

CASP for Robot Control in Hybrid Domains

Ryan Young, Marcello Balduccini, and Ankush Israney

Drexel University

Abstract. The task of planning in hybrid domains has recently attracted considerable attention, motivated by the potential for useful practical applications. Lately, approaches have been developed that resulted in efficient planning algorithms. In this paper, we push the investigation further and (1) develop execution monitoring and diagnostic algorithms for hybrid domains based on Constraint ASP (CASP); (2) propose an agent architecture that ties together planning, execution monitoring, and diagnostics for hybrid domains; (3) demonstrate our approach on two simple, but non-trivial scenarios, including one on a physical robot.

1 Introduction

A *hybrid domain* is a domain that combines discrete and continuous dynamic behaviors. Many interesting problems involve reasoning about domains of this kind, especially in robotics. The task of planning in hybrid domains has recently attracted considerable attention, motivated by the potential for useful practical applications. Reasoning in hybrid domains requires rich models to capture the interactions between discrete and continuous change. Lately, breakthroughs in research have enabled the development of efficient planning algorithms capable of handling hybrid domains. However, execution monitoring and diagnostic reasoning tasks have not received as much attention.

In this paper, we push the investigation further, leveraging advances in Constraint Answer Set Programming (CASP) [3] to develop algorithms for execution monitoring and diagnostic reasoning in hybrid domains, including domains with non-linear dynamics. We propose an agent architecture tying together planning with the other reasoning components. When unexpected conditions arise, our architecture is not only capable of inferring their likely causes, but also of estimating the values of related numerical properties. Inferred values can then be used to find workarounds even when normal conditions cannot be restored.

Our approach extends techniques from Reasoning about Actions and Change (RAC) [7] and reduces reasoning to finding models of logical theories. We use the same formalization of knowledge (action description, goal, history, hypotheses) for all reasoning tasks. To demonstrate the effectiveness of our architecture, we implemented an agent control loop capable of running in simulated environments and of operating a physical robot. We demonstrate our architecture on two simple, yet non-trivial scenarios. Our first scenario is simulated, while our second scenario is performed by a physical robot – to the best of our knowledge, the first example of use of CASP for robot control in hybrid domains.

The paper is structured as follows. In the next section we present background information necessary to understand our work and contributions. Next, we introduce our agent architecture and reasoning algorithms. In the following section, we present two scenarios that demonstrate our architecture. Finally, we discuss related work and draw conclusions.

2 Preliminaries

In this section, we introduce CASP, a logic-based, declarative programming paradigm that we will use in the rest of the paper for the representation of hybrid domains and for the specification of reasoning algorithms. CASP extends Answer Set Programming (ASP) and retains its non-monotonic nature, but enables efficient solving of numerical problems. *Constants, terms, atoms, literals* are defined as usual in logic programming. Traditional logic programming variables are called *discrete variables*. The language also includes symbols for *numerical variables*. *Constraints* are expressions of the form $e_1 \circ e_2$, where \circ is a comparison operator (e.g., =, <) and e_1, e_2 are mathematical expressions over numerical variables, pre-defined mathematical functions and (numerical) constants. An *extended literal* is either a literal or a constraint. In the simplified version of CASP that we consider, a *rule* r is a statement:

$$h_1 \vee \dots \vee h_k \leftarrow l_1, \dots, l_m, \text{not } l'_1, \dots, \text{not } l'_n. \quad (1)$$

where h_1, \dots, h_k are extended literals, l_i, l'_i are literals and “not” is the so called default negation operator. The *head* of the rule ($head(r)$) is the set $\{h_1, \dots, h_k\}$, while $\{l_1, \dots, \text{not } l'_n\}$ is its *body* ($body(r)$). If $head(r)$ contains a constraint, then $head(r)$ must be a singleton. The intuitive meaning of (1) is that, if every l_i holds and no l'_i does, then at least one h_i must hold; additionally, if the head is a constraint, then the numerical variables that occur in it must be assigned values that make the constraint true. If the body is empty, the rule is called *fact* and \leftarrow is dropped. If the head is empty, then the rule is called *denial*, and intuitively means that its body must never be satisfied. A program is a set of rules. Programs not containing default negation are called *definite* programs. The semantics of CASP programs is defined in three steps, as follows.

A set A of extended literals is *consistent* if no two complementary literals, a and $\neg a$, belong to A . An extended literal l is *satisfied* by a consistent set of extended literals A (denoted by $A \models l$) if $l \in A$. A set $\{l_1, \dots, l_k\}$ of extended literals is satisfied by a set of extended literals A if each l_i is satisfied by A . A consistent set of extended literals A is *closed* under a definite program Π if, for every rule r of the form (1) such that $body(r)$ is satisfied by A , $head(r) \cap A \neq \emptyset$. A consistent set A is an *answer set* of definite program Π under the ASP semantics if A is closed under all the rules of Π and is set-theoretically minimal among all sets closed under all the rules of Π .

To define answer sets of arbitrary programs, we introduce the notion of *reduct* of a program Π with respect to a set of extended literals A , denoted by Π^A . The reduct is obtained from Π by deleting every rule r such that $l \in A$ for some

expression of the form *not l* from *body(r)*. When no rules are left that can be deleted, all remaining expressions of the form *not l* are removed from the bodies of the remaining rules. The notion of answer set under the ASP semantics can be extended to arbitrary programs by stating that a consistent set A is an *answer set of Π under the ASP semantics* if it is an answer set of Π^A .

For the final part of the definition of the semantics of CASP, we need the following terminology. An *assignment* is a pair $\langle x_i, a \rangle$, where x_i is a numerical variable and a is a numerical constant. A *compound assignment* is a set of assignments. A *complete assignment* w.r.t. a set X of numerical variables is a compound assignment such that every variable of X is assigned a (unique) value. A *solution* to a constraint γ is a complete assignment w.r.t. the variables in γ that satisfies γ . A solution to a set C of constraints is an assignment that is a solution to every $\gamma \in C$. Given an answer set A under the ASP semantics, $\eta(A)$ denotes the set of all constraints from A .

Definition 1. *A pair $\langle A, \alpha \rangle$ is an answer set of a program Π under the CASP semantics iff A is an answer set of Π under the ASP semantics and α is a solution of $\eta(A)$.*

In the rest of this paper, we drop the phrase “under the CASP semantics” whenever possible. Next, we describe the agent architecture and its components.

3 Agent Architecture

Our architecture is based on the agent control loop shown in Algorithm 1. We make the simplifying assumption that, while the environment may be partially observable, observations are correct and initial knowledge about discrete properties is complete. We assume the existence of a clock providing an *absolute time*. In our representation, each state has a duration, and is thus associated with a beginning time, β , and an ending time, ϵ , both expressed in terms of absolute time. In the following, we will sometimes refer to absolute times by the term *timepoint*. Durative actions can be expressed as a pair of a *start* and *stop* actions using the duration of the states between the start and stop to express the duration of the action. At the start of the loop, the agent is given a goal and knowledge about the initial state. Based on this information, the agent uses an externally provided function to check whether the current state satisfies the goal and, if so, the loop terminates.¹ Otherwise, the agent finds a plan. In our architecture a plan is a sequence of pairs, each consisting of a set of actions and a state duration (of the following state) for a step of the plan. Once a plan is found, the agent executes its first step. Actions are assumed to be executed instantaneously by the underlying control module. Then, the agent waits for the specified state duration before proceeding to the next step in the plan.

While the agent waits, it gathers observations and checks whether there are any contradictions with what the agent expects. If contradictions are detected,

¹ Goal-checking arbitrary continuous valued states is a difficult problem and is outside the scope of this work.

the agent attempts to explain the discrepancy. This is achieved via diagnostic reasoning. The diagnostic reasoning component of our architecture results in an *explanation*. An explanation is a set of hypotheses about either (a) properties that must have had a different initial value than expected, or (b) *exogenous actions* that must have occurred undetected. Once a hypothesis is found, the agent replans. if no unexpected observations are detected, the agent continues executing the original plan.

```

Input:  $G$ : goal;  $OBS$ : observations about the initial state
unexpected_obs_found := true; // the planner must be run
HIST := OBS; HYP := {}; curr_step := 0;
while not goal_reached( $G, HIST \cup HYP, curr\_step$ ) do
  if unexpected_obs_found then
    | % Plan
    |  $P := \text{plan}(HIST \cup HYP, curr\_step, G)$ ;
    % Execute part of the plan
     $\langle actions, state\_duration \rangle := \text{pop}(P)$ ;
    control.execute(actions);
     $HIST := \text{update}(HIST, actions)$ ;
    % Move to the next state
     $curr\_step := curr\_step + 1$ ;
    % Observe the environment for the duration of the state
    unexpected_obs_found := false;
    while not elapsed(state_duration) and unexpected_obs_found = false do
      |  $HIST := HIST \cup \text{control.observe}(curr\_step, \text{control.clock}())$ ;
      % Detect and explain any unexpected observations
      if unexpected_obs(HIST  $\cup$  HYP) then
        |  $HYP := \text{explain}(HIST, HYP, curr\_step)$ ;
        | unexpected_obs_found := true;

```

Algorithm 1: Agent Control Loop

3.1 Domain Representation

Our representation builds upon techniques from the research on RAC. We represent a state by a set of Boolean properties, beginning and end times of the state, and a set numerical properties. Note that the value of numerical properties may change over the duration of a state. Because of space considerations, we present a slightly reduced fragment of our representation, and omit some elements that are already present in the related work [5, 2].

Let \mathcal{B} be a set of *Boolean fluents*, \mathcal{N} be a set of *numerical fluents*, \mathcal{A} be a set of *agent actions* and \mathcal{E} be a set of *exogenous actions*. All actions in \mathcal{A} and \mathcal{E} are instantaneous. Durative actions are obtained by a pair of instantaneous *start/stop* actions, as discussed earlier. A (*Boolean*) *fluent literal* is a Boolean

fluent b or its negation $\neg b$. As usual, we represent the evolution of the domain in terms of a linear sequence of discrete states and, thus, we relate the effects of actions to steps in the evolution of the domain. The truth value of a fluent literal l at (discrete) step i is represented by the CASP literals $h(l, i)$ and $\neg h(l, i)$. Notice that truth value of fluent literals is expressed in terms of discrete steps. The value of a numerical fluent n , on the other hand, is expressed in terms of absolute time t , and is represented by a numerical CASP variable $v(n, t)$. Additionally, expressions $\beta(i)$ and $\epsilon(i)$ represent the beginning and ending timepoints of step i . Thus, the value of n at the beginning (resp., end) of step i is represented by $v(n, \beta(i))$ (resp., $v(n, \epsilon(i))$). Relation $within(t, i)$ holds if timepoint t is within the timespan of step i . The effects of actions are represented by CASP rules of the following types:

$$h(\mathbf{l}_0, I + 1) \leftarrow h(\mathbf{l}_1, I), \dots, h(\mathbf{l}_n, I), o(\mathbf{a}, I). \quad (2)$$

$$h(\mathbf{l}_0, I) \leftarrow h(\mathbf{l}_1, I), \dots, h(\mathbf{l}_n, I). \quad (3)$$

$$\leftarrow h(\mathbf{l}_1, I), \dots, h(\mathbf{l}_n, I), o(\mathbf{a}, I). \quad (4)$$

$$v(\mathbf{n}, T) \circ e(I, T) \leftarrow within(T, I), h(\mathbf{l}_1, I), \dots, h(\mathbf{l}_n, I). \quad (5)$$

$$ab(\mathbf{n}, I) \leftarrow h(\mathbf{l}_1, I), \dots, h(\mathbf{l}_n, I). \quad (6)$$

where: I is a variable ranging over non-negative integers, representing steps in the evolution of the domain; T represents absolute time²; $\mathbf{l}_0, \dots, \mathbf{l}_n$ are fluent literals; \mathbf{a} is an action; \mathbf{n} is a numerical fluent. Statements (2),(3), and (4) follow the structure of traditional laws from action languages for discrete domains, such as \mathcal{AL} [7]. Specifically, (2) is a *discrete dynamic law*, saying that, if \mathbf{a} occurs (relation o) when all of $\mathbf{l}_1, \dots, \mathbf{l}_n$ hold (relation h), then \mathbf{l}_0 holds in the following state.³ Statement (3) is a *discrete state constraint*, stating that, whenever $\mathbf{l}_1, \dots, \mathbf{l}_n$ hold, \mathbf{l}_0 also holds. Statement (4) is an executability condition, specifying the conditions under which an action cannot be executed.

The next two laws are novel. We develop them specifically for the representation of hybrid domains and to allow for multi-modal reasoning on such representation. Statement (5) is a *numerical state constraint*, intuitively describing how the value of numerical fluent \mathbf{n} changes during the duration of the state at step I in the evolution of the domain. Symbol \circ denotes an arbitrary mathematical comparison operator and $e(I, T)$ is a mathematical expression over numerical constants and expressions $\beta(I)$ and $v(\mathbf{n}, \beta(I))$. Finally, (6) is an *exception law*, which accompanies (5) and states that, under the given conditions, the value of the numerical fluent changes within the duration of the state.

It is worth stressing that the fact that \circ is not limited to equality in law (5) distinguishes our representation from that of comparable state-of-the-art approaches. Our approach provides more flexibility in the representation of actions.

² Technical readers may be concerned by the performance impact of having such a variable. As will become clear later, the domain of this variable represents a small, discrete set of timepoints, and, thus, does not significantly affect performance.

³ Representing combinations of Boolean and numerical conditions is possible with more advanced types of rules, as demonstrated in [2].

For example, if a certain object being pushed may travel at a speed that is anywhere between 2 and 3 *m/s*, the distance it covers may be represented by the law:

$$\begin{aligned} v(dist, T) &\geq v(dist, \beta(I)) + 2 \cdot (T - \beta(I)) \leftarrow within(T, I), h(pushed, I). \\ v(dist, T) &\leq v(dist, \beta(I)) + 3 \cdot (T - \beta(I)) \leftarrow within(T, I), h(pushed, I). \end{aligned}$$

As we will see later, this is especially useful for the purpose of diagnostic reasoning.

An *action description AD* is a set of laws together with the general axioms:

$$h(B, I + 1) \leftarrow h(B, I), \text{not } \neg h(B, I + 1). \quad (7)$$

$$\neg h(B, I + 1) \leftarrow \neg h(B, I), \text{not } h(B, I + 1). \quad (8)$$

$$v(N, T) = v(N, \beta(I)) \leftarrow within(T, I), \text{not } ab(N, I). \quad (9)$$

$$\beta(I + 1) = \epsilon(I). \quad (10)$$

$$within(\beta(I), I). \quad (11)$$

$$within(\epsilon(I), I). \quad (12)$$

where B ranges over Boolean fluents, N over numerical fluents, and I over steps. Rules (7) and (8) encode the well-known *law of inertia* for Boolean fluents. We introduce (9), which we call the *law of intra-state inertia*, to capture the intuition that the value of N should be assumed to stay the same within a state unless explicitly changed. The purpose of (6) is to block the application of (9) whenever the value of a fluent is known to change within a state. (10) formalizes the instantaneous nature of the actions. The last two axioms specify that the initial and final timepoints of a step are within the timespan of that step.

Initial state, history, hypotheses. The initial state is formalized as follows. As usual, the value of Boolean fluents in the initial state is encoded by CASP literals of the form $h(l, 0)$. If the value of numerical fluent n at the beginning of the initial state is known with certainty, then it is expressed by a fact:

$$v(n, \beta(0)) = \nu.$$

In this work, however, we are interested in investigating combinations of diagnostic reasoning and forms of uncertainty. Thus, we allow for the specification of assumptions about numerical fluents. The fact that a numerical fluent n is *assumed* to have had value ν at the beginning of the initial state is encoded by a rule:

$$v(n, \beta(0)) = \nu \leftarrow \text{not } \neg expected(n).$$

Thanks to CASP's non-monotonic nature, the numerical fluent is assumed to have value ν in the initial state unless the reasoner concludes that there is evidence to the contrary. Such evidence is expressed by CASP literal $\neg expected(n)$.

A *history HIST* is the collection of observations about fluent values and action occurrences. We assume that agent actions are fully observable, while the observations about fluents and exogenous actions may be incomplete. For

simplicity, at this stage observations are assumed to be correct.⁴ A *history* is a collection of expressions of the form:

$$obs(b, true, i) \tag{13}$$

$$obs(b, false, i) \tag{14}$$

$$obs(n, t, i, \nu) \tag{15}$$

$$hpd(a, i) \tag{16}$$

Expressions (13) and (14) state that Boolean fluent b was observed to be true (resp., false) at step i . Expression (15) states that the value of numerical fluent n was observed to be ν at step i and timepoint t . Finally, expression (16) states that action a was observed to happen at step i .

A *hypothesis HYP* is a collection of statements about fluent values and exogenous action occurrences. A hypothesis is what the agent constructs in order to explain observations that contradicts its expectations. The hypothesized occurrence of an exogenous action is represented by an expression $o(e, i)$ where e is an exogenous action. The hypothesis that a numerical fluent n had an unexpected initial value is represented by an expression $\neg expected(n)$. Both histories and hypotheses are used by our execution monitoring and diagnosis algorithms.

In our approach, reasoning is reduced to finding answer sets of suitable programs. For example, let AD be an action description and consider an initial state that begins at timepoint τ_1 , ends at τ_2 , and whose fluent values are encoded by a set of statements ι . The possible successor states resulting from the execution of a are given by the answer sets of $AD \cup \iota \cup \{\beta(0) = \tau_1. \epsilon(0) = \tau_2.\} \cup \{o(a, 0).\}$.

3.2 Planning

Following the approach from [5, 2], planning is reduced to computing an answer set $\langle A, \alpha \rangle$ of program $HIST \cup HYP \cup \mathcal{M}_P \cup AD$, where \mathcal{M}_P follows the typical structure of ASP planning modules, consisting of choice rules and denials for the selection of occurrences of actions at the various steps up to a given maximum plan length. The approach is well-known [2].

Function `plan` from Algorithm 1 finds a shortest plan by iterating the process while increasing the maximum plan length. The plan is encoded by atoms $o(a, i) \in A$. Beginning and end timepoints of states are given by the values of $\beta(i)$ and $\epsilon(i)$ in α . The expected fluent values are encoded by atoms $h(l, i) \in A$ and by the values assigned to $v(n, t)$ by α .

It should be noted that concurrent durative actions are allowed in a straightforward way. In fact, the pair of *start* and *stop* actions that signal the beginning and end of a durative action can be interleaved with those of other durative actions, resulting in concurrent durative actions.

Execution monitoring and diagnostic reasoning in hybrid domains are discussed next.

⁴ This assumption can be lifted, but doing so is not the topic of the present paper.

3.3 Execution Monitoring and Diagnostics

Function `unexpected_obs` from Algorithm 1 reduces the detection of unexpected observations to the task of finding answer sets of suitable programs. The core of the task is the *monitoring module*, \mathcal{M}_M , which includes the rules:

$$\leftarrow \text{obs}(B, \text{true}, I), \neg h(B, I). \quad (17)$$

$$\leftarrow \text{obs}(B, \text{false}, I), h(B, I). \quad (18)$$

$$v(N, T) = V \leftarrow \text{obs}(N, T, I, V). \quad (19)$$

$$\text{within}(T, I) \leftarrow \text{obs}(N, T, I, V). \quad (20)$$

The first two rules encode the *reality check axiom* [1], which intuitively states that it is impossible for the observations about Boolean fluents to contradict the agent’s expectations. Rule (19) is a novel counterpart of the reality check axiom for numerical fluents. Intuitively, the rule states that, if n was observed to have value ν at timepoint t , then this must match the agent’s expectation about the value of the fluent, i.e. $v(n, t)$ must equal ν . Rule (20) links timepoints and steps as provided by the available observations.

This approach can be easily elaborated to accommodate more sophisticated monitoring. For instance, it is often unrealistic to expect that the observed value of numerical fluents will match perfectly the agent’s expectations. Thus, rule (19) can be replaced by:

$$v(N, T) \geq V - E \leftarrow \text{obs}(N, T, I, V), \text{error}(N, E). \quad (21)$$

$$v(N, T) \leq V + E \leftarrow \text{obs}(N, T, I, V), \text{error}(N, E). \quad (22)$$

$$\neg \text{error}(N, 0) \leftarrow \text{error}(N, V), V \neq 0. \quad (23)$$

$$\text{error}(N, 0) \leftarrow \text{not } \neg \text{error}(N, 0). \quad (24)$$

The first two rules intuitively state that, if n is observed to have value ν and the measurement error on fluent n is $\pm e$ (encoded by an atom $\text{error}(n, e)$), then the agent’s expected value for n , $v(n, t)$, must be $\nu - e \leq v(n, t) \leq \nu + e$. The next two rules state that the default measurement error is 0. The approach can easily be extended further, e.g. by accounting for relative measurement errors.

Given a set $HIST$ of observations and a set HYP of current hypotheses, function `unexpected_obs` reduces the task of checking for unexpected observations to that of checking whether the CASP program

$$HIST \cup HYP \cup \mathcal{M}_M \cup AD \quad (25)$$

has an answer set. If the program has no answer set, then some observations are unexpected. In that case, the explanations must be found, which is accomplished by function `explain`, described next.

Function `explain` (Algorithm 2) takes as input the history of the domain, the current set of hypotheses, and the current step. If needed, the function can completely revise the current hypotheses. This allows the control loop to commit to a single set of hypotheses for planning and execution purposes, but to backtrack

over it during diagnosis if later observations warrant that. The approach shown in Algorithm 2 considers possible subsets of the current hypotheses, in the order established by user-provided function *prioritize*. For each subset H selected, the function invokes a CASP solver (function *CASP_solve*) on the CASP program discussed below to find a set of hypotheses that includes H and explains the observations. If no such set can be found ($\Delta = \perp$), the function removes, from the remaining subsets of HYP , those that can be ruled out based on the failed attempt on H . For instance, it is easy to see that, if H did not lead to an explanation, any $H' \supset H$ can be dropped. At the core of the diagnostic task is the

```

Function explain(HIST, HYP,  $\gamma$ )
Input: HIST: history
         HYP: current hypotheses
          $\gamma$ : current time step
Output: a set of hypotheses or  $\perp$  if no explanation exists
HYP_space := prioritize( $2^{HYP}$ , HIST);
foreach  $H$  in HYP_space do
     $\Delta = \text{CASP\_solve}(\text{HIST} \cup H \cup \mathcal{M}_D \cup AD)$ ;
    if  $\Delta \neq \perp$  then
        | return extract_hyp( $\Delta$ );
    HYP_space := remove_subsumed( $H$ , HYP_space);
return  $\perp$ ;
    
```

Algorithm 2: Function *explain*

diagnostic module, \mathcal{M}_D , which consists of \mathcal{M}_M together with the rules:

$$\begin{aligned}
 o(E, I) \vee \neg o(E, I) &\leftarrow I < \text{curr_step}. \\
 \neg \text{expected}(N) \vee \text{expected}(N).
 \end{aligned}$$

The first rule states that the agent is allowed to hypothesize the unobserved occurrence of any exogenous action E at any past step. The second rule says that any numerical fluent may have had an unexpected initial value. An additional directive (omitted) ensures that cardinality-minimal explanations are returned⁵. An explanation can thus be found by computing an answer set of:

$$HIST \cup H \cup \mathcal{M}_D \cup AD. \tag{26}$$

Next, we illustrate the behavior of the architecture by means of two examples.

4 Scenarios

To illustrate our approach, we present two scenarios featuring planning, execution monitoring, and diagnostic reasoning in the presence of non-linear dynamics, the second of which was implemented on a robot.

⁵ This is achieved by the *#minimize* directive of the underlying CLINGO solver (<https://sourceforge.net/projects/potassco/files/guide/>).

4.1 Scenario 1: Battery Charging

Consider a domain in which a robot needs to charge a battery to full capacity. The robot can pick up the battery, insert it into a charger, remove it, start and stop the charger. If the battery is faulty, the robot can also repair it, which can be performed even with the battery in the charger, but only when charging is stopped. When a functional battery is being charged, its charge level changes over time according to the equation $lv(t) = 100 + e^{-\frac{t}{20}}(i - 100)$, where i is the initial charge level, t is the charge time, and e^x is the exponential function.⁶ If the battery is faulty, it may charge faster or slower than normal. Additionally, someone may bump into the robot at any time, causing the robot to become misaligned with the charger. In that case, inserting the battery causes a bad connection, and a slower charge rate than nominal. The robot is not allowed to intentionally charge a badly connected battery. As a remedy to a bad connection, the robot can remove the battery, which has the additional effect of realigning the robot. The corresponding action description includes laws such as:

$$\begin{aligned} v(lv, T) &= 100 + (v(lv, \beta(I)) - 100) \cdot \exp(-0.05 \cdot (T - \beta(I))) \leftarrow \\ &\quad \text{within}(T, I), h(\text{charging}, I), \neg h(\text{bad_conn}, I), h(\text{ok}(b), I). \\ v(lv, T) &< 100 + (v(lv, \beta(I)) - 100) \cdot \exp(-0.05 \cdot (T - \beta(I))) \leftarrow \\ &\quad \text{within}(T, I), h(\text{charging}, I), h(\text{bad_conn}, I), h(\text{ok}(b), I). \\ ab(lv, I) &\leftarrow h(\text{charging}, I). \end{aligned}$$

The first law describes the charging behavior of normal battery. The second describes the behavior of a badly connected battery. The last rule states that, if a battery is charging, its charge level may change within a state.

Let us consider how the agent copes with an undetected bump. Suppose that, initially, the battery is discharged, functional, out of the charger and the robot is properly aligned. The goal is to charge the battery to full capacity. At the beginning of Algorithm 1, `plan` is invoked to find a plan for charging the battery. It is not difficult to see that such a plan consists of the sequence of actions: pick up the battery, insert it, start charging, end charging. Start and end times are assigned accordingly. For instance, in our scenario, the time elapsed between the start-charging and the end-charging actions is of 272 units.

Next, the plan is executed and monitored as specified in Algorithm 1. Let us suppose that, 45 time units after the execution of the start-charging action, the robot observes that the charge level is 43. The observation is added to history *HIST* in the form of a statement $obs(lv, 45, 3, 43)$, where step 3 corresponds to the state between the start-charging and the end-charging actions. Next, the algorithm executes `unexpected_obs`. As can be easily seen from the above formulas, the expected charge level under the stated conditions is 89. Hence, it is not difficult to see that program $HIST \cup HYP \cup \mathcal{M}_M \cup AD$ is inconsistent. (At this point, $HYP = \emptyset$.) This triggers the execution of `explain`. An explanation is found by computing an answer set of (26).

⁶ We developed this equation as an approximation of the non-linear charging characteristics of various kinds batteries, e.g. <http://www.ti.com/lit/an/snva557/snva557.pdf>.

Possible explanations are a battery failure or a bump.⁷ Assuming that the former is selected, *HYP* is updated to include the hypothesis and the agent replans. The new plan instructs it to stop charging, repair the battery, and resume the charging process. Suppose that, in reality, the cause of the observed charge rate is a bad connection. Then, when the robot resumes charging, it will still observe an unexpectedly slow charge rate. History *HIST* is updated with the new observation and function `explain` is triggered again. For simplicity, suppose that user-supplied function `prioritize` is defined so that it selects the empty set first. In principle, this might be due to a sequence of two battery-fail actions, one of which after the battery was repaired. On the other hand, both observations of a slow charge rate could be due to the occurrence of a bump that went undetected. Because the diagnostic module finds cardinality-minimal diagnoses, it is obvious that the answer set of (26) will encode the latter explanation. *HYP* is now updated accordingly. The agent can now replan. According to the new plan, the robot will remove and reinsert the battery, which will result in a correctly aligned battery (unless a further bump occurs), and will then proceed to charging. If no unexpected events occur, the battery will be charged as expected.

4.2 Scenario 2: Robot Navigation

In this experiment, we demonstrate how our agent is capable of inferring features of the abnormal behavior of the domain and of using such information to its advantage to achieve its goal. Note that this is different from several state-of-the-art approaches, which are focused on dealing with the normal behavior of the domain and on restoring such normal behavior, e.g. through repair actions, when abnormalities occur.

This experiment has been carried out on a Pioneer P3-AT robot, tasked with traveling from location (0cm, 0cm) to (150cm, 150cm) and only allowed to perform 90° turns. All motion actions (forward, turn left, turn right) are durative, so the robot must plan their respective durations accurately to achieve the desired outcome.⁸ We assume instant acceleration to the max linear and angular speeds, empirically measured at 27.29cm/s and 17.33°/s. The actual speeds depend on the charge level of the robot’s battery⁹. The action description includes laws linking charge level, direction of travel and distance covered. For instance, the law for the distance along the horizontal axis is:

$$v(x, T) = v(x, \beta(I)) + v(lv, \beta(I)) \cdot s_{max} \cdot (T - \beta(I)) \cdot \cos(v(d, \beta(I))) \leftarrow \Gamma. \quad (27)$$

⁷ The diagnostic module will also find possible time steps at which the exogenous actions must have occurred. We omit the details to save space.

⁸ The domain is chosen for illustration only. We do not necessarily advocate using high-level planning for arbitrary navigation tasks.

⁹ In order to ensure repeatability of the experiments, we used a simulated charge level rather than the actual one. That is, the robot’s low-level control module was modified to reduce the speed proportionally to the value of the simulated charge level without notifying the high-level control module.

where x is a numerical fluent representing x coordinate of the robot, lv is the battery level, d is the direction (angle) of travel, s_{max} is the maximum linear speed, and Γ is a set of conditions that are satisfied while the forward action is in progress. It is worth noting the non-linear nature of the expression.

The robot’s location is observable, while the battery level is non-observable and assumed to be initially 100% by means of a rule:

$$v(lv, \beta(0)) = 100 \leftarrow \text{not } \neg\text{expected}(lv).$$

For simplicity, we assume that the level does not change during the experiment. Under the assumption of a full charge, the plan found by function `plan` is: forward for 5.5s¹⁰, turn left for 5.2s, forward for 5.5s. For illustration purposes, suppose that the robot moves forward and then observes its position before turning left. Also, suppose that the robot observes that its x coordinate is 80cm, causing a statement $obs(x, 5.5, 2, 80)$ to be added to *HIST*. Clearly, the addition of such a statement causes CASP program (25) to be inconsistent, since the expected value of x is 150cm. Hence, `explain` is triggered by Algorithm 1. It is not difficult to see that the answer set found by `explain` contains a literal $\neg\text{expected}(lv)$, indicating that the initial charge must have been different from its expected value. Such literal is added to the set *HYP* of hypotheses. Additionally, and remarkably, it can be shown that component α of the answer set includes an assignment of value to numerical fluent lv corresponding to a charge of 53%. That is, the diagnostic algorithm indirectly determines the initial charge level from (27) and from the observed distance covered. Note that this is achieved by means of the same general-purpose encoding used for planning and for the detection of unexpected observations.

Next, the robot replans. The process seamlessly takes into account the inferred battery level of 53% thanks to the content of *HYP*, which is passed to `plan`. Because of that, the plan returned is adjusted for the lower battery level, resulting in longer durative actions: first, the robot will move forward from its current position for an extra 4.8s, in order to finally reach (150cm, 0cm). Then, it will turn left for 9.73s. Then, forward for 10.3s. Assuming that no unexpected events occur from this point on, it is easy to see that this plan will allow the robot to reach its destination. It is worth stressing again that this is made possible by the architecture’s ability to infer the value of unobservable fluents from the available observations and to leverage the inferred knowledge in subsequent reasoning tasks.

The control loop is implemented in Python and uses CASP solver EZCSP 1.7.9. A video of the robot in action is at <https://goo.gl/KuLRHH>.

5 Related Work

While our work is situated in at an intersection of areas that is largely unexplored, there is a vast amount of literature on the individual topics. Due to space

¹⁰ Strictly speaking, this is encoded in the plan by a $start(fwd)$ action at timepoint 0, followed by $stop(fwd)$ at timepoint 5.5s.

considerations, we highlight only the approaches that are most closely related to ours. [1] proposes an agent architecture capable of planning, execution monitoring, diagnostic reasoning, and replanning. This approach is limited to discrete domains. [5] address hybrid domains, but does not investigate execution monitoring, diagnostics, or agent architectures. [2, 4] focus on planning in hybrid domains from PDDL+ domain specifications. Effects of actions on numerical properties do not include uncertainty; agent architecture, execution monitoring, diagnostics are not considered. Various papers discuss the use of logic-based architectures in robotics, such as [8, 6]. At the high level, however, they view the domain as discrete, and typically rely on low-level control or ad-hoc computations for dealing with any non-discrete features.

6 Conclusion

We proposed what to the best of our knowledge are the first CASP-based representation methodology and reasoning algorithms for execution monitoring and diagnosis in hybrid domains, including those with non-linear dynamics. We also presented an agent architecture tying together planning, execution monitoring and diagnostic reasoning. When unexpected conditions arise, our architecture is not only capable of inferring their likely causes but also of estimating the values of related numerical properties. Inferred values can then be used to find workarounds even when normal conditions cannot be restored. We demonstrated our approach on two non-trivial examples, one of which on a physical robot.

References

1. Balduccini, M., Gelfond, M.: Diagnostic reasoning with A-Prolog. *Journal of Theory and Practice of Logic Programming (TPLP)* 3(4–5), 425–461 (Jul 2003)
2. Balduccini, M., Magazzeni, D., Maratea, M.: A CASP-Based Approach to PDDL+ Planning. In: *Constraint Satisfaction techniques for planning and Scheduling (COPLAS16)* (2016)
3. Baselice, S., Bonatti, P.A., Gelfond, M.: Towards an Integration of Answer Set and Constraint Solving. In: *Proceedings of ICLP 2005* (2005)
4. Bryce, D., Gao, S., Musliner, D., Goldman, R.: SMT-based Nonlinear PDDL+ Planning. In: *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence (AAAI’15)*. pp. 3247–3253 (2015)
5. Chintabathina, S., Gelfond, M., Watson, R.: Modeling Hybrid Domains Using Process Description Language. In: *Proceedings of ASP ’05 – Answer Set Programming: Advances in Theory and Implementation*. pp. 303–317 (2005)
6. Erdem, E., Patoglu, V., Saribatur, Z.G.: Diagnostic Reasoning for Robotics Using Action Languages. In: *13th International Conference on Logic Programming and Nonmonotonic Reasoning*. pp. 317–331 (2015)
7. Gelfond, M., Lifschitz, V.: Action Languages. *Electronic Transactions on AI* 3(16) (1998)
8. Zhang, S., Yang, F., Khandelwal, P., Stone, P.: Mobile Robot Planning using Action Language BC with an Abstraction Hierarchy. In: *13th International Conference on Logic Programming and Nonmonotonic Reasoning* (2015)