Pursuing Intelligent Behavior in Cyber—Physical Systems by Lightweight Diagnosis

Martin Zimmermann, Franz Wotawa,* and Ingo Pill

Intelligence in its decisions is a trait that people have grown to expect from a cyber—physical system, in particular that it makes the right choices at runtime, that is, those that allow it to fulfill its tasks, even in case of faults or unexpected interactions with its environment. Analyzing how to continuously achieve the currently desired (and possibly continuously changing) goals and adapting its behavior to reach these goals is undoubtedly a serious challenge. This becomes even more challenging if the atomic actions a system can implement become unreliable due to faulty components or some exogenous event out of its control. Herein, a solution for the presented challenge is proposed. In particular, it is shown how to adopt a lightweight diagnosis concept to cope with such situations. The approach is based on rules coupled with means for rule selection that is based on previous information regarding success or failure of rule executions. Furthermore, Java-based framework of the lightweight diagnosis concept is presented, and the results obtained from an experimental evaluation considering several application scenarios are discussed. At the end, a qualitative comparison with other related approaches that should help the readers decide which approach works best for them is presented. An interactive preprint version of the article can be found here: https://www.authorea.com/doi/full/10.22541/au.163578445.51350502.

1. Introduction

Be it our smartphones having to last through the day, automated industry plants, or autonomous robots and cars, we expect all these systems to make smart and informed decisions to effectively and efficiently perform their tasks, regardless of encountered internal issues like faults and changes in the environment.

There are a variety of techniques that we can rely on for implementing such behavior, including calculi like the situation calculus [1,2] or dynamic planning concepts that allow us to react to changes in the environment [3]. Regardless of the technique used, in principle, we are searching for action sequences that allow us to achieve our current goals. In this context, we certainly not only have to check whether we correctly perceive and assess the environment, but in practice, we’re also likely to suffer from unreliable actions. With our work as presented in this article, we are focusing on the latter.

Reasoning about an issue’s exact origin(s), for example, with model-based diagnosis (MBD) [4,5], allows us to search for an ideal mitigation strategy. In practice, however, 1) there is seldom enough data to precisely isolate a problem’s source(s) so that we end up with a set of candidates and 2) the reasoning’s complexity might prohibit us from making fast decisions. Moreover, in MBD, we require in addition to observations of the system a system model that captures the behavior sufficiently to allow deriving diagnosis candidates.

Complementing detailed and complete diagnosis concepts like MBD, which can be considered being more heavy weighted, in recent years, spectrum-based fault localization (SFL) [6] has been gaining attention. With SFL we evaluate execution data about which component was involved in this or that observed behavior. The result is a ranking of how suspicious the individual components are for causing the failing behaviors. Traditionally, SFL has been employed for software debugging, but it was shown recently in the study by Pill et al. [7] how to translate the idea for a static diagnosis of knowledge bases used in automated reasoning. As will be discussed in the next section, there we observed which of the knowledge base’s rules were involved in the individual reasoning tasks and whether the tasks were successful. Because of the ease of implementing SFL and the smaller number of information required for diagnosis, SFL presents a representative of lightweighted diagnosis methodologies.

We elaborate on such related work in that we translate and extend the basic SFL concept to accommodate also dynamic, live settings. Our concept thus allows us to continuously evaluate the success rates of system actions via considering the success/failing of previously executed action sequences a.k.a. plans.
We continuously update a corresponding reliability measure for each action and use these data to 1) re-evaluate currently implemented action plans and 2) when reasoning about new plans for achieving future goals. We require only very limited data for our concept, that is, which sequences failed or succeeded in the past and which components were involved. From a technical perspective, we describe the system’s actions via specific rules, pre-, and postconditions and use these data for our reasoning.

While we do not isolate an issue’s exact source(s) (like with MBD), we will show in our evaluation that our compromise between preciseness and computational complexity allows us to dynamically, effectively, and efficiently cope with faults and other events that result in unreliable actions. As we will discuss, we rely solely on SFL for our reasoning and thus passively observed executions without deploying any exploratory component like in reinforcement learning (RL).

2. Preliminaries

MBD is undoubtedly a powerful technique for precisely isolating a problem’s origin(s). We need a special model for this reasoning though, and while MBD is complete with respect to this model, the entailed computations can become quite complex. That is, the diagnosis search space is exponential in the number of the health state variables that we have to introduce. The more health state variables we have, the more faults can be found, but the larger the search space is. Usually, we end up with more than one diagnosis matching the data, so that we have to choose one as a working hypothesis.

With SFL, we take a different approach and consider the involvement of components in failing and passing behavior. The reasoning then follows the idea that some component that is always involved in faulty behavior but never in correct behavior is very suspicious of being the source of the troubles (and vice versa). As components are usually involved in both faulty and correct behavior, as well as the possibility that some faulty components cancel each other out, leading to correct behavior, many similarity coefficients, for example, see other studies, are proposed for computing a component’s suspiciousness have been proposed.

An intrinsic advantage of considering multiple executions in SFL by default is that fault masking (when multiple faults lead to correct output observations) has less effects on the reasoning, that is, if the set of observed behaviors is representative enough to contain also behavior without the masking effect. If the set is indeed representative and the faults always mask each other, then we are possibly facing an equivalent mutant, so we might want to consider the “faults” as implementation alternatives. For their computation, we consider the corresponding execution data about which component was involved in which behavior (stored in a matrix also referred to as spectrum) and whether some behavior is violating or complying with our expectations (the so-called error vector). Based on the components’ suspiciousness values, we establish a ranking.

**Definition 1** An activity matrix or spectrum $A$ is an $n \times m$ matrix, where for each of the $n$ system components we have $m$ rows for $m$ considered behaviors $b_j$. A cell $a_{ij}$ takes the value 1 iff component $c_i$ is involved in $b_j$, and 0 otherwise.

**Definition 2** An error vector $e$ for some spectrum $A$ (Def. 1) is a vector of length $m$ ($a_1 \times m$ matrix) such that $e_j = 1$ iff $b_j$ in $A$ violates the expectations, and $e_j = 0$ otherwise.

From $A$ and $e$, we derive for each $e_j$ four frequencies, $n_{CN}(e_j)$, $n_{CE}(e_j)$, $n_{VN}(e_j)$, and $n_{VE}(e_j)$ that capture how many Correct and Violating behaviors (the rows) in $A$ some $c_i$ was Executed or Not. In Table 1, we summarized the calculation of the four frequencies.

From these, we can compute several similarity coefficients like Ochiai, Tarantula, or Jaccard for estimating a component $c_i$’s suspiciousness $D(e_i)$.

| Coefficient | Formula |
|-------------|---------|
| Ochiai      | $D(c_i) = \frac{n_{VE}(e_i)}{\sqrt{(n_{VN}(e_i) + n_{VE}(e_i)) \cdot (n_{VN}(e_i) + n_{CE}(e_i))}}$ |
| Tarantula   | $D(c_i) = \frac{n_{VE}(e_i)}{n_{VN}(e_i) + n_{VN}(e_i) + n_{CE}(e_i)}$ |
| Jaccard     | $D(c_i) = \frac{n_{VE}(e_i)}{n_{VN}(e_i) + n_{VN}(e_i) + n_{CE}(e_i)}$ |

For software, it is very easy to come up with these data. So while SFL was originally developed for that domain, it has been employed also in other contexts. In the study by Pill et al., it was shown, for example, that we can exploit SFL in the context of logic reasoning with knowledge bases. While a knowledge base is not a program that we execute in the traditional sense, one can record the rules that are used when reasoning about a problem and use these data to define the spectrum. The reasoning processes for individual problems with the same knowledge base then define $b_j$ for $A$. For defining the error vector, it was suggested to inspect whether one would derive a contradiction and whether one would fail to derive the expected conclusions.

In Section 3, we show how to extend this concept to a live setting and a continuous assessment of a system’s action reliability. Our aim will not be to establish a ranking about which rules fail in practice but rather to express our confidence in the individual rules working out as expected.

There are quite a variety of tools and reasoning engine techniques that we could have adapted for evaluating our reasoning concept in practice. As our motivation has been to identify and reason about reliable action sequences, we focused on an available framework that encodes actions into a knowledge base of rules. While we then reason with these rules describing the actions. This RBL (Rule-Based Language) framework available for Java programs was proposed in the study by Zimmermann et al. Some interesting aspects for us were that RBL not only allows executing sequences, but

**Table 1.** Four frequencies catching how often some component $e_i$ was involved in specific behavior $n_{aq}(e_i)$.

| Frequency     | Description |
|---------------|-------------|
| $n_{CN}(e_i)$ | # of correct behaviors (C) s.t. $c_i$ was not executed (N). |
| $n_{CE}(e_i)$ | # of correct behaviors (C) s.t. $c_i$ was executed (E). |
| $n_{VN}(e_i)$ | # of violating behaviors (V) s.t. $c_i$ was executed (E). |
| $n_{VE}(e_i)$ | # of violating behaviors (V) s.t. $c_i$ was not executed (N). |
there is also some functionality to continuously (re-)design the sequence during execution.

Planning with rules is of course not a new concept introduced by RBL and has been studied before, for example, in other studies.[11–13] The reason we chose RBL for demonstrating our concept is that already its original runtime engine allows deriving and executing some action sequence and it was also designed to exploit feedback from the execution in some continuous replanning concept. This made it an ideal candidate for adopting our concept of implementing an engine that allows making intelligent decisions where we continuously assess the situation, derive diagnostic data via a special SFL concept, and derive new plans that are most promising on achieving the desired goals.

In RBL, a system’s environment is modeled by a dynamic list of corresponding beliefs whereas the system is described via rules. Such a rule comprises its preconditions, its postconditions, an action (Definition 4), a repair routine (Definition 5), and a weight assigning our confidence in this rule’s success.

**Definition 3** A rule \( R = (G, P, a, r, w) \) consists of a finite set \( G \) of preconditions, a finite set \( P \) of postconditions, an action \( a \), a repair routine \( r \), and a weight \( w \), where \( G \) and \( P \) are nondisjoint sets. Rule \( R \) is guarded by \( G \) and can only be executed if all \( g \in G \) are (currently) known as beliefs. If executing \( R \) (and thus \( a \)) is successful, \( \forall p \in P \) are added to (or removed from) the runtime engine’s beliefs according. If it fails, \( R \)’s repair routine \( r \) is invoked.

**Definition 4** An action \( a \) is a function that interacts with the environment. It returns \( T \) (true) if the interaction was successful and \( F \) (false) otherwise.

**Definition 5** A repair routine \( r \) is a function that has to be designed by the user and repairs the runtime engine’s belief such as to reach a correct and coherent state.

**Definition 6** A finite plan \( \pi \) is a finite sequence of rules \( R_1, \ldots, R_n \), such that the individual rules’ preconditions are met and the desired goal is reached when executing \( \pi \). RBL’s Plan—Execute—Update cycle perfectly fits our concept with its three stages that we can adopt also for our SFL-based reasoning concept. As we will show in Section 3, we will consider our reasoning concept in the planning phase to generate reliable action sequences and update the spectrum in the update phase.

1) **Planning:** Search for a plan \( \pi \) with the highest chance of success (lowest costs related to a plan’s actions’ weights).
2) **Execution:** Execute a plan’s rule sequence and track the rules’ successfullness. If executing rule \( R \)’s associated action failed, a) \( \pi \)’s execution is terminated and b) the \( R \)’s repair routine is invoked leading to a coherent state.
3) **Update:** Update the rules’ execution history and recalculate the rules’ weights.

While we chose RBL for our implementation, we opted to keep our concept general and did not want it to be restricted to using RBL’s custom domain-specific language. Consequently, we decided to implement the planning domain definition language (PDDL)[14] for our front-end, to show that our approach is available and applicable to all domains compatible with PDDL (and that it likewise can be implemented also in other frameworks).

PDDL was introduced in 1998 as a domain-independent planning language compatible with many algorithms. The main components of PDDL are the domain description and the problem description.[14] In the first, we define a set of actions with preconditions and effects. These actions usually have parameters that are populated with predicates during planning. In the problem description, we describe the initial state and the goal state—also using predicates. The planning algorithm’s job is then to derive a plan (as an ordered list of actions and their corresponding parameter assignments) that when executed leads from the initial state to the goal state.

### 3. Using Diagnostic Reasoning to Compute a System’s Action Reliability and Foster Intelligent Behavior

As we outlined in Section 2, traditionally, SFL has been deployed in a static context where we consider a test suite’s execution for an a posteriori identification of faulty components. Our aim is quite different in that we focus on a live setting where we continuously collect new execution data. Constantly analyzing these data via diagnostic reasoning, we establish a reliability measure for each of the system’s actions (they act as our “components”), recognize failed action sequences, and when replanning (and when deriving future plans), we aim to select the most reliable actions (technically it is their rules) for achieving our goals, that is, those rules (and their sequences) that are least likely to fail.

The three-phase cycle implemented in RBL’s runtime engine as discussed in the preliminaries perfectly fits the demands for our control concept, so that we did not have to implement it ourselves but focused on adapting the engine for our reasoning. Also, other dynamic planning environments feature similar control concepts so that our approach could be easily adopted, thereby needing only some adaptations to accommodate our reasoning. RBL’s engine, for instance, used a quite different cost function (a specific weight model) and related plan optimization concept, so we needed to adapt the planning algorithm to support our own SFL-related reasoning. However, in general, once we translate the static SFL idea to a dynamic context, deploying it does not require massive changes in a corresponding runtime engine like that of RBL. That is, in principle, we need to continuously: 1) derive and execute a plan \( \pi \); to achieve the current goals, 2) get feedback about \( \pi \)’s success, and 3) compute the individual rules’ frequencies and in turn their suspiciousness/reliability to be considered when making future decisions (when deriving future plans).

Please note that we use the terms actions and rules interchangeably in this manuscript, as technically we reason with rules that describe the system’s actions.

Formally and from an abstract point of view, for adopting SFL in our dynamic context, we have to add another row to \( A \) for specifying which rules were part of \( \pi \); whenever some plan \( \pi \) failed or succeeded. Furthermore, we have to enlarge the error vector \( \varepsilon \) to report also whether \( \pi \) failed or not. From \( A \) and \( \varepsilon \), we can compute similarity coefficients via the formulae depicted in Section 2. In principle, also a sliding window could be used such as to only consider recent data—which might be desirable for some dynamic applications.

In practice, the computation is less complex as we can keep track of all the rules’ four frequencies’ values, and whenever a plan \( \pi \) fails or succeeds, we increase the appropriate frequencies by one and recalculate the coefficients. In our context, we do not establish a ranking with these values but consider them as reliability measures for the corresponding rule. This value describes
how likely a component is failing, so that like for standard SFL applications, we have that the lower the value, the less likely a component is to cause troubles. A plan’s reliability is then computed from the reliability of its individual actions as follows.

**Definition 7** A plan π′s reliability r(π) is the sum of the plan’s individual rules’ reliability.

In **Algorithm 1**, we explicitly illustrate the steps needed for our concept. We show the action calls and repair routine invocations associated with a rule and illustrate the required loops and decisions. Please note that function update_frequencies(E,res) serves to update the frequencies and subsequently recompute the reliability values for all the individual rules as outlined earlier. It has two arguments: a list E containing those rules that have been executed for plan π and res encoding π’s success.

Please note that if one would like to use a sliding window, she or he would also have to store data about whether Ri ∈ π for all rules R, and an executed plan π (i.e., the rows in A) in a FIFO buffer. When we learn data about a new πn, we might not only have to accommodate these new data, but some old πi might fall out of the window so that we have to remove its influence. Via implementing a corresponding FIFO buffer, we could accomplish this easily.

With our definition of a plan’s reliability, we obviously search for a plan π with the lowest value r(π). Consequently, we do not reason about plans of minimal length, but desire plans of minimal costs in terms of r(π), which directly relates to the risk of a plan’s failing. If there are multiple plans with the same “optimal” r(π), we select the one created first. Note that, in such a case also heuristics that choose a plan of minimal length, or one such that the contained rules’ maximum reliability value is the lowest could be adopted.

In this context, it is also important to note that our concept is orthogonal to the incorporated planning algorithm/concept—as long as one can use our simple risk-related cost function drawing

on results from lightweight diagnostics in that algorithm. So whether a derived plan π is globally or locally optimal (consider a complete vs greedy search) depends on the incorporated planning algorithm. Consequently, the planning stage is not in the primary focus of our presentation, but we focus on 1) the exploitation of diagnostic data that describe the reliability of a system’s action as well as on 2) how to exploit such data in the planning stage of a corresponding engine via a specific cost function in the form of a plan’s reliability (see Def. 7). While completeness and soundness in terms of finding a plan optimal in the context of the chosen heuristic (the suspiciousness coefficient like Ochiai) thus depend on the planning algorithm, we can easily deduce the complexity of our computations.

**Theorem 1** Computing a rule’s reliability coefficient is done in constant time, so that computing all of them is linear in the number of rules. Computing a plan π’s reliability is linear in the length of its rule sequence.

**Proof:** As we keep track of a component’s frequency values and only have to update them via simple additions and subtractions (the latter only in case of a sliding window), their computation and that of the chosen suspicious metric like Ochiai is in constant time. We do this for each rule so that we’re linear in their number for the entire computation. Computing a plan π’s reliability means summing up the contained rules’ values (see Def. 7) so that we are linear in the length of π’s sequence.

When implementing our concept in practice, there are some aspects that we have to consider in relation to the similarity coefficients, though. For instance, after a cold start, insufficient data would result in a division by zero or a value of zero when computing the coefficients. As we are using the values directly as reliability measures, and therefore for planning, in our proof-of-concept implementation, we assign small values (0.000001) as a rule’s reliability measure in such cases. This follows the idea that after a cold start, we assume that the rules are rather healthy than faulty.

As we stated in Section 2, PDDL is one of the most used planning description languages. To make our approach as universal as possible, we wanted to show that our approach can be used in combination with PDDL and therefore can be integrated into every system that uses PDDL. However, using PDDL for our approach is not straightforward. We can see from Lst. 1 that actions in PDDL are abstract actions still requiring concrete parameters when we want to execute that action. Because our approach uses the feedback from the actual execution of an action, it is a good idea to also reason about concrete actions and the current state of the world. Creating concrete actions from abstract actions is called grounding in different research areas. For our implementation of grounding, we have two important requirements. 1) The calculation of abstract actions to concrete actions has to be dynamic. 2) We have to be able to reuse already discovered concrete actions in the next plan.

To fulfill those two requirements, we implemented a reachability-based algorithm for computing concrete actions. Starting from the initial state, we 1) calculate which concrete actions could be executed from this state. For each such concrete action, we 2) calculate how it transforms the state. If the state is a previously encountered state, we 3a) link the state to the already encountered state. If this is a new state, we 3b) continue at (1). The result is a graph where the nodes represent all different states of the world, and the edges represent their transformation.
through concrete actions. We can now use the reliability score of the concrete actions as edge weights and use traditional path-finding algorithms to generate a plan. In our implementation, we use Dijkstra’s algorithm. Retaining this graph throughout different plans enables us to rediscover already used concrete actions.

It is important to note that with this approach, we reason with the maximum level of information about the concrete execution, that is, the state of the world before execution and the concrete action. This, of course, is not the only option. For example, reasoning only about abstract actions would also be possible, for example, if we know the environment does not influence the result. Nevertheless, using our SFL approach to represent the confidence in an action’s success would still be applicable.

4. Experimental Section

To evaluate our approach, we chose the scenario of a warehouse where an agent equipped with our intelligent reasoning system (further called intelligent agent) had to fetch randomly placed items while avoiding other agents. For representing such scenarios, we use a grid world consisting of discrete cells like those illustrated in Figure 1. Our intelligent agent can move around in its world via single actions for moving to the north, east, south, or west from its current cell, that is, if it is not blocked by a wall or another agent occupying the target cell (then it would remain in the current cell). Technically, we implemented our scenario as an extension to the OpenAI gym environment gym maze (https://github.com/MattChanTK/gym-maze).

The intelligent agent knew about the warehouse domain and scenario from PDDL descriptions (Lst. 1,2). In some experiments, also all walls were described in these files, and in some, the intelligent agent had to learn their location via the feedback of a failed move. Please note that it did not learn whether a move was blocked by an agent or a wall, so that these other agents added noise to the observed learning data.

All experiments used the same PDDL domain file shown in Lst. 1, where we described three actions. First, move enabled the intelligent agent to move as described earlier. Second, pickup allowed it to pick items up (if it is in the same cell), and third, via put it could put an item down in a put location.

The basic structure of all our PDDL problem files is shown in Lst. 2. Depending on the example configuration, specific atoms, that is, (connected room_X1_Y1 room_X2_Y2), might be omitted from the initial state, such as to indicate that there is a wall between the two cells. For each fetch task, the intelligent agent receives a new random item location, which is added to the PDDL initial state on the fly and a PDDL goal to bring the item to the put location. The intelligent agent then starts at (0/0) in the grid, has to go to the item’s location, pick it up, go back to the put location (0/0), and, finally, deliver the item by putting it down, thus fulfilling the PDDL goal. If it is necessary to replan, the intelligent agent will start from its current location.

Listing 1: PDDL Domain for all examples

define ( domain robot
strips)
(: predicates (at ? r) (connected ? r1 ? r2) (holding ? i) (itemat ? i ? r) (putlocation ? r))
(: action move
: parameter (? from ? to)
: precondition (and (at ? from) (connected ? from ? to))
: effect (and (not (at ? from)) (at ? to)))
(: action pickup
: parameter (? room ? item)
: precondition (and (itemat ? item ? room) (at ? room))
: effect (and (holding ? item) (not (itemat ? item ? room))))
(: action put
: parameter (? room ? item)
: precondition (and (putlocation ? room) (at ? room) (holding ? item))
: effect (and (itemat ? item ? room) (not (holding ? item))))
Listing 2: PDDL Problem for Shelves a priori
(define (problem strips_robot)
  (:domain robot-strips)
  (:objects room 0 0 room 0 1 [... item])
  (:init (at room 0 0)
    (location room 0 0)
    (connected room 0 0 room 0 1)
    (connected room 0 1 room 0 0)
  [...]
  (:goal (and)))

We used different warehouse sizes (5x5, 8x8, or 11x11 cells) and numbers of other agents (0, 1, 4) in our experiments. The last parameter of a configuration was the setup of the experiment in terms of walls and an agent’s a priori knowledge of them as described later. When conducting the experiments, we investigated 100 different random fetch sequences for a specific configuration and reported the average values. Each such fetch sequence consisted of 100 fetch tasks. We also showed the average performance for each of them (over the 100 runs), such as to investigate the performance increase experienced. Please note that after finishing a fetch sequence, the learnt knowledge was discarded. For reason of space, we report only on a few selected configurations in this section. The results for all configurations and the code for the experiments are available on GitHub (https://github.com/martinzimmermann/RBL-test-programs/releases/tag/CPS-RTSA).

Shelves a priori: The grid world contained shelf cells that the agent cannot enter and around which two normal cells were placed (see Figure 1 on the left). Items can only be located next to a shelf. For this setup, the PDDL problem contained the shelves’ location (Lst. 2), so that an agent can move around efficiently. The challenge of this setup was that multiple agents operate in the same warehouse. The intelligent agent knows nothing about their locations and can only learn about them by colliding with them. Still, the intelligent agent was expected to fetch items efficiently.

Shelves a posteriori: The setup was similar to the previous one, but the PDDL problem file did not contain data about the shelves. Thus, these data were learnt by the intelligent agent via the move actions’ reliability for the neighboring cells such as to be able to move around in the grid efficiently (while still having to avoid other agents).

Maze: The third setup was a randomly generated maze (see Figure 1 on the right), where the intelligent agent does not know the layout of the maze but has to learn it through colliding with walls. Please note that as corridors were only one cell wide, it was not possible to bypass other agents. Thus, there were no other agents in this setup.

5. Results

In Figure 2, we report the average number of steps needed per item fetch over all 100 runs. We see that for all given configurations, the intelligent agent in the a posteriori setup needs fewer steps over time. This confirms our hypothesis that with our SFL approach, the used reliability measurement enhances planning. We also see that this task gets more difficult when adding more agents, as this generates more random noise. The random noise makes it harder for the agent to distinguish between temporary failures (i.e., other agents) and permanent faults (i.e., shelves).

For a priori, we are not able to confirm such an improvement. This is no surprise, as for this setup, the only unknown information about the world is the movement of the other agents. The movement is random and not learnable by our intelligent agent, as random behavior is, in general, not learnable. The small variance of steps needed can be explained by the random generation.
of the item locations. In Figure 2, we see that also the minimum, maximum, and median steps needed for a priori stay consistent over time.

One of our main focus points was to compare the a priori and a posteriori setups. Figure 3 shows that the a posteriori setup performs worse than the a priori setup. This is due to the fact that the problem of solving the a posteriori setup is much harder. First, it consists of more possible actions (moves through shelves are also considered during planning for a posteriori), and second, much information about the world, that is, the location of the shelves, is unknown to the intelligent agent. It is remarkable that for the $8 \times 8$ a posteriori configuration with 0 agents, a similar performance as a priori is reached after only around 100 fetches (Figure 3a). This could be due to the reason that using 0 agents make the scenario static, although still not known by the agent. For the other configurations, we also see a strong trend toward the performance of the a priori configurations. However, in our experiments, they never reach the same performance. It is not clear if just more fetches, meaning more training data, are needed to learn to distinguish between shelves and agents, or if they will never converge toward the a priori performance. To answer this, further experiments with longer fetch sequences are necessary.

During our investigation, we could not yet explain why, for most configurations, the performance of the first few fetches gets significantly worse before the performance gets better again. The only connection we could draw was that we sometimes saw similar behavior while performing RL in a different domain. Further research is necessary to find the root cause of this behavior. However, this was not a major concern for us, as for all configurations, in the end, we performed better than the first fetch.

In Figure 2, we show the maximum, median, and minimum number of steps required per fetch over the 100 runs. Interestingly, the median, similar to the average, of a posteriori converges toward the median of a priori. The maximum, in contrast, does not get smaller over time. The reason for this could be that the world is not sufficiently explored to calculate reliable plans for all locations after only 100 fetches. With roughly 100 cells, where an item could be placed, this seems plausible. Similarly, there is a high chance that an item location is close to the start when considering 100 runs. Having an item close to the start would explain the very stable minimum for both a posteriori and a priori.

In Table 2, we compared average total steps, total plans, and total runtime summed over a whole run, that is, 100 fetches, for the $11 \times 11$ grid a priori and a posteriori configuration. For the configuration a priori, 0 agents, the intelligent agent only needs 100 plans. That means no replanning was necessary. This was expected as everything about the world is known. As the number of agents increases, the number of replans increases. More replans are needed because there is a higher chance for the intelligent agent to collide with another agent. For the a posteriori setup, we see a similar increase in total plans and steps between 0 agents and 4 agents. However, the difference between the a priori and a posteriori is significant. One explanation for the difference could be that, for a priori, the information about the shelves is known, and the intelligent agent only collides with agents. In contrast, for a posteriori, the intelligent agent, in the beginning, collides mostly with shelves. Because of lack of knowledge, the agent generates a plan that bypasses the shelf by just one block. Mostly, a shelf is right next to another shelf. This, in turn, leads to another collision with only a single step taken. The average steps per plan also support this explanation.

![Figure 3](image-url)  

Figure 3. Results of selected configurations. The x-axis shows the specific fetch of the fetch sequence. The y-axis shows the average number of total steps needed for this fetch over all 100 fetch sequences. We can see that for configurations with 0 Agents, the a posteriori avg. steps converge toward the a priori avg. steps fast (a,b). However as the number of agents increases, this gets slowed down (c,d).
From a priori, this is, on average, 15 steps per plan, and for a posteriori, this is, on average, only 2 steps per plan.

From Table 2, we see that the runtime mostly correlates with the number of plans needed. Although we implemented some improvements for the RBL planning algorithm, it is still by far the largest runtime bottleneck. Only a fraction of the planning time is used for the SFL reliability calculation, which only increases with the number of possible actions for which the reliability should be calculated, as we proved in Section 3. Combining the SFL approach with a better planning algorithm would for sure result in better runtime.

For the Maze, there were improvements similar to the shelves a posteriori configurations. This shows that we can also improve our plans in very complex environments.

6. Comparison with other Approaches

In this section, we compare RBL to other approaches in the literature. Because the different approaches excel in different areas, and no other approach focuses on the same topic as RBL, we performed a qualitative comparison between the approaches. First, we describe the other approaches and compare them with RBL. Afterward, we compare the approaches all together considering seven different characteristics.

6.1. RL

RL gained huge recognition as being able to produce agents that can dynamically act in diverse environments. Typically RL solves Markov decision problems where the agent tries to maximize the cumulative reward. Usually, this is done by first training an agent on a huge amount of sample data and then deploying it to a real situation. Although it is possible to train an RL agent also during operation, this is usually not done as the agent’s learning performance greatly depends on the exploration versus exploitation tradeoff. The biggest difference between RL and RBL is that RL does not use any a priori knowledge and, therefore, always starts learning from scratch. With the exploitation of the knowledge given to RBL, it is possible to deploy RBL to a real situation directly and solve the problem right from the start. Even in the case that situations were unforeseen in the a priori knowledge, RBL can learn to circumvent these and stay operational. In contrast, RL agents can achieve much better performance than RBL on a given task. During training, RL agents also take exploratory actions, which lead them to acquire new knowledge about the world. RBL only takes exploratory actions when a failure occurs and only in the scope of the knowledge provided.

6.2. POND

POND provides an interesting approach to solve partial-observable and nondeterministic planning problems. It combines different search techniques and heuristics and switches between them dynamically. Furthermore, it uses a base representation for the problem from which other representations can be calculated, which are then used for the search or heuristic.

Compared with RBL, it probably performs better when everything about the problem is known at the start time. However, POND lacks the capability to use feedback from the execution to reevaluate its plans, and it is also not able to learn new information.[15]

6.3. FF-Replan

FF-replan was the winner of the 2004 International Probabilistic Planning Competition. It achieves this by first constructing a deterministic planning problem out of the probabilistic planning problem and replanning when it encounters a state that differs from the expected one. The conversion from a probabilistic plan to a deterministic plan is either done by single outcome or all outcome. Single outcome chooses a single action among many probable ones depending on a heuristic, and all outcome creates a separate action for each. FF-replan in all-outcome mode is quite similar to our approach. In RBL, we are not concerned with probabilistic actions per se. However, we permit each action to fail without knowing before which action and when the action will fail. We could emulate such behavior in FF-replan by adding a fail outcome to every action. However, FF-replan would still not learn from failures like RBL and would probably get stuck as soon as reality would not conform to its knowledge (e.g., a wall is in reality where there is none in the model).[16]

6.4. PRM-RL

PRM-RL combines probabilistic roadmaps (PRM) and RL. It first trains an RL agent via Monte Carlo selection on a similar environment as the target environment to get an agent that can successfully move in the environment. After this step, the agent is deployed to the target environment, and the PRM builder creates a PRM based on a uniform sampling of the agent’s movement. Only collision-free point-to-point navigation is retained in the PRM. After these two training steps, PRM-RL can successfully navigate the target environment. Although PRM-RL currently lacks the ability to learn in operation like RBL, it is not hard to imagine that the PRM builder could also be run as soon as the PRM model differs from the environment and therefore signals that a fault occurred. A benefit of PRM-RL is that it does not need a priori knowledge, and it infers everything from training. However, the RL agent and the PRM builder need this training to function properly, compared with RBL, which can be used without training at all.[17]
For the characteristics, we selected “Needs model,” “Needs Probabilities,” “Needs Training,” “Learns during operation,” “Performs Exploration,” “Failure resilient,” and “Guarantees.” We think these are the most important characteristics when someone wants to decide which approach is to be used. Our results are presented in Table 3, and a detailed description of the characteristics can be found.

**Needs model:** Describes if the approach needs a model to be usable. This can be in the form of PDDL, PDDL-like, or other nonformal information. Usually, approaches that have this model available perform better than others as they do not have to learn a model first. However, it is not always easy to get a model. ✓ means the approach needs this information, and × means the approach does not need the information.

**Needs probabilities:** Describes if the approach needs to know probabilities of the nondeterministic actions to function. Like the approach before, this information is often not easy to obtain or even impossible in a dynamic scenario. ✓ means the approach needs information, and × means the approach does not need the information.

**Needs training:** Describes how much training the approach needs before it can be used. ✓ means training is required before the approach can be used, for example, in a simulation, × means the approach either does not need training or will learn during operation.

**Learns during operation:** Describes if the approach can learn during operation. ✓ means that the approach will learn during operation, × means the approach is fixed during operation.

**Performs Exploration:** Describes if the approach can perform exploratory actions. This is useful if only partial information is available, and for example, new actions are tried or actions for knowledge gain are performed.

**Failure resilient:** Describes if the approach can handle unforeseen circumstances. For example, if the approach is able to adapt if a fault occurs during operation. ✓ means the approach can deal with unforeseen circumstances, and × means the approach will fail if an unforeseen circumstance occurs.

**Guarantees:** Describes if the approach gives some guarantees.

**FF-Replan:** Because FF-replan only selects actions with a non-zero percent chance of leading to a goal, FF-replan is guaranteed to reach the goal eventually if there are no dead ends and the environment is equivalent to the provided model.

**RBL:** Similar to FF-replan also, RBL takes only actions that have a chance to lead to a goal. However, because RBL also updates the reliability of the actions, the environment does not have to be equivalent to the provided model. Therefore, we can guarantee that an agent with RBL will eventually reach the goal if there are no dead ends and there is a possible action sequence in the model that would lead to the goal, meaning, as long as there are redundancies in the model of which not all are blocked.

### Table 3. Qualitative comparison of different approaches.

| Approach | Needs model | Needs probabilities | Needs training | Learns during operation | Performs Exploration | Failure resilient | Guarantees |
|----------|-------------|---------------------|----------------|-------------------------|----------------------|------------------|------------|
| RBL      | ✓           | ×                   | ×              | ✓                       | ×                    | ✓                | ✓          |
| RL       | ×           | ×                   | ✓              | ×                       | ×                    | ✓                | ✓          |
| POND     | ✓           | ✓                   | ×              | ✓                       | ×                    | ✓                | ×          |
| FF-Replan| ✓           | ✓/×                 | ×              | ✓                       | ×                    | ✓                | ✓          |
| PRM-RL   | ×           | ×                   | ✓              | ×                       | ✓                    | ×                | ×          |

7. Related Work

Nilsson presented in his work[18] the concept of Teleo-Reactive programs, which are formalism of action sequences an agent can take to reach goals in uncertain environments. Teleo-Reactive programs are an ordered list of actions with preconditions. The first action of this list, whose preconditions are met, is executed indefinitely. Through clever construction of Teleo-Reactive programs, an agent can then deal with uncertainty.

Krenn and Wotawa[19] proposed a similar approach. Instead of just using the first rule of the list, the rules’ selection frequency could be dynamically updated during operation. The rules’ selection frequency was in this approach based on biological processes of DNA transcription. Furthermore, rules do not have preconditions only, but also postconditions.

Our approach builds upon this line of research. Our approach extends this further with two main contributions. First, we created an interface to PDDL, enabling the approach to apply to a wide range of already existing models. Second, instead of the biological inspiration, we use SFL, which was already proven to be successful at detecting faulty components in software testing, which is more akin to the problem of distinguishing faulty from nonfaulty actions.

Another related line of research is replanning and plan repair. Other studies[20,21] use feedback from the execution to change the current beliefs of the system. Wilkins[22] plans multiple plans and chooses a new one when one fails. Draper et al.[23] handle uncertainty by incorporating conditions into the plan. Georgeff follows[24] a similar approach and uses partial planning and delayed decisions.

All those approaches have in common that, in case of a failure, they have to either change the beliefs, locate the fault, or gather additional information about the environment to choose a (new) plan. In contrast, our approach does not have to locate the fault but rather uses statistics to locate the plans’ reliable actions. In particular, the calculation with SFL is very inexpensive and can be adapted for different approaches. Even extending former-mentioned approaches should be possible as long as we can integrate a cost function into the planning algorithm. To the best of our knowledge, we do not know of a similar approach for replanning.

Finally, also RL is related to our approach. RL, as previously mentioned, is capable of solving a wide variety of tasks. Modern RL consists mostly of two intertwined research directions. First,
try-and-error learning inspired by Minsky.\textsuperscript{[25]} Second, optimal control, which has its beginnings in the 1950s, mostly by Bellman, Bellman\textsuperscript{[26]} proposed an approach, dynamic programming, which could solve optimal control programs. However, this approach did not scale well to higher dimensionalities. Sutton and Barto took inspiration from these two approaches and mixed it with temporal difference to create the modern RL.\textsuperscript{[27]} A lot of improvements and demonstrations were made over the last few years.\textsuperscript{[28,29]}

However, a core problem still remains that RL agents usually need many training samples till they are operational. Also, little research about how to bootstrap an agent with models like PDDL was done. Our approach uses a model as a bootstrap process and research about how to bootstrap an agent with models like PDDL bases are novel in general, combining both and adopting our context are indeed novel contributions that lead to new insight in SFL, including exploration stages with specific strategies. That is, entirely unlimited exploration could be exploited to gather broader knowledge at the cost of performance in the tasks themselves but also limiting the exploration to plans that deviate in performance only within some boundary $\epsilon$ to the optimal one could provide a more limited but still more educated picture. In such future research, it would also be interesting to consider effects from temporal considerations when associating the blame of a plan’s failure to individual actions or considering previous executions from less important in the SFL spectrum as recent ones.

Acknowledgements

The financial support by the Austrian Federal Ministry for Digital and Economic Affairs and the National Foundation for Research, Technology, and Development is gratefully acknowledged.

Conflict of Interest

The authors declare no conflict of interest.

Data Availability Statement

The data that support the findings of this study are available from the corresponding author upon reasonable request.

Keywords

diagnostic reasoning, replanning, self-adaptation, spectrum-based fault localizations

Received: November 2, 2021

Revised: December 7, 2021

Published online: December 26, 2021

\begin{thebibliography}{99}
\bibitem{1} C. Boutilier, R. Reiter, M. Soutchanski, S. Thrun, in 17th National Conf. on Artificial Intelligence and 12th Conf. on Innovative Applications of Artificial Intelligence, 2000, pp. 355–362.
\bibitem{2} S. Gspandl, I. Pill, M. Reip, G. Steinbauer, A. Ferrein, in IJCAI 2011, Proc. of the 22nd Int. Joint Conf. on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16–22, 2011, 2011, pp. 900–905.
\bibitem{3} D. Connell, H. M. La, in 2017 IEEE Int. Conf. on Systems, Man, and Cybernetics (SMC), 2017, pp. 1429–1434.
\bibitem{4} J. de Kleer, B. C. Williams, Artif. Intell. 1987, 32, 97.
\bibitem{5} R. Reiter, Artif. Intell. 1987, 32, 57.
\bibitem{6} R. Abreu, P. Zoeteweij, A. J. C. van Gemund, in ASE, IEEE Computer Society, Los Alamitos 2009, pp. 88–99.
\bibitem{7} I. Pill, F. Wotawa, in 2018 IEEE Int. Symp. on Software Reliability Engineering Workshops, ISSRE Workshops, Memphis, TN, AAAI Press, Paolo Alto, CA October 2018, pp. 192–199.
\bibitem{8} R. Abreu, P. Zoeteweij, R. Golsteijn, A. J. C. van Gemund, J. Syst. Softw. 2009, 82, 1780.
\bibitem{9} J. A. Jones, M. J. Harrold, in ASE, ACM, New York 2005, pp. 273–282.
\bibitem{10} M. Zimmermann, F. Wotawa, Softw. Quality J. 2020, 1.
\bibitem{11} A. L. Blum, M. L. Furst, Artif. Intell. 1997, 90, 281.
\end{thebibliography}
[12] R. E. Fikes, N. J. Nilsson, *Artif. Intell.* **1971**, *2*, 189.

[13] H. Kautz, B. Selman, in *Proc. of the Thirteenth National Conf. on Artificial Intelligence—Volume 2, AAAI’96*, AAAI Press, Menlo Park **1996**, pp. 1194–1201.

[14] M. Ghallab, A. Howe, C. Knoblock, D. Mcdermott, A. Ram, M. Veloso, D. Weld, D. Wilkins, *PDDL—The Planning Domain Definition Language*, **1998**.

[15] D. Bryce, in *ICAPS 2006*, **2006**, p. 58.

[16] S. Yoon, A. Fern, R. Givan, in *Proc. of the Seventeenth Int. Conf. on Automated Planning and Scheduling, ICAPS’07*, AAAI Press, Menlo Park **2007**, pp. 352–359.

[17] A. Faust, O. Ramirez, M. Fiser, K. Oslund, A. G. Francis, J. Davidson, L. Tapia, *CoRR* **2017**, abs/1710.03937.

[18] N. J. Nilsson, *J. Artif. Int. Res.* **1994**, *1*, 139.

[19] W. Krenn, F. Wotawa, in *Intelligent, Fault Adaptive Control of Autonomous Systems*, Springer Netherlands, Dordrecht, **2009**, pp. 175–188.

[20] P. Aschwanden, V. Baskaran, S. Bernardini, C. Fry, M. Moreno, N. Muscettola, C. Plaunt, D. Rijsman, P. Tompkins, in *Spacecraft Autonomy: Using AI to Expand Human Space Exploration, Papers from the 2006 AAAI Fall Symp.*, volume FS-06-07 of *AAAI Technical Report*, AAAI Press, Menlo Park **2006**.

[21] S. Chien, R. Knight, A. Stechert, R. Sherwood, G. Rabideau, in *Proc. of the Fifth Int. Conf. on Artificial Intelligence Planning Systems, AIPS’00*, AAAI Press, Menlo Park **2000**, pp. 300–307.

[22] D. E. Wilkins, *Practical Planning: Extending the Classical AI Planning Paradigm*, Morgan Kaufmann Publishers Inc., San Francisco, CA **1988**.

[23] D. L. Draper, S. Hanks, D. S. Weld, *CoRR* **2013**, abs/1302.6801.

[24] M. P. Georgeff, A. L. Lansky, in *Proc. of the Sixth National Conf. on Artificial Intelligence—Volume 2, AAAI’87*, AAAI Press, Menlo Park **1987**, pp. 677–682.

[25] M. Minsky, *Proc. IRE* **1961**, *49*, 8.

[26] R. Bellman, *Bull. Am. Math. Soc.* **1954**, *60*, 503.

[27] R. Sutton, A. Barto, *Psychol. Rev.* **1981**, *88*, 135.

[28] C. Berner, G. Brockman, B. Chan, V. Cheung, P. Debiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse, R. Józefowicz, S. Gray, C. Olsson, J. Pachocki, M. Petrov, H. P. de Oliveira Pinto, J. Raiman, T. Salimans, J. Schlatter, J. Schneider, S. Sidor, I. Sutskever, J. Tang, F. Wolski, S. Zhang, *CoRR* **2019**, abs/1912.06680.

[29] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. P. Lillicrap, K. Simonyan, D. Hassabis, *CoRR* **2017**, abs/1712.01815.

[30] J. Clark, D. Amodei, *Faulty Reward Functions in the Wild*, Vol. **28**, **2016**, pp. 1189–1212.