If an additional effect later causes a monitored precondition violation, Algorithm 3 examines the observations of previously executed actions to identify the action that introduced the conflicting effect. The corresponding action specification is then updated, the candidate repair is validated through simulation, and replanning is triggered when required by the resulting execution state. Thus, additional effects can lead to model repair and replanning, but only when they become relevant to the execution of the current plan. If an unexpected effect does not affect any subsequent monitored precondition or remaining goal, the current framework records it but does not initiate corrective action.

The FO-LTL action monitoring algorithm outputs:

$$GenerateFOLTL(A, D) = \{\varphi^{pre}_{a_1}, \varphi^{post}_{a_1}, \varphi^{pre}_{a_2}, \varphi^{post}_{a_2}, \ldots, \varphi^{pre}_{a_n}, \varphi^{post}_{a_n}\}.$$

These formulae may be viewed as a conjunction of action-specific precondition and postcondition properties. In the implementation, however, we keep them as separate monitors for each action schema, allowing RVReplan to update only the monitors affected by an action-specification change. We can then automatically compute the monitor function for each FO-LTL (for parameterised monitors) or LTL (for instantiated monitors) formula using the algorithm presented in Havelund et al. (2020), which constructs the monitor as a Binary Decision Diagram (BDD). At the implementation level, all actions are split into two events: their start and end. The formulae check whether the preconditions and postconditions are satisfied at the start and end of the action, respectively.

RVReplan synthesises runtime monitors rather than directly interpreting PDDL operators, which define static schemas but lack temporal verification over execution traces. Compiling them into FO-LTL automata enables formal, trace-based validation and the creation of reusable parameterised monitors across grounded actions, ensuring efficient, domain-independent runtime checking.

The *Snapshot* tracks changes in the environment. A snapshot is a view of the current state of the environment, in which all planning-related propositions are stored along with their truth values. The *Snapshot* is initialised with the propositions defined in the problem file. This represents the planner's belief about the real environment's initial state (and thus its initial *Snapshot*). There is an assumption here that the initial state from the problem file matches the initial state of the real (or simulated) environment. The *Snapshot* is updated when new propositions have been generated as a result of the latest action executed. At the implementation level, it is a data structure used to track all propositions that contain information about the latest state we have observed in the environment.

4.3. Replan

When a violation is detected, two algorithms reason about it and determine whether replanning is necessary. The algorithm used depends on which monitor reported the violation, either a precondition or a postcondition monitor. In these algorithms, $FailAct$ denotes the action whose monitor reports a violation. The sets $PrevActs$ and $FutActs$ contain the previously executed and future ground action occurrences in the current plan, respectively.

A violation of the postconditions of an action triggers Algorithm 2. The postcondition monitor of the violated action identifies the failed action, the missing propositions that triggered the violation, the remaining future actions in the plan, and the set of available action schemas. The lack of expected postconditions means that the action did not produce some of the effects specified in the domain. The reasoning first checks whether any missing propositions are required by the preconditions of a future action. If this is not the case, the missing effects are inconsequential for the rest of the current plan, and execution can continue without interference. Otherwise, the action specification of the failed action is updated to reflect that it no longer produces the missing effects. The corresponding postcondition monitor is updated, and the change is propagated to the action specification used by the planner during replanning. The algorithm then calls $SimulateAndReplan()$ to validate the update before replanning.

Algorithm 3 reasons about violations in the preconditions of an action. It first checks whether the missing propositions are a direct consequence of the expected effects of previously executed actions in the plan.

Algorithm 2: Postcondition replanning algorithm.

Function *PostconditionReplanning(*$FailAct, P, FutActs, A$*)*
- **Input:** The failed action $FailAct$ whose postcondition monitor reports a violation; the set of missing propositions P causing the violation; the set of future actions $FutActs$ that remain to be executed; and the set of available action schemas A.
- **Output:** An updated monitoring and replanning process that repairs the violation when any missing proposition in P is relevant to the preconditions of a future action.

if $\exists_{a \in FutActs}.\exists_{p \in P}.p \in \alpha_a^- \cup \alpha_a^+$ **then**
  $UpdateMonitor(FailAct, (\beta^+_{FailAct} \cup \beta^-_{FailAct}) \setminus P)$;
  return $SimulateAndReplan(FailAct, P, \{FailAct\}, A)$;

Algorithm 3: Precondition replanning algorithm.

Function *PreconditionReplanning(*$FailAct, P, PrevActs, A$*)*
- **Input:** The failed action $FailAct$ whose precondition monitor reports a violation; the set of missing propositions P causing the violation; the set of previously executed actions $PrevActs$; and the set of available action schemas A.
- **Output:** A replanning decision. If the violation is caused by the effects of a previously executed action, the planner replans without $FailAct$. Otherwise, the relevant previous action monitors are updated, and simulation-based replanning is invoked.

if $\exists_{a \in PrevActs}.\exists_{p^+ \in P}.p^- \in \beta_a^-$ ***or*** $\exists_{a \in PrevActs}.\exists_{p^- \in P}.p^+ \in \beta_a^+$ **then**
  return $Replan(A \setminus \{FailAct\})$;
else
  $UpdatedActs = \{\}$;
  for $a \in PrevActs$ **do**
    $Post^{+-} = post(a) \setminus (\beta_a^- \cup \beta_a^+)$;
    if $Post^{+-} \cap \overline{P} \neq \emptyset$ **then**
      $UpdatedActs = UpdatedActs \cup \{a\}$;
      $UpdateMonitor(a, \beta_a^- \cup \beta_a^+ \cup (Post^{+-} \cap \overline{P}))$;
  return $SimulateAndReplan(FailAct, P, UpdatedActs, A)$;

If this check succeeds, then the current plan was invalid with respect to the updated execution context, and the planner attempts to replan without using the failed action schema. That is, the failed action schema remains available to the planner, but it is ignored for that instance of the replanning call to avoid being stuck in an infinite loop.

Example: simulation of an action-model update. Consider the plan fragment

```
load-truck(p,t,l1);
drive-truck(t,l1,l2,c);
unload-truck(p,t,l2).
```

The action `load-truck(p,t,l1)` is expected to add `in(p,t)` and delete `at(p,l1)`. Suppose that its real execution deletes `at(p,l1)` but fails to add `in(p,t)`. The postcondition monitor therefore reports a violation, and Algorithm 4 is called with

`FailAct` = `load-truck(p,t,l1)` and $P = \{$`in(p,t)`$\}$.

The diagnosis procedure removes the missing effect from the corresponding `load-truck` action schema and adds this action to `UpdatedActs`.

The simulation begins with a copy of the current Snapshot and first attempts to execute `FailAct` using the updated action specification. Because the real execution has already deleted `at(p,l1)`, the precondition check initially fails. The missing precondition is added only to the simulated Snapshot, allowing the candidate update to be evaluated without modifying the real environment. The simulator then emits the action-start event, applies the revised effects of `load-truck`, and emits the action-end event. Since the updated specification no longer requires `in(p,t)` as an effect, the updated postcondition monitor reports no violation.

The simulation then proceeds to the remaining plan. The preconditions of `drive-truck(t,l1,l2,c)` hold, so its effects are applied to the simulated Snapshot. When the simulation reaches `unload-truck(p,t,l2)`, its precondition monitor detects that `in(p,t)` is missing and returns `unload-truck` as the violating action. Because `unload-truck` is not in `UpdatedActs`, Algorithm 4 retains the candidate update and invokes replanning using the revised domain model.

If the re-executed `load-truck` action had instead violated its updated monitor, its previous specification and monitor would have been restored, and the action would have been excluded from that replanning instance. If the entire simulated execution had completed without a violation, the update would have been retained and execution in the real environment would have resumed from the point at which it was paused.

If the check does not succeed, then, under our assumption that runtime changes are caused only by known action effects, some previously executed action must have introduced additional effects not present in its specification. For every previously executed action, the algorithm computes the propositions observed after execution that were not part of the action's expected effects. If these additional propositions intersect with the opposite-polarity set $\overline{P}$ of the failed action's missing pre-

conditions, then the corresponding postcondition monitor is updated to include the conflicting observations. Here, $post(a)$ denotes the signed proposition updates observed after executing a, while $\beta_a^- \cup \beta_a^+$ denotes the signed effects expected from its action specification. For a signed proposition p^+ or p^-, let $\overline{p}$ denote the same proposition with the opposite polarity, and for a set P let $\overline{P} = \{\overline{p} \mid p \in P\}$. An additional effect can explain a missing precondition only when it establishes the opposite valuation of that precondition. Consequently, the diagnosis compares additional observations with $\overline{P}$ rather than merely with the proposition name.

Algorithm 4 defines the simulation-and-replanning procedure. This creates a copy of the current state of the environment (called the simulated environment) that starts from the same point in time as when the violation was detected in the real environment. The simulated environment attempts to run the same action that failed, but now uses the updated action specifications. If the action fails due to a precondition violation, we add the missing propositions to the simulated environment to test the new actions. The simulation stops when a violation is detected and returns the violating action. Following the notational convention introduced in Section 2, any subsequent update is applied to the corresponding action schema. If no violations are detected, the postcondition monitors retain their updated properties; however, the plan's execution in the real environment resumes from the point where it stopped before the simulation. Otherwise, if the violation detected during the simulation corresponds to one of the updated actions, then the monitor and specification of that violating action are restored to their previous versions, and replanning is attempted without using that action (similarly to Algorithm 3, the failed action schema is retained, but ignored in this replanning instance). If the detected violation does not correspond to one of the updated actions, we retain the action-specification changes and call replanning with the updated domain specification.

The simulated environment follows the same monitoring cycle used during normal execution. Starting from a copy of the current Snapshot, the failed action is first re-executed using its updated specification, after which the remaining actions are executed through the same connector interface. Their preconditions and postconditions are evaluated by the corresponding generated monitors, including any candidate monitor updates introduced by the diagnosis procedure. The $Simulate()$ function proceeds through the remaining actions until a monitor reports a violation, in which case it returns the corresponding action, or until the simulated execution completes without a violation. Reusing the same connector and monitoring infrastructure avoids introducing a separate validation semantics for candidate repairs and allows their consequences to be evaluated against the same trace-based observations used during normal execution.

The following running example illustrates in more detail how a candidate action-model update is propagated through the simulated execution and how the resulting monitor verdict determines the subsequent replanning decision.

The simulated environment performs a symbolic look-ahead over the updated model: if no violation is found, execution can continue; if a downstream action violates a monitor, replanning is triggered; and if an updated action itself violates its specification during simulation, the candidate update is reverted, the previous specification and monitor are restored, and replanning proceeds without removing the action schema.

A monitor violation does not cause the corresponding action schema to be removed from the planning domain. When a candidate action-model update is required, the revised specification is first evaluated in the simulated environment. If simulation completes without a violation, the update is retained and execution resumes. If another action violates a monitor, the update is retained and replanning is performed using the revised domain model. If instead one of the actions whose specification has just been modified violates during simulation, the candidate update is rejected, the previous specification and monitor are restored, and replanning is performed using the preceding domain model. The action schema itself remains available to the planner. An action schema is ignored from the current replanning instance only in the specific cases identified by Algorithms 3 and 4: when the failed action is incompatible with effects already expected from previous actions, or when an updated action continues to violate its specification during simulation.

Algorithm 4: Simulate and replan algorithm.

```
Function SimulateAndReplan(FailAct, P, UpdatedActs, A)
    Input: The failed action FailAct whose monitor reports a violation; the set of missing propositions P causing the violation; the set of actions UpdatedActs whose monitors or specifications have been modified; and the set of available action schemas A.
    Output: A decision based on the outcome of the simulation. If no violation is detected, execution continues with the updated action specifications. If the simulated violation corresponds to one of the updated actions, that action's monitor is restored, and replanning is performed without that action. Otherwise, if another action violates a monitor, replanning is performed using the full set of available action schemas A.
2   violationAct = Simulate(FailAct, P, UpdatedActs);
3   if violationAct = none then
4       return continue;
5   if violationAct ∈ UpdatedActs then
6       UpdateMonitor(violationAct, β⁻_violationAct ∪ β⁺_violationAct);
7       return Replan(A \ {violationAct});
8   else if violationAct ∉ UpdatedActs then
9       return Replan(A);
```

Distinguishing between missing and additional propositions clarifies whether an action failed in its preconditions or effects. A *missing* proposition marks an expected condition that does not hold, while an *additional* proposition records an unexpected observation that may later contribute to a monitored precondition violation. This distinction guides RVREPLAN in deciding whether to revise an action's effects or adjust preceding actions, motivating the separation between Algorithms 2 and 3.

It is important to distinguish repairing an inaccurate action specification from guaranteeing that the planning goal remains reachable. RVREPLAN can update the effects of an existing action to reflect its observed runtime behaviour, but it cannot create capabilities that are absent from the domain. After such an update, successful replanning therefore requires that the revised planning problem still admits a plan from the current Snapshot. If the corrected action specification reveals that a capability required by every solution is no longer available, the goal may become unreachable. In that case, the planner correctly reports that no solution exists.

4.4. Discussion of the diagnosis

The set-based tests in Algorithms 2 and 3 implement a restricted causal diagnosis over the monitored symbolic trace. Let

$$E(a) = \beta_a^+ \cup \beta_a^-$$

denote the signed effects expected for an executed action and let $O(a) = post(a)$ denote the signed proposition updates observed after its execu-

tion. The corresponding missing and additional effect sets are

$$M(a) = E(a) \setminus O(a), \qquad X(a) = O(a) \setminus E(a).$$

For a postcondition violation, the propositions reported by the monitor form a subset of $M(FailAct)$. Under sequential deterministic execution, full observability, and the absence of exogenous changes, the action that has just completed is therefore the only action occurrence that can explain this immediate discrepancy. RVREPLAN proposes removing the missing effects from its specification only when they are relevant to the remaining execution, and the proposed repair is subsequently checked through simulation.

A precondition violation requires a different diagnosis because its cause may lie earlier in the trace. Let P be the set of unsatisfied signed preconditions. If an expected effect of a previously executed action intersects $\overline{P}$, then the current domain model already explains the conflicting valuation and there is no evidence for modifying that previous action; RVREPLAN therefore replans using the current model. Otherwise, the candidate sources of the discrepancy are

$$C(P) = \{a \in PrevActs \mid X(a) \cap \overline{P} \neq \emptyset\}.$$

Each member of $C(P)$ is an earlier action for which the trace contains an unexpected effect with precisely the polarity needed to explain the later precondition failure. These observed effects form candidate model repairs, which are then subjected to the symbolic simulation before being retained. Thus, the set checks do not attempt to infer an arbitrary latent root cause; they construct explanations that are justified by the observed trace under the framework's execution assumptions. If several previous actions satisfy this criterion, RVREPLAN does not claim to identify a unique physical cause: the supported candidates are propagated to the validation stage. Richer diagnosis under noise, concurrency, partial observability, or competing latent fault hypotheses would require additional causal or probabilistic models and remains outside the current scope.

The current approach assumes that detected discrepancies represent persistent changes in action behaviour rather than transient execution errors. It therefore does not currently distinguish a model-level change from a temporary failure caused by environmental or sensor noise. A concrete extension would introduce a persistence-confirmation layer between a monitor verdict and the action-model update performed by Algorithms 2 and 3. The first observation of a discrepancy would be treated as tentative and would not immediately modify the action schema. The Connector and Mapper could first request an additional observation of the propositions involved in the violation and, for actions explicitly declared safe to retry, repeat the action for a bounded number of attempts. Only a discrepancy confirmed by repeated observations or executions would be classified as persistent and passed to the existing RVREPLAN repair procedure. If the discrepancy disappears, it would instead be classified as transient and the domain model would remain unchanged.

For noisy sensor inputs, the same confirmation layer could operate on confidence information supplied by the Mapper or by an external state-estimation or sensor-fusion component. Low-confidence observations would therefore be confirmed before becoming symbolic updates to the Snapshot. This extension would preserve the current repair and replanning algorithms: once a discrepancy has been classified as persistent, the existing RVREPLAN workflow would proceed unchanged. When an action cannot safely be retried, confirmation could rely on repeated sensing rather than repeated execution. Designing and evaluating such a persistence classifier is outside the present deterministic setting, but this provides a concrete integration path without requiring changes to the planning formalism itself.

Table 1
Monitor synthesis time (in seconds) per domain. Case (i) is the experiment with a postcondition violation, case (ii) is the experiment with a precondition violation, and case (iii) is the experiment with both a precondition and a postcondition violation. (P) denotes the synthesis of parameterised monitors, and (I) denotes the synthesis of instantiated monitors.

Domain	Initial synthesis (P)	Initial synthesis (I)	Runtime synthesis (P)	Runtime synthesis (I)
logistics	300.1	7.7	i) 3.82 ii) 16.68 iii) 20.71	i) 17.11 ii) 17.32 iii) 30.18
depot	914.6	6.9	i) 9.97 ii) 96.87 iii) 107.65	i) 15.90 ii) 12.89 iii) 28.70
satellite	85.9	5.3	i) 4.05 ii) 16.68 iii) 21.74	i) 13.33 ii) 13.82 iii) 13.97

5. Experiments

RVREPLAN is implemented[2] in Python and integrates two external tools: Fast Downward (FD) (Helmert, 2006) for planning and DejaVu (Havelund et al., 2018) for runtime-monitor synthesis and execution.

The evaluation is organised into two complementary studies. The first is a controlled characterisation on three IPC domains, designed to analyse monitor-synthesis cost, replanning behaviour, and plan stability in detail. The second is an expanded implementation-level evaluation introduced to assess the framework over a broader benchmark set, provide explicit comparison points, and evaluate recovery without introducing fallback actions. The two studies therefore serve different purposes: the former provides detailed engineering characterisation, while the latter provides the evidence for model-repair correctness, recoverability, and a baseline comparison.

5.1. Controlled characterisation study

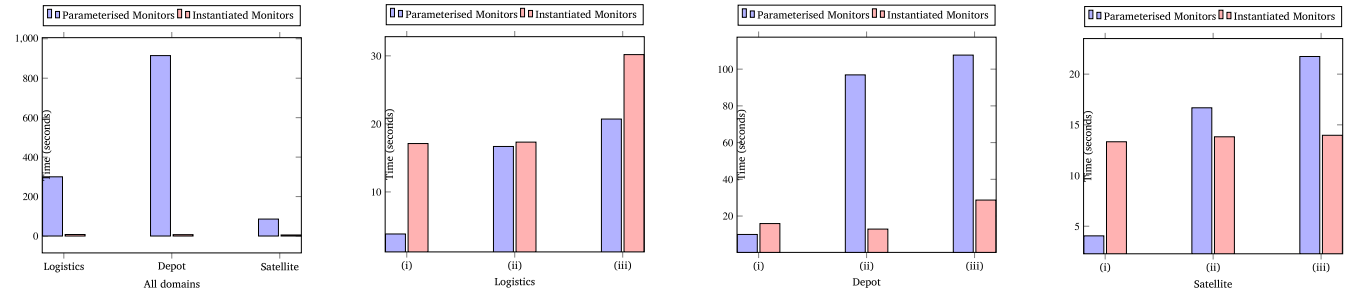
The first study uses the three IPC domains *Logistics*, *Depot*, and *Satellite*, one representative problem per domain, and three controlled failure scenarios: (i) a missing expected effect producing a postcondition violation, (ii) an additional effect that later produces a precondition violation, and (iii) a combined execution containing both types of fault. The study was run with Python 3.10.12, Fast Downward 22.12+, and DejaVu 1.0 on an 11th Gen Intel Core i9-11900KF at 3.50 GHz with 64 GB RAM and Ubuntu 22.04.2 LTS. Each configuration was repeated 20 times and the reported values are averages.

This study was designed to exercise the complete monitor–repair–simulation–replanning cycle. For that reason, small fallback action were deliberately added to preserve solvability after the injected persistent faults. For example, in *Logistics*, a failure of `load_truck` was paired with human-assisted loading actions, while an additional effect that unbalanced an aircraft was paired with a `balance_airplane` action. These actions are not generated by RVREPLAN and are not a requirement of the architecture; they are part of the controlled experimental scenario. The expanded study in Section 5.2 removes this assumption entirely and allows corrupted benchmark instances to become genuinely unsolvable.

5.1.1. Monitor synthesis

Table 1 summarises monitor synthesis times for both parameterised and instantiated variants. The latter achieves significantly faster initial synthesis, as also shown in Fig. 2a, consistent with Ferrando and Cardoso (2022). Instantiated monitors remain more efficient as long as the number of plan actions stays below a domain-dependent threshold.

[2] RVREPLAN implementation and all experimental artefacts are available at https://github.com/AngeloFerrando/RVREPLan



(a) Initial synthesis time for all the domains.

(b) Runtime synthesis time for the Logistics domain.

(c) Runtime synthesis time for the Depot domain.

(d) Runtime synthesis time for the Satellite domain.

Fig. 2. Monitor synthesis results.

At runtime, when a monitor reports a violation, one or more monitors may have to be updated. From Table 1, we can conclude that the initial synthesis requires much more time in the case of parameterised monitors, while it requires less time in the case of instantiated ones. In the case of parameterised monitors, this is because, at the beginning, all monitors must be synthesised (two monitors per action schema, to be precise, one for its preconditions and one for its postconditions). In this case, if we are using parameterised monitors, then we only have to synthesise and replace the monitors for the actions that have changed. When using instantiated monitors, we need to resynthesise all the monitors after replanning, as this will have generated a new plan. This variation in performance depends on how many actions (and how many parameters) have changed. Nonetheless, at runtime, the synthesis time in the instantiated case is generally lower than that of its counterpart. These results are also shown as plots for each domain in Fig. 2(b–d). The more actions and their parameters that change, the longer the runtime synthesis time for the parameterised monitors will be. Similarly, a higher plan size generated during replanning will increase the runtime synthesis time of instantiated monitors. Since runtime monitor synthesis affects execution time, we avoid regenerating all monitors whenever a single modification is required. Therefore, we generate two monitors for each action schema: one for its preconditions and one for its postconditions. This localises monitor updates to the action schemas whose specifications change, rather than requiring global monitor regeneration.

The synthesis time reported in Table 1 is determined mainly by DejaVu (Havelund et al., 2018). RVREPLAN's generation of both FO-LTL and LTL formulae takes less than 0.5 seconds; DejaVu takes the rest of the time to compile the formulae into monitors. These measurements show that monitor compilation, rather than formula generation or replanning logic within RVREPLAN, is the main bottleneck in the current implementation. This dependency has different implications for initial and runtime synthesis. Initial monitors can be generated before deployment when the original domain and plan are known, allowing much of this cost to be incurred offline. Runtime synthesis, however, occurs after an action specification or plan changes and therefore directly delays the resumption of execution. This is particularly relevant for parameterised monitors when several action schemas are updated, and for instantiated monitors when replanning produces a new plan whose monitors must all be regenerated.

RVREPLAN is not conceptually tied to DejaVu: the generated temporal formulae form an interface between the framework and the monitor-synthesis backend. DejaVu was selected because it supports the FO-LTL formulae required by the parameterised monitors, but a different backend could be used if it provides the required logic and monitor interface. Tools with faster propositional-LTL compilation could reduce the latency of instantiated monitors, although they would not necessarily support the quantified formulae used by the parameterised variant. Further work could evaluate alternative synthesis backends and investigate techniques such as caching previously generated monitors and synthesising independent action monitors in parallel. The performance reported here should therefore be interpreted as the performance of the current DejaVu-based implementation rather than an inherent lower bound of the RVREPLAN approach.

Other RV libraries with faster monitor compilation times could provide faster synthesis, such as LamaConv,[3] Spot (Duret-Lutz et al., 2022) or Lola (D'Angelo et al., 2005). LamaConv does not support FO-LTL, therefore we would not be able to use the parameterised monitors, but their monitor compilation speed could justify using only instantiated monitors.

5.1.2. *Replanning and plan behaviour*

The remaining experiments focus on the replanning mechanism and are therefore agnostic to whether monitors are parameterised or instantiated. Fig. 3(a–c) present the results for replanning time, total plan size, and replanning plan size, respectively. Table 2 reports these results.

For each experiment in Table 2, we report the planning time required to generate the initial plan, the total time spent on replanning, the size of the executed plan (how many actions were executed successfully), and finally, the average size of the plans generated through replanning. We do not show the times for the simulated environment, as these were trivial (less than a second) since we do not have temporal information about the actions.

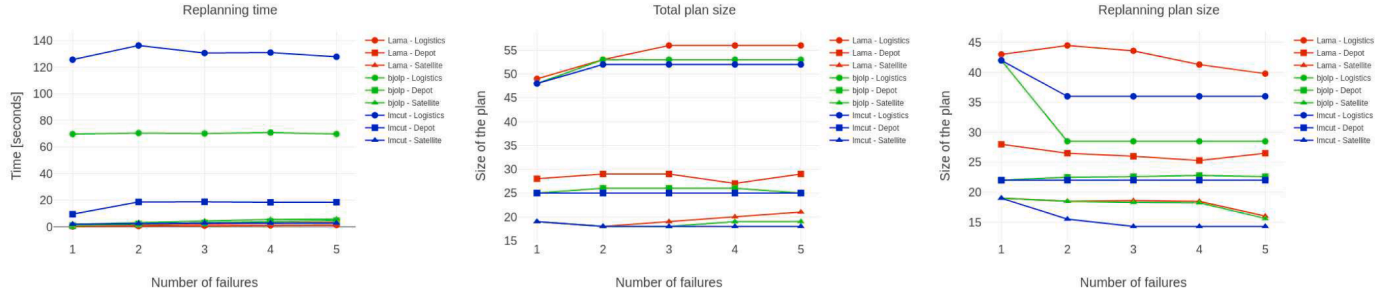
As expected, the replanning time is affected by the number of violations detected at runtime that ultimately require replanning. Case (iii) takes longer than cases (i) and (ii) because it has more violations. In many cases, case (i) has less replanning time than case (ii), which is expected because, in case (i), the action is simplified by removing some of its effects, while, in case (ii), effects are added to an action.

Cases i) and ii) exhibit similar replanning times for most combinations of domains and planner configurations. Additional effects may improve heuristic guidance when they contribute towards goals or useful intermediate states, but they may also increase search effort when they introduce conflicts or interact unfavourably with state-space constraints. The planner configuration can have a substantial effect on recovery latency. In particular, the optimal configurations may spend considerably longer searching after a runtime model update. This should not be interpreted as additional monitoring cost introduced by RVREPLAN: once replanning is requested, RVREPLAN delegates the revised planning problem to Fast Downward, and the reported replanning time is dominated by that planner's search. The difficulty of the revised task may differ significantly from that of the original problem because an effect has been added or removed and the planner is invoked from the current execution state. For an optimal configuration, finding a replacement plan is not sufficient; the search must also establish its optimality, which can make recovery unsuitable for time-critical execution even on otherwise compact domains.

[3] https://www.isp.uni-luebeck.de/lamaconv

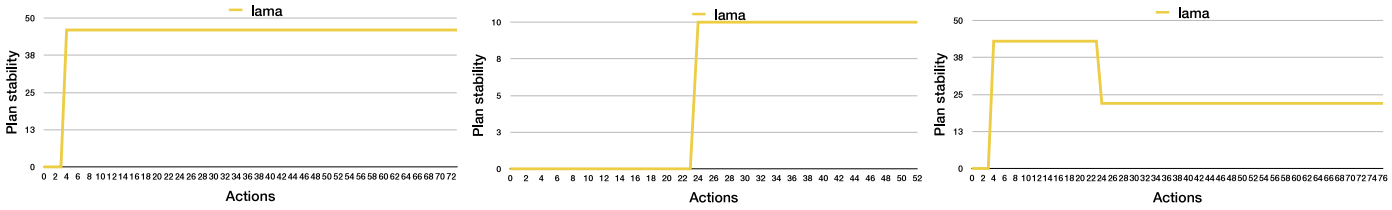
Table 2
Replanning results. Times are in seconds. – means the planner did not conclude within 10 min. **Bold** identifies the best results.

Planner	Domain	Initial plan time	Total replan time	Total plan size	Replan plan size
lama	logistics	**0.28**	i) **0.30** ii) **0.30** iii) **0.57**	i) **73** ii) **52** iii) **76**	i) **67** ii) **31** iii) **58**
	depot	**0.43**	i) **0.38** ii) **0.44** iii) **0.75**	i) 35 ii) 28 iii) 54	i) 29 ii) 18 iii) 26
	satellite	**0.27**	i) **0.26** ii) **0.28** iii) **0.52**	i) 22 ii) 22 iii) 31	i) **19** ii) 15 iii) 21
bjolp	logistics	15.38	i) – ii) – iii) –	i) – ii) – iii) –	i) – ii) – iii) –
	depot	0.91	i) 0.77 ii) 0.46 iii) 1.16	i) **26** ii) **25** iii) **30**	i) 25 ii) **12** iii) **18**
	satellite	0.30	i) 0.76 ii) 0.88 iii) 28.95	i) **21** ii) **20** iii) **28**	i) **19** ii) **14** iii) **18**
lmcut	logistics	785.51	i) – ii) – iii) –	i) – ii) – iii) –	i) – ii) – iii) –
	depot	12.76	i) 2.53 ii) 1.77 iii) 10.35	i) **26** ii) **25** iii) 32	i) **21** ii) **12** iii) 19
	satellite	1.77	i) 1.01 ii) 31.80 iii) 104.78	i) **21** ii) **20** iii) **28**	i) **19** ii) 16 iii) **18**



(a) Replanning time. (b) Total plan size. (c) Replanning plan size.

Fig. 3. Replanning time, total plan size, and replanning plan size.



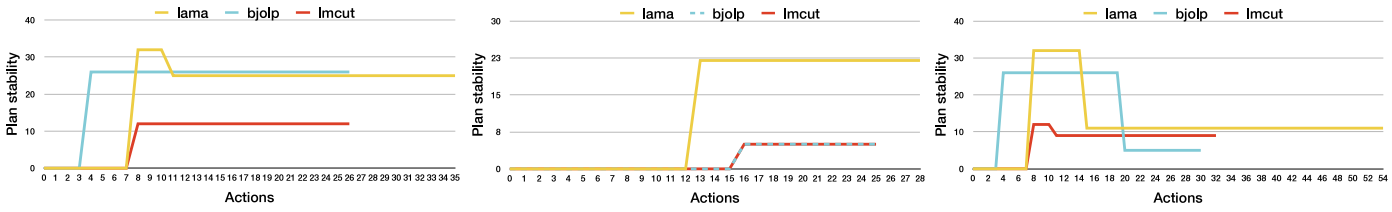
(a) Logistics (case i). (b) Logistics (case ii). (c) Logistics (case iii).

Fig. 4. Plan stability in the Logistics domain.

RVREPLAN is planner-agnostic and does not require an optimal planner. A practical deployment can therefore select the planning policy according to its recovery requirements. When bounded response time is more important than plan optimality, a satisficing configuration such as `lama-first` can be used as the primary replanner. An implementation may also impose a per-replanning time budget and fall back from an optimal configuration to a satisficing one if the budget is exceeded. Optimal configurations remain appropriate when solution quality is more important than recovery latency.

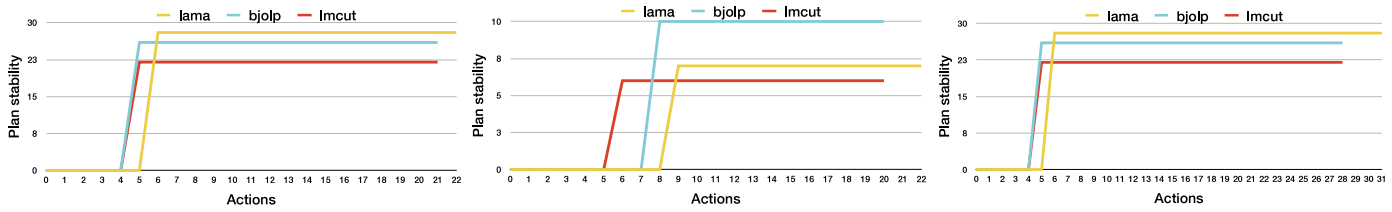
5.1.3. *Plan stability*

Plan stability was first introduced in Fox et al. (2006) as a metric for replanning and plan repair, quantifying deviations between the original



(a) Depot (case i). (b) Depot (case ii). (c) Depot (case iii).

Fig. 5. Plan stability in the Depot domain.



(a) Satellite (case i). (b) Satellite (case ii). (c) Satellite (case iii).

Fig. 6. Plan stability in the Satellite domain.

and new plans. Since then, it has been used extensively in the planning literature, for example, more recently in Babli et al. (2023).

The original definition of plan stability in Fox et al. (2006) is based on the difference between two plans.

Definition 3. The difference between an original plan π_0 and a new plan π_1, $D(\pi_0, \pi_1)$, is the number of actions from π_1 not in π_0 plus the number of actions from π_0 not in π_1.

Note that the symmetry relation is significant because plans can have different lengths; therefore, comparing only one way does not provide an accurate representation of plan stability.

The focus of RVREPLAN in this paper is to detect changes in action specifications at runtime, update them accordingly, and use the new ones to guide replanning. Our approach was not developed with plan stability in mind. In fact, we can expect plans to change considerably when replanning occurs because the actions themselves are changing, and these actions are the main elements used to measure plan stability. Nonetheless, this can still be a valuable metric for understanding, at a glance, how different the plans are.

One limitation of the original definition of plan stability is that it does not explicitly account for execution runs in which replanning occurs multiple times. It was only used to compare the stability of two plans. It also compared two static plans in isolation and did not consider replanning within the plan, which is necessary if replanning occurs online (at execution time). For example, when executing a plan π_0 with two successful actions, we detect that an action has changed, necessitating replanning and yielding a new plan π_1. Then $D(\pi_0, \pi_1)$ should not consider the first two actions of π_0.

In this paper, we extend the concept of plan stability to encompass an entire execution, including any number of replanning events necessary for that run. Let r denote the number of replanning events in a particular execution. Thus, $r = 0$ means that the original plan is executed without replanning, $r = 1$ means that one new plan is generated, and so on. If $r = 0$, plan stability is 0, meaning that the plan is perfectly stable (*i.e.*, the lower the stability value, the more stable it is). Otherwise, the plan stability of an execution trace is:

$$\sum_{i=0}^{r-1} D(\pi_i[a_i + 1 :], \pi_{i+1})$$

That is, the sum of all the differences between pairs of adjacent plans from an execution trace. Note that $[a_i + 1 :]$ is a traditional slicing operator, where a_i is the number of actions successfully executed before the ith replanning event, *i.e.,* actions that appear before will not be considered because they have already been executed and do not need to be replanned.

In Figs. 4–6, we show the results for plan stability split by our three different domains and by the three other cases of failure injection. On the x-axis, we have the total number of actions executed, and on the y-axis, we have the plan stability value at that point in execution. For example, in *Depot, case (i)*, both *bjolp* and *lmcut* only had to replan once. The former replanned at action 3, resulting in a plan stability of 26, and the latter replanned at action 7, with a plan stability of 12. *lama* had to replan twice, first at action 7 with a plan stability of 32 and then at action 11 with a plan stability of 25, resulting in a total plan stability of 57. Note that in the logistics domain, only *lama* was able to finish planning/replanning under the time limit.

Intuitively, one would expect that when replanning occurs close to the end of the plan, the stability values should be lower than those from earlier. This can be observed in the Depot, case (i), plot in Fig. 5 for *lama*, which had to replan twice and whose second stability value was lower than the first, and in Depot, case (iii), where we see more stable plans (lower stability values) for all three planners. This observation holds only when comparing the stability of replans within a single planner configuration. In general, *lama* has the worst plan stability (higher values) when compared to others, which makes sense because *lama* does not provide optimal solutions. Therefore, whenever it has to replan, the new plan may differ significantly from the previous one. Conversely, *bjolp* and *lmcut* provide optimal solutions, but in our experiments, *lmcut* performs better. The specific reason is outside the scope of this paper, but we postulate that it is likely related to their search heuristics.

5.2. Expanded evaluation and baseline comparison

The second study empirically validates whether the conclusions of the controlled characterisation persist under broader benchmark coverage and without engineered fallback actions. It also introduces explicit comparison points so that the contribution of action-model repair can be separated from state-only replanning and from perfect model information.

The experiments were run on an Apple M3 Pro-system with 12 CPU cores and 18 GB RAM, using Python 3.14.6, Fast Downward 22.12+, and DejaVu 1.0. The planner timeout was set to 30 s and the timeout for a complete RVREPLAN execution was set to 600 s. The configurations are deterministic and each unique configuration was executed once; timing summaries below therefore aggregate across different fault configurations rather than repeated runs of the same configuration. Final success

Table 3
Benchmark breadth and fault distribution. *Rec.* and *Unrec.* denote faults for which the Oracle reference respectively finds a replacement plan or proves that no plan exists under the persistent modified model.

Domain	Problems	Objects	Initial facts	Goals	Missing	Additional	Rec.	Unrec.
Logistics	5	15–22	30–44	4–8	5	5	0	10
Depot	5	13–21	36–68	2–10	5	5	0	10
Satellite	5	12–25	17–69	3–8	5	5	5	5
Gripper	5	8–16	15–31	4–12	5	5	0	10
Miconic	5	3–15	7–71	1–5	5	5	0	10
Overall	**25**	**3–25**	**7–71**	**1–12**	**25**	**25**	**5**	**45**

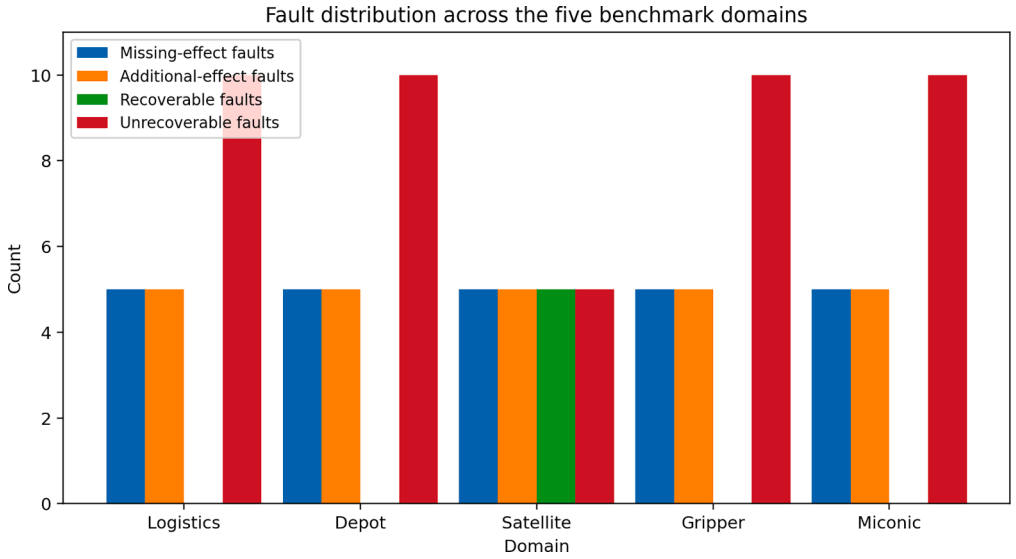


Fig. 7. Fault distribution across the five benchmark domains. Each domain contributes five missing-effect faults and five additional-effect faults. The only recoverable faults in the frozen matrix are the five additional-effect cases in *Satellite*; the remaining 45 faults are Oracle-certified as genuinely unrecoverable under the persistent modified model.

is checked independently against the complete original PDDL goal. We distinguish a valid plan, `NO_PLAN`, planner timeout, planner error, and framework error.

5.2.1. Expanded setup

We evaluate five classical planning domains—*Logistics*, *Depot*, *Satellite*, *Gripper*, and *Miconic*—using five problem instances per domain. The PDDL files are public IPC benchmark encodings copied without semantic changes from the Fast Downward benchmark repository.[4] Across the 25 problems, the number of objects ranges from 3 to 25, the initial state contains between 7 and 71 facts, and the goal contains between 1 and 12 propositions. Table 3 summarises the benchmark.

For each problem we inject two persistent one-effect discrepancies: one *missing effect*, in which an effect specified by the PDDL action no longer occurs, and one *additional effect*, in which execution produces a state change absent from the nominal action specification. This gives 50 fault configurations. All 50 faults change the observable state when injected, and in all 50 cases the discrepancy remains relevant to a later precondition or to a remaining goal. Unlike the previous experiments, no fallback action is introduced to force the corrupted problem to remain solvable.

The main comparison uses `lama-first` and three execution configurations. The first is the executable RVReplan implementation. The second, *State-only*, is a baseline control that replans from the observed current state after a discrepancy while retaining the nominal action model; it isolates the contribution of repairing the action specification. *Oracle* is a perfect-information reference, not a competing method: before replanning, it is given the exact persistent model modification that was injected. Oracle therefore establishes whether a replacement plan exists when the changed action behaviour is known exactly.

The final design contains 25 nominal RVReplan executions and 150 paired recovery executions (50 faults × three configurations). A separate planner sensitivity analysis selects one missing-effect and one additional-effect fault from a representative problem in each domain and evaluates RVReplan with `lama-first`, `seq-opt-bjolp`, and `seq-opt-lmcut`. The ten `lama-first` observations are reused from the main experiment, and the two optimal configurations add 20 executions. Overall, the study contains 195 unique executions and 205 analytical rows.

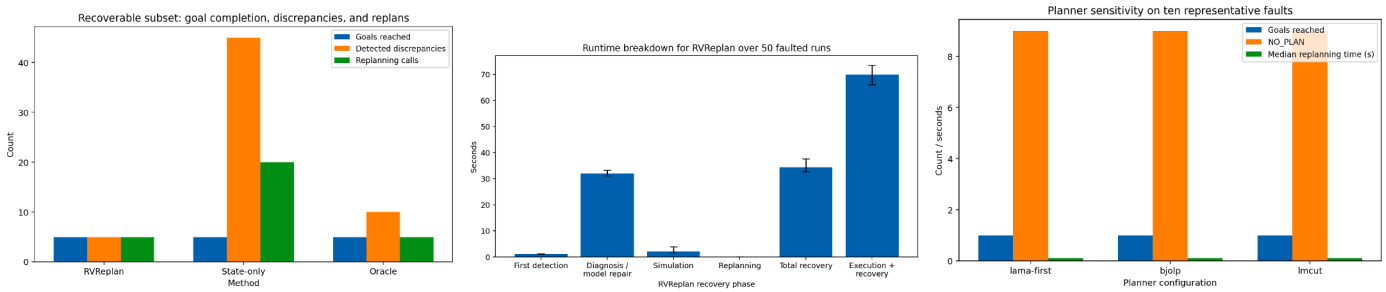
5.2.2. Recovery effectiveness and model repair

All 25 nominal RVReplan executions reach the complete original PDDL goal with zero discrepancies and zero replanning calls. In the faulted runs, all 50 injected discrepancies are encountered and detected by RVReplan. The Oracle reference classifies five faults as recoverable and 45 as genuinely unrecoverable under the persistent modified model. The five recoverable cases are additional-effect faults in the Satellite domain.

Fig. 7 shows this distribution across the five domains.

As shown in Table 4, all three configurations reach the goal in all five recoverable cases. While RVReplan achieves the same goal-completion rate as the state-only replanning baseline, the difference is instead visible in repeated interaction with the persistent model discrepancy. RVReplan records exactly one discrepancy and one replan per recoverable fault, for five discrepancies and five replans in total. State-only reaches

[4] https://github.com/aibasel/downward-benchmarks



(a) Comparison on recoverable cases. (b) Median runtime breakdown (IQR). (c) Planner-sensitivity results.

Fig. 8. Recovery performance, runtime breakdown, and planner sensitivity.

Table 4
Recovery outcomes. Replanning calls are totals over the five recoverable faults. Oracle is a perfect-information reference.

Method	Rec. goals	Unrec. NO_PLAN	Replans
RVREPLAN	5/5	45/45	5
State-only	5/5	45/45	20
Oracle reference	5/5	45/45	5

the same goals while retaining the incorrect action model, and consequently records 45 discrepancies and 20 replans. Our action-model repair therefore removes the 40 repeated discrepancies beyond the first and reduces replanning from 20 to 5 calls in this subset. Fig. 8a makes this contrast explicit.

For the other 45 faults, Oracle returns NO_PLAN in every case. RVREPLAN and State-only return the same result, with no timeout, recovery-limit termination, planner error, or framework error. These cases represent genuine capability-loss situations rather than failures of the runtime-verification mechanism. In particular, RVREPLAN still detects all 45 discrepancies and repairs 44 of the 45 action models exactly before the planner establishes that the goal is unreachable.

Across all 50 faults, the final action model maintained by RVREPLAN matches the injected faulty schema in 49/50 cases (98%). The 49 repairs are true-positive effect modifications; no false-positive schema modification is introduced. The single false negative occurs for a missing-effect fault in the Depot domain: the discrepancy is observable and is detected by the monitors, but the current diagnosis path replans without committing a model update. This negative case distinguishes successful failure detection from successful model repair and identifies a concrete limitation of the current diagnosis rules. State-only retains the nominal model by design, whereas Oracle matches the faulty model by construction.

5.2.3. *Runtime overhead and monitor lifecycle*

Table 5 characterises the runtime overhead of the executable RVREPLAN configuration across the 50 faulted runs. We report medians and interquartile ranges (IQR) because the costs vary across domains and fault locations. The diagnosis/model-repair phase includes regeneration and compilation of the candidate runtime monitors and dominates the current prototype latency. By comparison, Fast Downward replanning itself has a median of only 0.117 s in the main experiment. The same pattern is shown visually in Fig. 8b.

These figures characterise the current DejaVu-based prototype and should not be interpreted as a fair runtime comparison with State-only or Oracle. The latter are lightweight experimental controls and do not generate, compile, and launch the external monitors executed by RVREPLAN. The measurements instead identify monitor generation/compilation as the main engineering bottleneck and motivate caching, incremental regeneration, or a faster compatible monitoring backend as discussed earlier in the paper.

Successful replanning also requires the monitors to remain aligned with the new plan. Five main-experiment runs and two additional

Table 5
Runtime breakdown for the executable RVREPLAN arm over the 50 faulted runs. Values are median [IQR] in seconds.

Phase	Median [IQR] (s)
First detection	1.084 [0.991, 1.227]
Diagnosis/model repair	31.952 [31.112, 33.295]
Simulation	2.129 [1.364, 3.853]
Replanning	0.117 [0.115, 0.121]
Total recovery	34.378 [32.697, 37.635]
Execution + recovery	69.907 [66.015, 73.490]

Table 6
Planner sensitivity over ten representative faults. Times are medians in seconds.

Planner	Goal	NO_PLAN	Timeout	Replan	Execution
lama-first	1/10	9/10	0	0.117	68.938
seq-opt-bjolp	1/10	9/10	0	0.120	69.324
seq-opt-lmcut	1/10	9/10	0	0.117	71.622

planner- sensitivity runs produce replacement plans. In all seven cases, the execution evidence records a new plan identity, regenerated properties, newly compiled monitor, and monitor reload before execution resumes. The remaining faulted runs terminate with planner-certified NO_PLAN and therefore require no replacement-plan monitor.

5.2.4. *Planner sensitivity*

The previous evaluation showed that optimal Fast Downward configurations could become expensive after a runtime update. To separate this planner-dependent issue from the behaviour of RVREPLAN, we evaluate ten representative faults—one missing and one additional effect from each benchmark domain—with the satisficing lama-first configuration and the optimal seq-opt-bjolp and seq-opt-lmcut configurations. Table 6 reports the results. Fig. 8c complements the table with a direct visual comparison of the outcome counts and median replanning times.

All three planner configurations produce the same per-fault outcome: one recoverable case reaches the goal, and nine cases are proved unsolvable. No planner times out and no planner or framework error occurs. The median replanning times are also similar on this subset. These results do not imply that optimal and satisficing search have equivalent complexity in general; they show that the previous timeout behaviour is not intrinsic to RVREPLAN and does not reappear on the expanded representative subset. RVREPLAN is planner-agnostic, so a time-critical deployment can use a satisficing planner or impose a replanning budget and fall back to one if an optimal search exceeds the available recovery time.

Finally, the expanded evaluation deliberately retains negative and boundary cases. One observable Depot fault remains unrepaired, and the five recoverable cases all originate from Satellite additional effects.

The experiments also remain within the deterministic, fully observable setting assumed by the current framework.

6. Conclusions

We have introduced RVREPLAN, a framework for bridging the gap between classical planning and execution, using automatically synthesised RV monitors that can update action specifications and reason about replanning. The main methodological contribution is a closed loop that goes beyond detecting plan failure or replanning from an updated state by identifying discrepancies in the action model itself, repairing the affected operator, evaluating the repair, and using the revised domain to guide subsequent execution. One of our main goals with the modular implementation of RVREPLAN is to position it as a benchmark tool for experimenting with and evaluating planning, execution, replanning, and plan repair techniques. The notion of *Connector* and *Mapper* allows RVREPLAN to be completely detached from the real environment.

Taken together, the two experimental studies provide complementary evidence. The controlled characterisation exposes the detailed synthesis, replanning, and plan-change behaviour of RVREPLAN under deliberately recoverable scenarios, while the expanded study tests the corrected implementation over a broader benchmark, includes explicit comparison configurations (baseline and oracle), and distinguishes recoverable faults from genuine loss of planning solvability.

In future work, we plan to incorporate plan-repair monitors into our framework. These monitors will actively participate in the replanning process by attempting to repair the plan using information from runtime traces; for example, they can interrupt the planning process if they perceive a change in a proposition while replanning is underway, or they can validate the plans generated during replanning against new states using an automatic plan validation tool such as VAL (Howey et al., 2004). Additionally, we would like to investigate the use of monitors for plan repair, reducing the need to invoke a planner from scratch after every detected failure.

The current implementation assumes fully observable, deterministic environments to isolate interactions between runtime verification and replanning. Such settings encompass domains such as logistics and robotic scheduling, where predictable action outcomes are achieved. Extending RVREPLAN to partial observability, sensor noise, and stochastic effects is left for future work. The monitor-driven structure is compatible with such extensions, for example, through three-valued semantics or uncertainty-aware monitors. Once these extensions are done, the next step would be to evaluate the extended RVREPLAN in an integrated robotics or smart-environment system to provide stronger evidence of practical applicability.

CRediT authorship contribution statement

Rafael C. Cardoso: Conceptualization, Methodology, Validation, Formal analysis, Investigation, Visualization, Writing – original draft, Writing – review & editing; **Angelo Ferrando:** Conceptualization, Methodology, Software, Formal analysis, Writing – original draft.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

RVREPLAN code and all experiments are available at: https://github.com/AngeloFerrando/RVREPLan

References

Aineto, D., Celorrio, S. J., & Onaindia, E. (2019). Learning action models with minimal observability. *Artificial Intelligence*, *275*, 104–137. https://doi.org/10.1016/j.artint.2019.05.003

Aineto, D., Gaudenzi, A., Gerevini, A., Rovetta, A., Scala, E., & Serina, I. (2023). Action-failure resilient planning. In *ECAI 2023 - 26th European conference on artificial intelligence, September 30, - October 4, 2023, Kraków, Poland - including 12th conference on prestigious applications of intelligent systems (PAIS 2023)* (pp. 44–51). IOS Press (*vol. 372*). Frontiers in Artificial Intelligence and Applications. https://doi.org/10.3233/FAIA230252

Arora, A., Fiorino, H., Pellier, D., Métivier, M., & Pesty, S. (2018). A review of learning planning action models. *The Knowledge Engineering Review*, *33*, e20. https://doi.org/10.1017/S0269888918000188

Babli, M., Sapena, O., & Onaindia, E. (2023). Plan commitment: Replanning versus plan repair. *Engineering Applications of Artificial Intelligence*, *123*, 106275. https://doi.org/10.1016/j.engappai.2023.106275

Bartocci, E., Falcone, Y., Francalanza, A., & Reger, G. (2018). Introduction to runtime verification. In *Lectures on runtime verification - introductory and advanced topics* (pp. 1–33). Springer (*vol. 10457*). Lecture Notes in Computer Science. https://doi.org/10.1007/978-3-319-75632-5_1

Bauer, A., Leucker, M., & Schallhart, C. (2010). Comparing LTL semantics for runtime verification. *Journal of Logic and Computation*, *20*(3), 651–674. https://doi.org/10.1093/logcom/exn075

Bauer, A., Leucker, M., & Schallhart, C. (2011). Runtime verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology*, *20*(4), 14:1–14:64. https://doi.org/10.1145/2000799.2000800

Calvanese, D., De Giacomo, G., & Vardi, M. Y. (2002). Reasoning about actions and planning in LTL action theories. In *Proceedings of the 8th international conference on principles of knowledge representation and reasoning* (pp. 593–602). Morgan Kaufmann.

D'Angelo, B., Sankaranarayanan, S., Sanchez, C., Robinson, W., Finkbeiner, B., Sipma, H. B., Mehrotra, S., & Manna, Z. (2005). Lola: Runtime monitoring of synchronous systems. In *12th international symposium on temporal representation and reasoning (TIME'05)* (pp. 166–174). https://doi.org/10.1109/TIME.2005.26

Duret-Lutz, A., Renault, E., Colange, M., Renkin, F., Gbaguidi Aisse, A., Schlehuber-Caissier, P., Medioni, T., Martin, A., Dubois, J., Gillard, C., & Lauko, H. (2022). From spot 2.0 to spot 2.10: What's new? In *Proceedings of the 34th international conference on computer aided verification (CAV'22)* (pp. 174–187). Springer (*vol. 13372*). Lecture Notes in Computer Science. https://doi.org/10.1007/978-3-031-13188-2_9

Ferrando, A., & Cardoso, R. C. (2022). RVPLAN: Runtime verification of assumptions in automated planning. In *Proceedings of the 14th international conference on agents and artificial intelligence, ICAART 2022, volume 2, online streaming, February 3-5, 2022* (pp. 67–77). SCITEPRESS. https://doi.org/10.5220/0010776500003116

Fickert, M., Gavran, I., Fedotov, I., Hoffmann, J., Majumdar, R., & Ruml, W. (2021). Choosing the initial state for online replanning. In *Thirty-fifth AAAI conference on artificial intelligence, AAAI 2021, thirty-third conference on innovative applications of artificial intelligence, IAAI 2021, the eleventh symposium on educational advances in artificial intelligence, EAAI 2021, virtual event, February 2–9, 2021* (pp. 12311–12319). AAAI Press. https://doi.org/10.1609/aaai.v35i14.17461

Fikes, R. E., & Nilsson, N. J. (1971). Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, *2*(3), 189–208. https://doi.org/10.1016/0004-3702(71)90010-5

Fox, M., Gerevini, A., Long, D., & Serina, I. (2006). Plan stability: Replanning versus plan repair. In *ICAPS* (pp. 212–221). (*vol. 6*). https://aaai.org/papers/icaps-06-022-plan-stability-replanning-versus-plan-repair/.

Fritz, C., & McIlraith, S. A. (2007). Monitoring plan optimality during execution. In *Proceedings of the seventeenth international conference on automated planning and scheduling, ICAPS 2007, providence, Rhode Island, USA, September 22–26, 2007* (pp. 144–151). AAAI. https://aaai.org/papers/icaps-07-019-monitoring-plan-optimality-during-execution/.

Havelund, K., Peled, D., & Ulus, D. (2018). DejaVu: A monitoring tool for first-order temporal logic. In *3rd Workshop on monitoring and testing of cyber-physical systems, MT@CPSWeek 2018, Porto, Portugal, April 10, 2018* (pp. 12–13). IEEE. https://doi.org/10.1109/MT-CPS.2018.00013

Havelund, K., Peled, D., & Ulus, D. (2020). First-order temporal logic monitoring with BDDs. *Formal Methods Systems and Designs*, *56*(1), 1–21. https://doi.org/10.1007/s10703-018-00327-4

Helmert, M. (2006). The fast downward planning system. *The Journal of Artificial Intelligence Research*, *26*, 191–246. https://doi.org/10.1613/jair.1705

Howey, R., Long, D., & Fox, M. (2004). VAL: Automatic plan validation, continuous effects and mixed initiative planning using PDDL. In *16th IEEE international conference on tools with artificial intelligence* (pp. 294–301). https://doi.org/10.1109/ICTAI.2004.120

Lamanna, L., Saetti, A., Serafini, L., Gerevini, A., & Traverso, P. (2021). Online learning of action models for PDDL planning. In *Proceedings of the thirtieth international joint conference on artificial intelligence, IJCAI-21* (pp. 4112–4118). International Joint Conferences on Artificial Intelligence Organization. https://doi.org/10.24963/ijcai.2021/566

Lichtenstein, O., Pnueli, A., & Zuck, L. D. (1985). The glory of the past. In *Logics of programs, conference, Brooklyn college, New York, NY, USA, June 17–19, 1985, proceedings* (pp. 196–218). Springer (*vol. 193*). Lecture Notes in Computer Science. https://doi.org/10.1007/3-540-15648-8_16

Manna, Z., & Pnueli, A. (1989). Completing the temporal picture. In *Automata, languages and programming, 16th international colloquium, ICALP89, Stresa, Italy, July 11–15* (pp. 534–558). Springer (*vol. 372*). Lecture Notes in Computer Science. https://doi.org/10.1007/BFb0035782

Manna, Z., & Pnueli, A. (1992). The temporal logic of reactive and concurrent systems: Specification. Springer. https://doi.org/10.1007/978-1-4612-0931-7

Nau, D., Ghallab, M., & Traverso, P. (2004). Automated planning: Theory & practice. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. https://doi.org/10.1016/B978-1-55860-856-6.X5000-5

Pnueli, A. (1977). The temporal logic of programs. In *18th annual symposium on foundations of computer science, providence, Rhode Island, USA, 31 October - 1 November* (pp. 46–57). IEEE Computer Society. https://doi.org/10.1109/SFCS.1977.32

Stringer, P., Cardoso, R. C., Dixon, C., Fisher, M., & Dennis, L. A. (2023). Updating action descriptions and plans for cognitive agents. In *Proceedings of the 2023 international conference on autonomous agents and multiagent systems* AAMAS '23 (p. 2370–2372). Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems. https://doi.org/10.65109/ULOO1800