Explainable Robotic Plan Execution Monitoring under Partial Observability

by

Gokay Coruhlu

Submitted to the Graduate School of Sabanci University
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

Sabanci University

December 15, 2021

PhD DISSERTATION

Explainable Robotic Plan Execution Monitoring

under Partial Observability

APPROVED BY:

[redacted]	[redacted]
[redacted]	[redacted]
[redacted]	[redacted]
[redacted]	[redacted]
[redacted]	[redacted]
[redacted]	[redacted]

DATE OF APPROVAL: 15/12/2021

Explainable Robotic Plan Execution Monitoring under Partial Observability

Gokay Coruhlu

ME, Doctor of Philosophy, 2021

Dissertation Co-Advisors: Prof. Volkan Patoglu, Prof. Esra Erdem

Keywords: Plan execution monitoring, diagnostic reasoning, partial observability, guided replanning, hybrid planning, explanation generation, robustness of plans.

Abstract

Successful plan generation for autonomous systems is necessary but not sufficient to guarantee reaching a goal state by an execution of a plan. Various discrepancies between an expected state and the observed state may occur during the plan execution (e.g., due to failure of robot parts) and these discrepancies may lead to plan failures. For that reason, autonomous systems should be equipped with execution monitoring algorithms so that they can autonomously recover from such discrepancies.

We introduce a plan execution monitoring algorithm that operates under partial observability. This algorithm relies on novel formal methods for hybrid prediction, diagnosis and explanation generation, and planning. The prediction module generates an expected state after the execution of a part of the plan from an incomplete state, to check for discrepancies. The diagnostic reasoning module generates meaningful hypotheses to explain failures of robot parts. Unlike the existing diagnosis methods, the previous hypotheses can be revised, based on new partial observations, increasing the accuracy of explanations as further information becomes available. The replanning module considers these explanations while computing a new plan that would avoid such failures. All these reasoning modules are hybrid in that they combine high-level logical reasoning with low-level feasibility checks based on probabilistic methods. We experimentally show that these hybrid reasoning modules improve the performance of plan execution monitoring in service robotics applications with multiple bimanual mobile robots.

To evaluate the performance and to understand the applicability of the proposed execution monitoring algorithm, we introduce an execution simulation algorithm. This algorithm is based on a formal method that allows generation of

dynamic and relevant discrepancies, and simulation of all possible plan execution scenarios considering potential failures. This simulation algorithm can be used not only for testing execution monitoring algorithms subject to different conditions, but also to evaluate the robustness of plans. We illustrate these applications of our simulation algorithm in service robotics and cognitive factory settings with multiple mobile robots.

Kısmi Gözlemlenebilir Ortamlarda Açıklanabilir Robotik Plan Yürütme

Gökay Çoruhlu

Mekatronik Mühendisliği, Doktora, 2021

Doktora Tez Danışmanları: Prof. Volkan Patoğlu, Prof. Esra Erdem

Anahtar Kelimeler: Plan yürütme, hata tanısı, açıklama oluşturma, yönlendirilmiş yeniden planlama, hibrit planlama, uyuşmazlık oluşturma, plan yürütmesinin simülasyonu.

Özet

Otonom bir sistemin hedefine ulaşabilmesi için başarılı bir plan oluşturulması gereklidir ancak bu çoğu zaman yeterli değildir. Planın yürütülmesi sırasında beklenen ve gözlenen durumlar arasında çeşitli uyuşmazlıklar (örneğin, beklenmeyen harici olaylar, hedefte meydana gelen değişiklikler ya da robot parçalarının bozulmasından oluşabilecek farklılıklar) meydana gelebilir ve bu uyuşmazlıklar plan yürütmesinin başarısız olmasına sebep olabilir. Bu nedenle, otonom sistemler, bu tür uyuşmazlıklarla baş edebilmek için plan gözlemleme, hata tanısı ve yeniden planlama algoritmaları ile donatılmalıdır.

Bu doktora tezinde, kısmi gözlemlenebilir ortamlarda da uygulanabilen genel bir plan yürütme algoritması öneriyoruz. Bu algoritma, durum tahmini, hata tanısı, açıklama oluşturma ve yönlendirilmiş yeniden planlama gibi özgün hibrit modüllere dayanmaktadır. Tahmin modülü, uyuşmazlık kontrolü için, kısmi gözlemlenen bir durumda yürütülen bir planın sonunda beklenen durumu tahmin eder. Hata tanısı modülü uyuşmazlıkları robot parçalarının bozuklukları ile açıklayabilen anlamlı hipotezler oluşturur. Literatürdeki metotlardan farklı olarak, önerilen hata tanısı modülü, daha önce oluşturduğu hipotezleri yeni gerçekleştirilen gözlemler ışığında gözden geçirebilmekte ve bu sayede yeni bilgiler elde edildikçe daha doğru açıklamalar oluşturabilmektedir. Yeniden planlama modülü, oluşturulan açıklamaları göz önünde bulundurarak, var olan bozuk parçaların kullanımından kaçınan yeni planlar oluşturur. Tüm otomatik akıl yürütme modülleri hibrit yapıda olup yüksek-seviye mantıksal akıl yürütme ile düşük-seviye olasılık tabanlı fizibilite kontrollerinin birleşmesinden oluşmaktadır. Deneysel değerlendirmelerde hibrit akıl yürütme modüllerinin plan yürütmesinin başarımını arttırdığı, çoklu mobil manipülatörlerin yer aldığı servis robotiği uygulamalarında gösterilmiştir.

Ayrıca, önerilen plan yürütme algoritmasının başarımının ve uygulanabilirliğinin sistematik bir şekilde değerlendirilebilmesi için bir plan yürütmesi simülasyon algoritması öneriyoruz. Bu algoritma, dinamik olarak ilgili uyuşmazlıkların oluşturulmasına ve olası arızalar dikkate alınarak tüm plan yürütme senaryolarının simülasyonuna izin veren formal bir yönteme dayanmaktadır. Simülasyon algoritması, sadece farklı koşullara tabi plan yürütme algoritmalarını test etmek için değil, aynı zamanda farklı planların gürbüzlüğünü değerlendirmek için de kullanılabilir. Simülasyon algoritmamızın uygulamaları, servis robotiği ve bilişsel fabrika ortamlarında çoklu mobil robotlar kullanılarak gösterilmiştir.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Successful deployment of robotic assistants in real world environments necessitates these systems to be endowed with high-level cognitive functions that allow them to robustly deal with high complexity and wide variability of their surroundings. For that reason, autonomous robot systems are equipped with the ability to plan their actions to reach their goals. Independent of the underlying planning paradigm and/or algorithm, planning requires models of the environment and the robot to produce a valid plan. However, due to uncertainties and surprises, the world may evolve differently from the expected (as described by the model) during the execution of a plan. Therefore, robust execution of plans requires 1) detecting discrepancies between the expected and the observed states that affect validity of the rest of the plan (such discrepancies are called relevant), 2) diagnosing the cause of and generating explanations for these discrepancies, and 3) implementing a means of rational recovery. The process of verification of the continued validity of an executed plan, and implementation of a means of recovery, is known as *execution monitoring* [1,2].

Three main reasoning tasks take place during plan execution monitoring: *pre-*

diction, *diagnosis/explanation generation* and *replanning*. In particular, to be able to detect discrepancies, the expected state after the execution of the relevant part of the plan needs to be predicted, such that it can be compared with the current observed state. Once a discrepancy is detected, diagnostic reasoning is required to determine its cause, as it may have occurred early in the plan execution and may not have been easily detected by sensors. Finally, once the cause of the discrepancy is determined, a new plan needs to be generated by taking its explanation into account. Note that diagnostic reasoning not only to guide replanning but also to enable explanation generation.

Plan execution monitoring is commonly studied under the assumption of the full observability of the current state. However, privacy concerns, high cost of sensors, unreliability of sensors, or demanding environmental conditions (i.e., tight spaces, high temperatures, or corrosive environments) may render full observability infeasible for certain applications. In such cases, plan execution monitoring needs to be performed under *partial observability*, as only a portion of the current state can be monitored at any given time.

Plan execution monitoring under partial observability is challenging, as all three reasoning tasks of prediction, diagnosis and replanning need to be significantly extended to accommodate for unobservable states. In particular, for discrepancy detection under partial observability, it is not possible to compare the expected and the observed states, since full state observations are not available. Therefore, a discrepancy check is required to be performed based on the monitored fluents. Similarly, during diagnostic reasoning, one should generate explanations due to the discrepancies related to the monitored fluents. Finally, for replanning, the unobservable part of current state needs to be predicted, taking the diagnosis into account and ensuring consistency with the monitored fluents, such

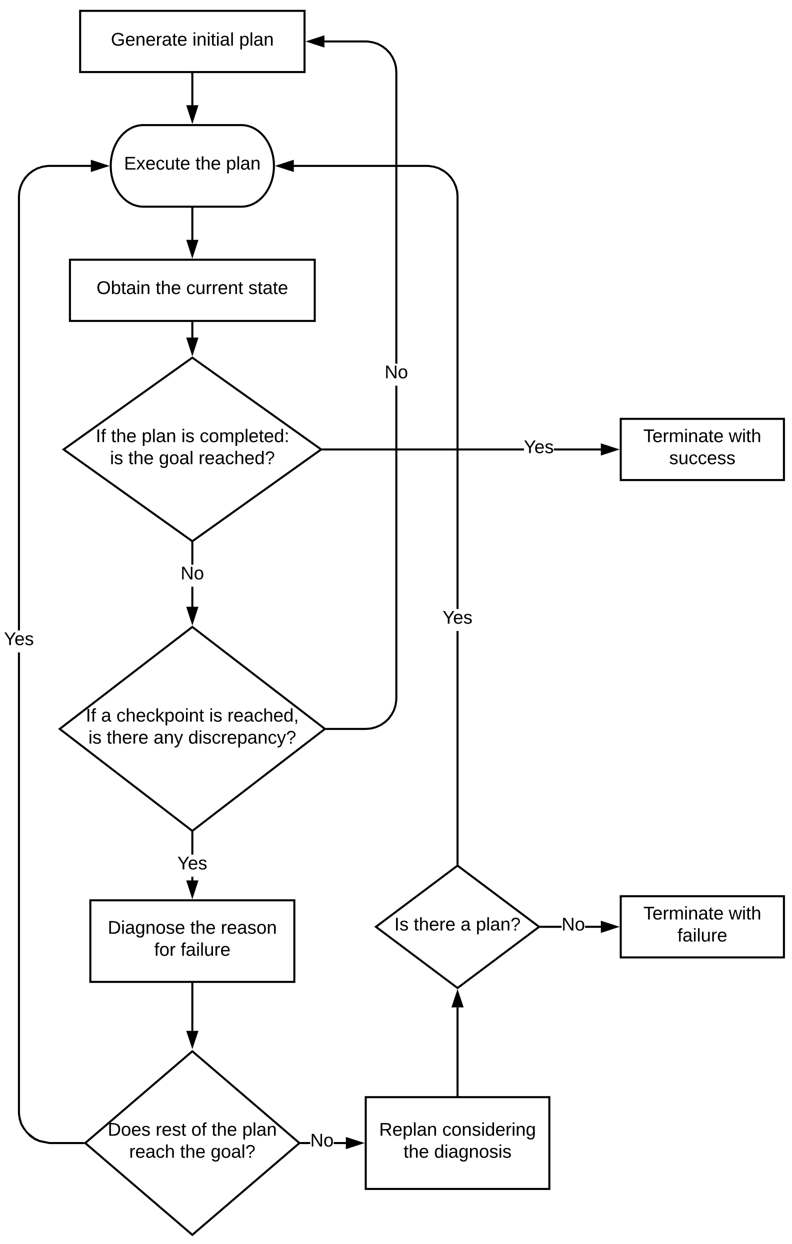

Generate initial plan
Execute the plan
Obtain the current state
If the plan is completed: is the goal reached?
Yes
Terminate with success
No
If a checkpoint is reached, is there any discrepancy?
No
Yes
Diagnose the reason for failure
Does rest of the plan reach the goal?
Yes
No
Replan considering the diagnosis
Is there a plan?
Yes
No
Terminate with failure

Figure 1.1.1: Plan Execution Loop

that a new plan can be computed.

In this dissertation, we present a formal method that extends model-based diagnostic reasoning to perform under partial observability and considers multiple observations, such that a possible hypothesis that explains observed discrepancies can be revised as new observations become available. These extensions not only enable the applicability of model-based plan execution monitoring to more realistic scenarios, but also improve the autonomy and robust plan execution of the robots, since it allows for the system to recover from incorrectly diagnosed failures without any external intervention. Figure 1.1.1 presents a flow chart of a planning and execution monitoring framework.

We formulate all the reasoning tasks required for plan execution using the expressive logic-based language of Answer Set Programming (ASP)—a knowledge representation and reasoning paradigm based on answer set semantics [3–5]—and solve them utilizing efficient ASP solvers. ASP has been used in various applications [6], including robotics [7]. We experimentally evaluate our methods over large sets of scenarios, to show the effectiveness of our novel diagnostic reasoning approach and usefulness of plan execution monitoring algorithm.

Before execution monitoring algorithms are deployed on autonomous systems, comprehensive testing is needed to evaluate their performances and to understand their applicability. With this motivation, we introduce a formal method for discrepancy generation with respect to the plan being executed to evaluate performance of execution monitoring algorithms through simulations. Discrepancies are introduced dynamically during the execution of a plan to simulate all possible plan execution scenarios with possible failures. This simulation algorithm also allows evaluation of the robustness of various plans. We present experimentally evaluation of the robustness of several plans under various execution monitoring algorithms.

1.1 Contributions

We present a novel plan execution monitoring algorithm that operates under partial observability. In particular, we introduce formal methods for the reasoning tasks of prediction, diagnosis and replanning required during execution monitoring. We also introduce a formal method for dynamic relevant discrepancy generation with respect to the plan being executed and we propose a novel simulation algorithm that enables systematic testing of execution monitoring algorithms.

The contributions of this dissertation can be summarized as follows.

(i) We present a formal method to *predict* states under partial observability. Our method integrates low-level feasibility checks (i.e., existence of collision-free trajectories) into high-level action descriptions, and thus can eliminate incorrect predictions of states reached by these actions. It also takes into account previous observations and diagnoses. We use hybrid prediction for three purposes: to detect discrepancies, to check their relevancies, and to infer a full current state for replanning. In the first case, hybrid prediction is utilized to compute the full expected state after executing some part of the plan from the initial state. In the second case, hybrid prediction is utilized to compute the full expected state after executing the rest of the plan from the partially observed current state. In this case, our method infers a full valid state from the current observations, that is closest to the expected state. In the third case, hybrid prediction is utilized to compute the full expected state after executing some part of the plan from the initial state, considering the current observations and current diagnosis.

(ii) We propose a formal method for *diagnostic reasoning* under partial observability to generate meaningful explanations for the relevant discrepancies.

Our method not only integrates low-level feasibility checks into high-level diagnostic reasoning, but also enables revisions of previous diagnoses such that more relevant explanations can be generated as further observations become available. We use hybrid diagnostic reasoning to identify the minimum number of robotic components that are most likely to cause the detected relevant discrepancy when they are broken. Our method also provides further explanations as to which actions of the plan fail and when, due to the inferred diagnosis.

(iii) We present a formal method for *replanning* under partial observability that generates new plans starting from the current state (with predicted unobservable part) and leading to the goal state, guided by the diagnosis with explanations. Our replanning method also integrates low-level feasibility checks into high-level task planning to ensure that the computed plans are executable in the real-world. Our replanning method also considers repairing a minimum number of broken parts, if needed.

(iv) Based on these ASP-based formal methods to detect discrepancies, generate diagnoses with explanations, and compute (re)plans, we introduce a novel *plan execution monitoring algorithm* that can be used under partial observability. Our algorithm can be used autonomously as well as interactively to involve the user in decisions and get their feedback (e.g., which diagnosis is more likely or which repairs are more preferable during replanning).

(v) We experimentally evaluate our plan execution monitoring algorithm over large sets of problem instances in a service robotics domain that involves multiple mobile manipulators, to investigate its scalability and effectiveness, with respect to different variations of diagnostic reasoning (i.e., with or without considering former observations and diagnoses) and replanning (i.e., with or without guidance by diagnosis).

(vi) We present a formal method for relevant discrepancy generation that might have occurred due to a failure of a robotic actuation action, after execution of some part of a plan. Our *discrepancy generation* is a generic method and applicable to robotic domains where the robotic actions may fail due to failures of robot components.

(vii) Based on this ASP-based formal method to generate discrepancies, we introduce a novel simulation algorithm that evaluates the performance of various execution monitoring algorithms by simulating all possible scenarios with relevant discrepancies. This algorithm is applicable to a variety of execution monitoring algorithms with/without diagnosis and/or replanning.

(viii) We introduce the notion of plan robustness and experimentally evaluate the robustness of different plans for a given planning problem in service robotics and cognitive factory settings with multiple mobile robots.

1.2 Outline

The rest of the dissertation is organized as follows:

Chapter 2 provides a comparative review of the related work on plan execution monitoring.

Chapter 3 overviews the representation of a transition system in ASP and presents the necessary tools to implement feasibility checks of robotic actions.

Chapter 4 introduces a formal definition of the planning problem in ASP, a formal definition of diagnosis problem using ASP, and presents novel formal methods to compute a most-probable minimum-cardinality diagnosis and to generate explanations for this diagnosis. Chapter 4 introduces a formal definition of prediction problem using ASP, and presents formal methods to detect a discrepancy and to determine its relevancy. Chapter 4 also introduces a formal definition of guided replanning, and presents formal methods to compute replans guided by the most recent diagnosis, and possibly by repairing a minimum set of robot components.

Chapter 5 overviews the proposed plan execution monitoring approach and presents the pseudo-code and details of the proposed algorithm. This section also presents a service robotics case study and provides evaluations of our methods over a comprehensive set of benchmark scenarios to show the effectiveness of our novel diagnostic reasoning approach and usefulness of plan execution monitoring algorithm.

Chapter 6 introduces novel methods to systematically evaluate various execution monitoring algorithms, presents a service robotics case study to demonstrate systematical generation of discrepancies and provides a case study for systematical evaluation of a cognitive factory.

Chapter 7 introduces a novel method to evaluate robustness of plans, and compare different execution monitoring algorithms in terms of their robustness.

Chapter 8 concludes the dissertation and discusses future research directions.

Chapter 2

Related Work

We are concerned about execution monitoring that not only detects failures/discrepancies but also finds out the reasons/diagnosis of these failures/discrepancies for a better-guided replanning. We are also concerned about evaluation of such execution monitoring algorithms. Therefore, let us examine the related work in three parts: studies about execution monitoring that utilize some sort of useful information for replanning, studies about diagnostic reasoning, and studies about testing of such algorithms.

Before the detailed discussions, let us clarify some terminology. We consider dynamic domains where the world states change over time due to direct or indirect effects of actions of the robots.

We understand discrepancies in the spirit of [8] as differences between an observed state and an expected state of the world (e.g., the table is empty while a glass was expected to be on it); discrepancies may lead to a failure of a plan. In that sense, our understanding of a discrepancy is different from "faults" [9], which are abnormal conditions or defects of physical components (e.g., deformation of a robot wheel); faults of a component may lead to the failure of the functionality of that component.

We consider a special type of faults, “broken components of robots”, as the cause of a discrepancy that lead to a plan failure. In that sense, our understanding of diagnostic reasoning is similar to fault diagnosis. Different from many various exiting approaches for fault diagnosis [9], we utilize model-based methods of automated reasoning.

As for plan failures, we consider failures that are caused by broken components. Other types of failures, such as failures due to human intervention [10], failures due to collisions with movable obstacles whose presence and location are not known in advance [11], and failures due to heavy objects that cannot be lifted alone [11], are investigated in complementary studies. Also, we do not consider action failures due to unreliable/incomplete execution of actions, which are usually handled at the low-level utilizing behaviour trees [12] or finite state machines via temporal logic [13, 14].

2.1 Execution Monitoring with More Informative Failures/Discrepancies

There are various related work (starting with SHAKEY [15]) on execution monitoring that utilize replanning upon detection of action failures; for a comprehensive summary of these approaches, we refer the reader to the thesis by Fritz [16] from the perspective of AI, and to [17] from the perspective of robotics and control.

Some execution monitoring approaches understand a failure as the failure of the current action’s execution in a plan, identify the reasons for the failure in terms of the unsupported preconditions and/or effects, and then utilize this knowledge while replanning [18,19]. For instance, Ambros-Ingerson and Steel [18] use a production system architecture where IF-THEN rules are used to map "flaws" (e.g., unsupported preconditions of a failed action) to "fixes" while replanning. Fichtner *et al.* [19] specify preconditions and effects of actions under the assumption that some properties are not "abnormal", and thus identify reasons of an action failure in terms of these abnormalities and utilize this information during replanning. Similarly, Winkler *et al.* [20] describe possible error cases and how to react to them, by high-level macros in goal definitions.

Some execution monitoring approaches consider discrepancies between the expected state and the observed state [8, 21, 22]. Discrepancies involve action failures, and more information; therefore, they can be useful in various ways depending on how they are defined and utilized. For instance, in addition to action failures, Lemai and Ingrand [21] consider temporal/resource conflicts (e.g., when an action takes longer time than planned), and perform plan repairs (e.g., postponing the remaining of the actions). De Giacomo *et al.* [8] understand a discrepancy as a failure of the remaining part of the plan, and perform replanning only when a discrepancy is relevant. In another work [23], where the goal is to compute a policy "in the now", the authors do not consider discrepancies but "reconsiderations" (e.g., when the precondition of an action is not achievable) to re-plan. In another study [24], the authors compute all the expected states during the execution of a plan in advance by a stochastic prediction function, and utilize them to detect discrepancies between the observed ones during plan execution; if a relevant discrepancy is detected then the prediction function is updated.

Some execution monitoring approaches further try to find explanations for discrepancies [22, 25, 26]. Zhang *et al.* [22] define failures (e.g., states where a cup is not observed to be in the main container), and utilize plan recovery methods using defaults (e.g., if the cup is not in the main container then by default it is in the auxiliary container). In another study [25], due to lack of support for defaults, the notion of "assumptive actions" is introduced to allow the system to generate assumptions about the initial state, that might cause the detected discrepancy between the observed state and the expected state. Eiter *et al*. [26] define a diagnosis of discrepancy as a "point of failure", where execution of a sequence of actions evolves differently than expected. This additional knowledge explains the detected discrepancy and allows the agent to reverse its plan until the point of failure and re-execute or repair the rest of the plan.

In our execution monitoring framework, similar to [8, 21, 22, 25, 26], we consider discrepancies between the expected state and the observed state, but with respect to a set of monitored fluents (under partial observability) so the observed state may be incomplete. Also, instead of continuous monitoring, we check for discrepancies from time to time; so the failure of a condition may not be observed at all until a discrepancy is detected.

Similar to [8], we check for the relevancy of a discrepancy for the execution of the rest of the plan, and perform replanning when the discrepancy is relevant. Different from the studies above, we identify possible causes of discrepancies by means of sets of broken components of robots and points of failures. We perform a novel method of diagnostic reasoning to find such possible causes under partial observability. Furthermore, we generate explanations based on such diagnoses about which actions could not be executed succesfully due to these diagnoses. The additional information due to the computed diagnoses and explanations is utilized to guide replanning.

2.2 Diagnosis of Discrepancies

Diagnoses of discrepancies are defined in various ways in the literature. For instance, according to Reiter and de Kleer [27, 28], a diagnosis of a discrepancy is described by a set of system components that are broken. As mentioned above, Eiter *et al.* [26] define a diagnosis of discrepancy as a point of failure. Roos and Witteveen [29] define a diagnosis of a discrepancy as a combination of some actions executed so far such that if these actions are qualified as abnormal, the observed plan state is compatible with predicted plan state. In this dissertation, we define diagnoses of discrepancies by means of sets of broken components of robots, and points of failures. Different from the studies [27, 28], we consider dynamic domains where the world states change over time, instead of a static system.

Concerning dynamic domains, diagnosis of failures or discrepancies by means of broken components is studied in the literature [30–32]. Let us examine these studies more in detail.

McIlraith *et al.* [30] consider hybrid systems modeled using automata and temporal casual graphs; their goal is to identify a most-probable diagnosis for a detected discrepancy. Different from this study, we do not model a robot as a hybrid system; instead, we model the robotic domain as a dynamic environment, where continuous variables regarding the robot are not represented explicitly but embedded in the high-level description of the domain. Our goal in execution monitoring is to find a most-probable diagnosis among the ones with minimum cardinality.

Balduccini and Gelfond [31], and Baral *et al.* [32] model dynamic domains using nonmonotonic formalisms, A-Prolog (based on answer set semantics [3, 4]) and Action Language $\mathscr{L}$ [33]. They introduce exogenous actions (e.g., a “break” action that causes parts to malfunction) in the domain descriptions to identify broken components that might cause discrepancies. Our approach is similar in that we use the nonmonotonic formalism of Answer Set Programming (ASP) [34–36], also called A-Prolog, to model a dynamic domain. Different from these studies, our approach does not need to introduce such a dummy action, since we can represent changes that do not involve a robotic action. Furthermore, unlike these approaches, our domain description is hybrid where the feasibility checks are embedded.

For replanning purposes, Balduccini and Gelfond [31], and Baral *et al.* [32] introduce exogenous "repair" actions in the domain descriptions to allow repairs of broken components. Different from these studies, our approach does not need to introduce such a dummy action, since we can allow repairs using weak constraints. To deal with replanning from a partially observable state, Baral *et al.* [32] consider sensing actions (i.e., to gather the missing knowledge about all the unobserved fluents via sensors) and conditional planning. In this dissertation, we do not consider sensing actions as the world is assumed to be partially observable even with the sensors, and thus present a method based on sequential planning. Though our replanning method can be extended to conditional planning (if the world can be observed via additional sensors), as discussed in our earlier studies [37–39].

In this dissertation, we are also concerned about generating explanations for relevant discrepancies, that not only provide information about which action in the plan being executed has failed and when, but also what might have caused such a failure in terms of broken parts. As the cause of a discrepancy may have occurred early in the plan execution and may not have been easily detected by sensors, we utilize diagnostic reasoning for explanation generation. In that sense, it is more general than the recent studies [40] in robotics that identify the cause of a robotic failure with respect to a taxonomy of failures, in terms of the current failed action, previous successful action, and current captured environmental context only. Note that we do not study explainability of plans [41,42] but explanation generation for relevant discrepancies in plan execution.

The study presented in this dissertation builds on and significantly extends the work in [43] in the following ways: (i) we assume that by default components are not broken initially, which allows the possibility of having components being broken at the very initial state, (ii) we enable the diagnostic reasoning method to revise its previous explanations whenever new observations become available, and (iii) most importantly, we significantly extend all the formal reasoning tasks in [43] to function under partial observability.

The most distinctive features of our method in comparison with the related work can be summarized as follows:

- works under partial observability and, furthermore, allows the monitored fluents to be partially observed,
- is integrated with low-level feasibility checks,
- generates meaningful hypotheses (i.e., the most-probable ones among the minimum cardinality diagnoses) about possibly broken robotic components,
- can generate explanations about which action has failed and when due to these broken parts,
- can determine which broken parts need to be repaired such that goal can be reached,
- can revise its previous hypotheses based on new observations, and
- does not require a dummy “break” action or a dummy “repair” action.

2.3 Testing of Execution Monitoring Algorithms

To develop fault-aware and reliable robot systems, simulation studies are essential for testing and benchmarking various execution monitoring approaches. However, to the best of our knowledge, there does not exist a generic simulation algorithm for execution monitoring where discrepancies are generated formally.

Livingstone [44] is a model based diagnosis engine which generates candidate diagnoses for the detected discrepancies, then suggest recovery actions to eliminate these discrepancies. Livingstone is used at NASA's demonstration spacecraft Deep Space 1 during various space missions. Lindsey and Pecheur [45] introduces Livingstone PathFinder (LPF) to test performance of Livingstone diagnosis engine. LPF accepts user-provided scenario scripts which consist of system commands and faults, then evaluates the performance of Livingstone in terms of successfully finding correct diagnoses by simulating the given scenarios.

Kurtoglu *et al*. [46] have introduced a fault catalog for Advanced Diagnostic and Prognostic Testbed (ADAPT) of NASA, which depends on a failure modes and effects analysis (FMEA). They have utilized user defined scenarios to test diagnostic systems which are generated by considering the results of FMEA analysis. During the execution, faults are injected to the testbed according to corresponding scenario and diagnostic system is tested in terms of the metrics they have defined.

In addition to using predefined scripts for evaluation, there are several other studies that utilize different methods to generate testing scenarios. For instance, in [8], the authors assume that the discrepancies can only be generated by exogenous actions and these exogenous actions are entered by the user after execution of a primitive action or a test evaluation. In [47], the user identifies "succeed, fail or yield no information" rates of each action or condition test. Another way of generating discrepancies in the literature is directly changing the world state by introducing/deleting information at any time during execution [48, 49].

We introduce a formal method for relevant discrepancy generation that might have occurred due to a failure of a robotic actuation action, after execution of some part of a plan. We also propose a novel algorithm for simulation of execution monitoring algorithms, that enables their systematic testing in simulation. In addition we introduce several metrics to evaluate performance of these algorithms.

Chapter 3

Preliminaries

Our contributions emerge from logic-based knowledge representation and automated reasoning, and thus utilize expressive logical formalisms and efficient automated reasoners to develop a general formal framework that not only can solve a variety of reasoning problems, including prediction, planning, guided re-planning, and diagnostic reasoning but also can orchestrate these reasoning modules by providing feedback in terms of constraints and preferences. For instance, thanks to the expressive formalism of ASP, we can utilize nonmonotonic constructs to express "defaults" (e.g., that every robotic part is assumed to be not broken by default, but it may get broken at any time). We can modularly express state/transition constraints, preferences, multi-objective optimizations, and ramifications. Here, elaboration tolerance [50] of the representation aids the synergy between different reasoning modules. Furthermore, we can integrate feasibility checks (and motion planning) into planning and other reasoning tasks by utilizing "external atoms" [51], which are more general than "semantic attachments" [52] used for classical planning, as the latter do not allow predicate extensions but only terms as arguments to attachments. This feature of external atoms is in particular useful for domains that include multiple movable obstacles.

Therefore, special languages for planning (e.g., PDDL) and the classical planners that support "semantic attachments" can be used for hybrid planning and guided re-planning in our framework but only in some cases, and necessitating substantial "surgeries" on operator definitions (e.g., ramifications and concurrency may require introduction of new operators [53], and adding a global constraint may require updating the preconditions and effects of every operator). However, even under these restrictions, the use of classical planners for diagnostic reasoning is not possible as our approach to diagnostic reasoning heavily relies on defaults.

3.1 Reasoning about Actions in ASP

Dynamic domains can be modeled as transition systems—directed graphs where nodes denote the world states of the domain and edges denote the transitions between these states caused by occurrences or nonoccurrences of actions in that domain.

We can represent the states and transitions denoted by a transition system in Answer Set Programming (ASP) [54] by means of formulas, called *rules*, of the form $\mathit{Head} \leftarrow \mathit{Body}$, where *Head* and *Body* are propositional formulas where each literal (i.e., a propositional atom q or its negation $\neg q$) is possibly preceded by default negation *not*. There are two sorts of negation in ASP [3]: $\neg$ is classical negation (as used in propositional logic) and *not* is default negation (as used in logic programming). The formula $\neg p$ intuitively expresses that we know that p does not hold; whereas *not* p expresses that we do not know that p holds.

If *Head* is $\bot$ then the rule is called a *constraint*. A finite set of rules is called a *program*. The models of a program are called *answer sets*, and they can be computed by special solvers, like CLINGO [55].

The idea is to solve planning, prediction and diagnosis problems over a transition system using the formal framework of ASP: Once the transition system modeling a dynamic domain is represented by a program, we combine this program with further constraints to solve a reasoning problem (e.g., planning). For instance, for a planning problem, the constraints describe the initial and goal states. Then the answer sets computed for the combined program characterize solutions (e.g., plans for the planning problems).

Before we formally define any reasoning problem in ASP, let us summarize how a transition system can be represented in ASP.

A planning domain description $\mathscr{D}^k$ of a dynamic domain as a transition system is formulated in ASP over a set $\mathscr{F}$ of fluent constants and a set $\mathscr{A}$ of action constants, with respect to a given upper bound k on time steps. This description relies on different forms of rules [7, Section 2]. Let us briefly go over some important types of rules relevant to our study.

For a formula H and an index i (for time step), let us denote by $H(i)$ the formula obtained from H by replacing every atom q by $q(i)$. Intuitively, $H(i)$ expresses that the formula H holds at time step $0 \leq i \leq k$.

<u>Effect rules:</u> Direct outcomes of actions are expressed with *effect rules* of the form

$$E(i+1) \leftarrow A(i), F(i) \tag{3.1}$$

where A is a conjunction of action atoms, E is a fluent literal, and F is a conjunction of fluent literals. This rule indicates that if the actions in A are executed at time step i where F holds then at the next state E holds. For instance, the following effect rule describes an effect of a “move” action of a mobile robot r navigating to a location l at time step i:

$$at(r,l,i+1) \leftarrow move(r,l,i).$$

It expresses that, as a direct effect of this action, the location of robot r changes to l at the next time step $i+1$.

Precondition rules: Preconditions of actions are expressed with *precondition rules* of the form

$$\leftarrow A(i), F(i), not\ G(i). \tag{3.2}$$

where A is a conjunction of action atoms, and F and G are conjunctions of fluent literals. The precondition rule above expresses that, to execute an action A at time step i at a state where F holds, the action's preconditions G must hold. For instance, according to the following precondition rule

$$\leftarrow move(r,l,i), at(r,l,i)$$

action $move(r,l)$ is possible if the robot is not already at the destination location l.

Hybrid rules: A *hybrid rule* is a rule where the right hand side of $\leftarrow$ includes external atoms. External atoms [51] are not fluent or action constants; their truth values are computed externally (out of ASP).

These rules are important for robotics applications since low-level feasibility checks for each action can be computed externally and then integrated into transition system description by means of external atoms. For instance, the following hybrid precondition rule ensures that, at time step i, a robot r can move from its current location x to its destination location l if there is a collision-free trajectory between them:

$$\leftarrow at(r,x,i), move(r,l,i), not\ \&moveIsFeasible[r,l,x](). \qquad (3.3)$$

The external atom $\&moveIsFeasible[r,l,x]()$ passes r, l, x as inputs to the external computation (e.g., a Python program) that calls a motion planner to check the existence of a collision-free trajectory for r from x to l, and then returns the result of the computation to the precondition rule.

Weak constraints: The ASP programs can be augmented with "weak constraints" [56]—expressions of the following form:

$$:\sim Body(t_1,...,t_n)[w@p,t_1,...,t_n].$$

Here, $Body(t_1,...,t_n)$ is a formula with the terms $t_1,...,t_n$. Intuitively, whenever an answer set for an ASP program satisfies $Body(t_1,...,t_n)$, the tuple $\langle t_1,...,t_n\rangle$ contributes a cost of w to the total cost function of priority p. The ASP solver tries to find an answer set with the minimum total cost. For instance, the following weak constraint

$$:\sim move(r,l,i), broken(r,base,i)[1@2]$$

instructs CLINGO to compute an answer set that does not include both $move(r,l,i)$ and $broken(r,base,i)$, if possible, with the assumption that robot *base* is broken, therefore cannot perform *move* action properly. However, if CLINGO cannot find such an answer set, it is allowed to compute an answer set with these atoms $move(r,l,i)$ and $broken(r,base,i)$ but with an additional cost of 1. Weak constraints are considered by CLINGO according to their priorities.

Nondeterministic choices: Transition systems rely on the possibility of *nondeterministic choices* about the occurrences of actions, and initial values of fluents. For instance, the following choice rule

$$\{move(r,l,i)\} \leftarrow$$

expresses that a robot r can choose to move to a location l at time step i. The possibility of this choice is subject to the action's preconditions. Similarly, the following choice rule expresses that initial locations l of objects o can be arbitrary at time 0:

$$\{at(o,l,0)\} \leftarrow .$$

State constraints: The nodes of a transition system characterize valid states described with *state constraints*. For instance, the following state constraint ensures the existence of a location for every object o at every time i:

$$\leftarrow \{at(o,l,i){:}loc(l)\}0$$

whereas the following state constraint ensures the uniqueness of a location for every object o:

$$\leftarrow 2\{at(o,l,i){:}loc(l)\}.$$

Here, the numbers 0 and 2 denote the upper and the lower bounds on the cardinality of subsets of $\{at(o,l,i){:}loc(l)\}$.

3.2 Feasibility Checks

We have utilized several feasibility checks for implementing the external atoms as in (3.3) to ensure the feasibility of robotic actions in dynamic domains. For each action these checks are implemented using OPENRAVE 0.9 [57] utilizing the following tools.

3.2.1 Convex Decomposition

Convex decomposition [58] converts an complex concave triangle mesh (i.e., robot) by approximating it with a collection of convex objects. This database is useful since using convex shapes ease dynamic collision checks by making them faster and more stable. An example of generated convex decomposition database for one of the robots used in simulations is given in Figure 3.2.1 where each link is approximated with a set of convex hulls.

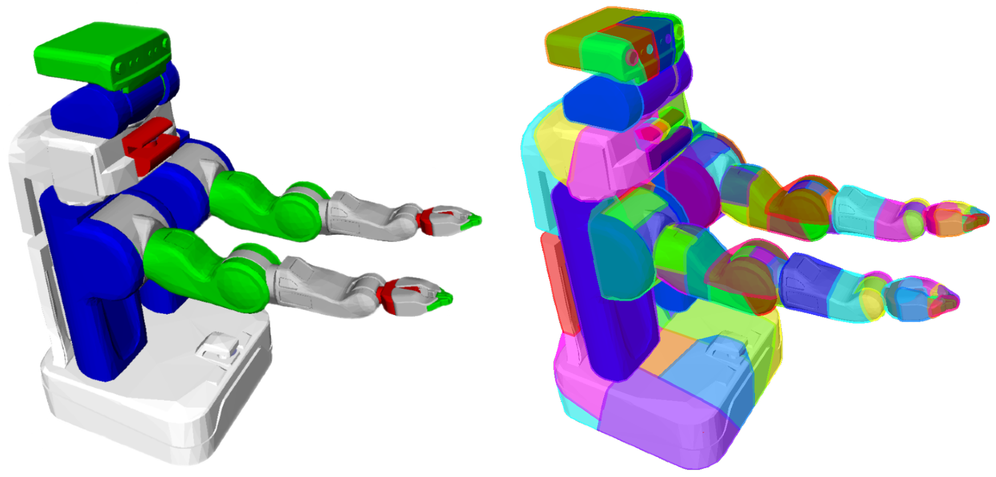


Figure 3.2.1: Convex decomposition of PR2 robot

3.2.2 Graspability

The grasp database [59] contains all possible force closure valid grasps for a specific gripper and a target object. Content of the database can be used during grasp planning procedure to determine kinematically-feasible collision-free trajectories. The grasp database is generated as follows:

- Bounding box of the target object surface is uniformly sampled to identify possible approach directions.
- Uniformly sampled points on the surface of the bounding box are used as starting points of rays that trace inwards and intersect with the surface of the object. The normal vectors of the objects surface at those intersection points are used as possible approach directions to grasp the object. An example is given in Figure 3.2.2 where possible approach directions are generated for a coffee pot.

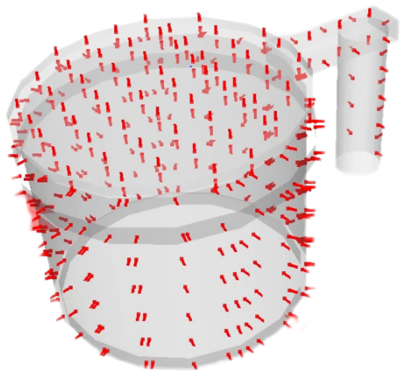


Figure 3.2.2: Possible approach directions for a coffee pot

- After initial pose of the gripper, preshape, and approach direction are chosen, the grasp planner module examines the contact points of the grasp and checks them for force closure. Figure 3.2.3 illustrates several candidate grasps.

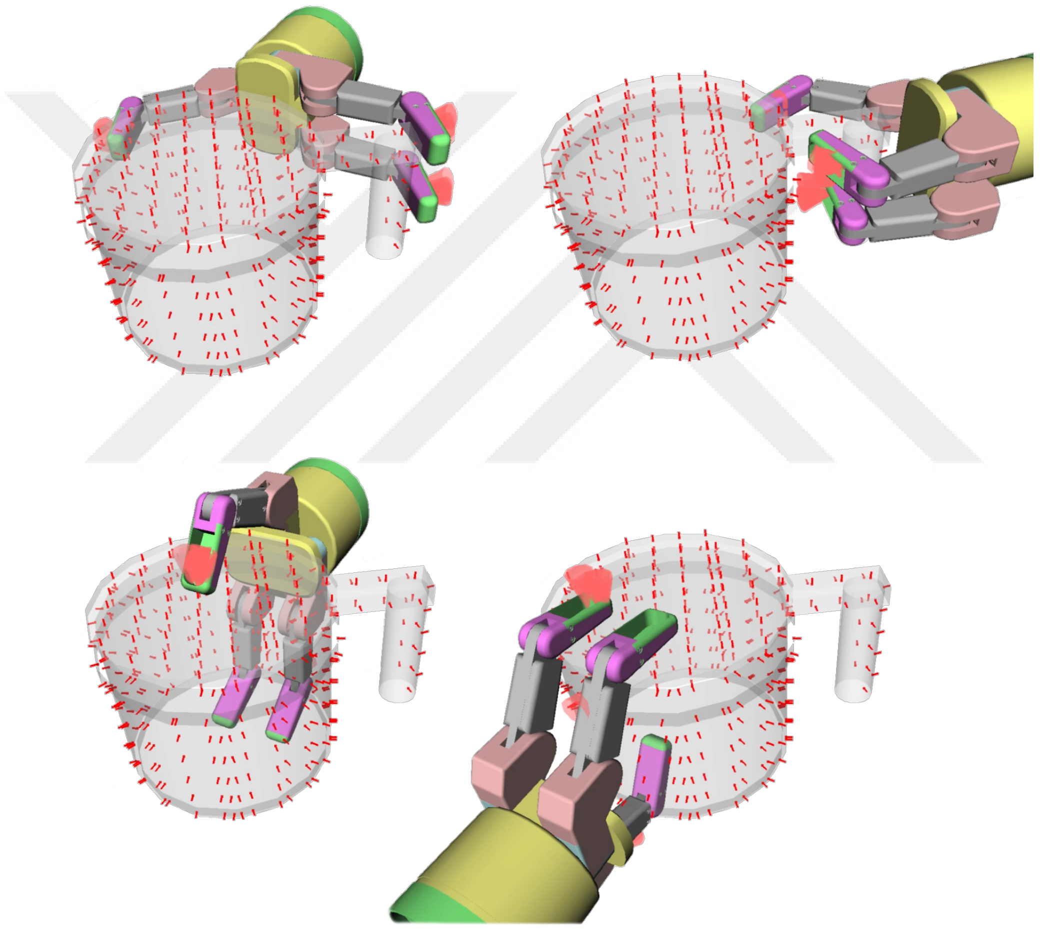


Figure 3.2.3: Examination of possible grasp

- Each successful grasp for the given gripper object pairs is recorded into the corresponding database. Figure 3.2.4 illustrates several successful grasps.

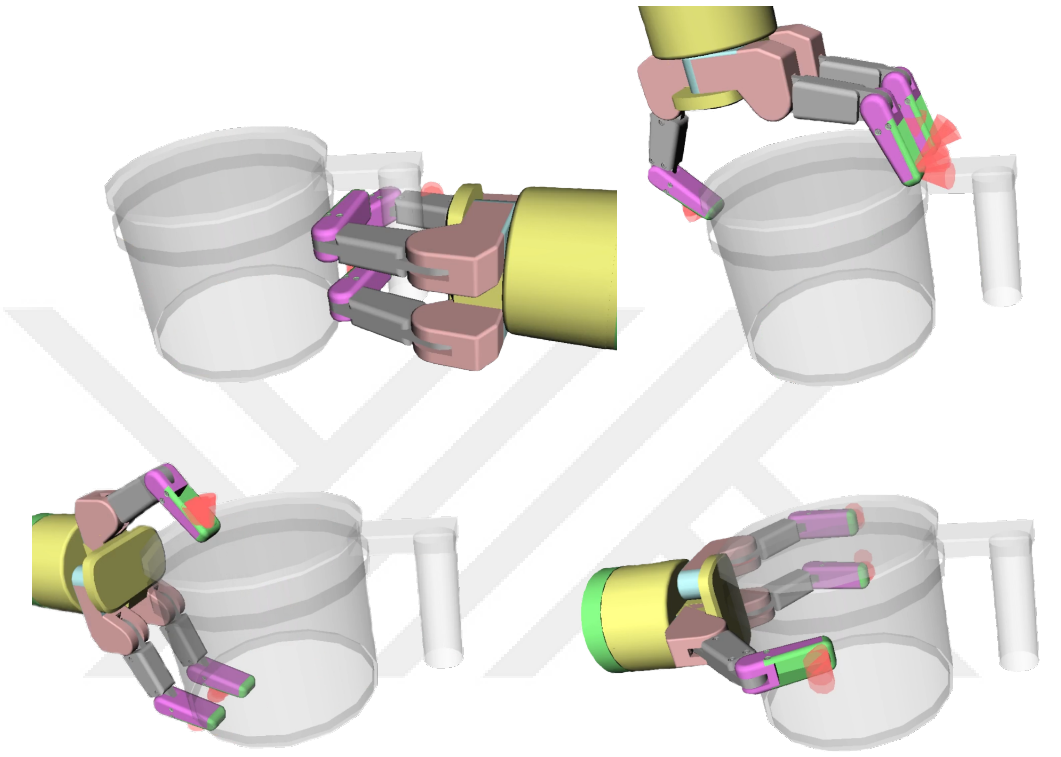


Figure 3.2.4: Some of the qualified graps

3.2.3 Inverse Kinematics

The Inverse kinematics [60] module generates analytical closed-form inverse-kinematic (IK) models for manipulators of robots. Generated models are used during motion planning where more than thousand of configurations are processed in each second. Therefore having an analytical solution rather than a numerical solution accelerates the motion planning procedure.

Furthermore, OPENRAVE's inverse kinematic module allow us to integrate following feasibility checks into inverse kinematic calculations:

- **Environment Collisions -** to check whether IK solution collides with the environment or not by utilizing Open Dynamics Engine (ODE).

- **Self Collisions -** to check whether robot collides with itself for the given IK solution or not by utilizing Open Dynamics Engine (ODE).

Figure 3.2.5 illustrates several feasible IK solutions for different end effector positions.

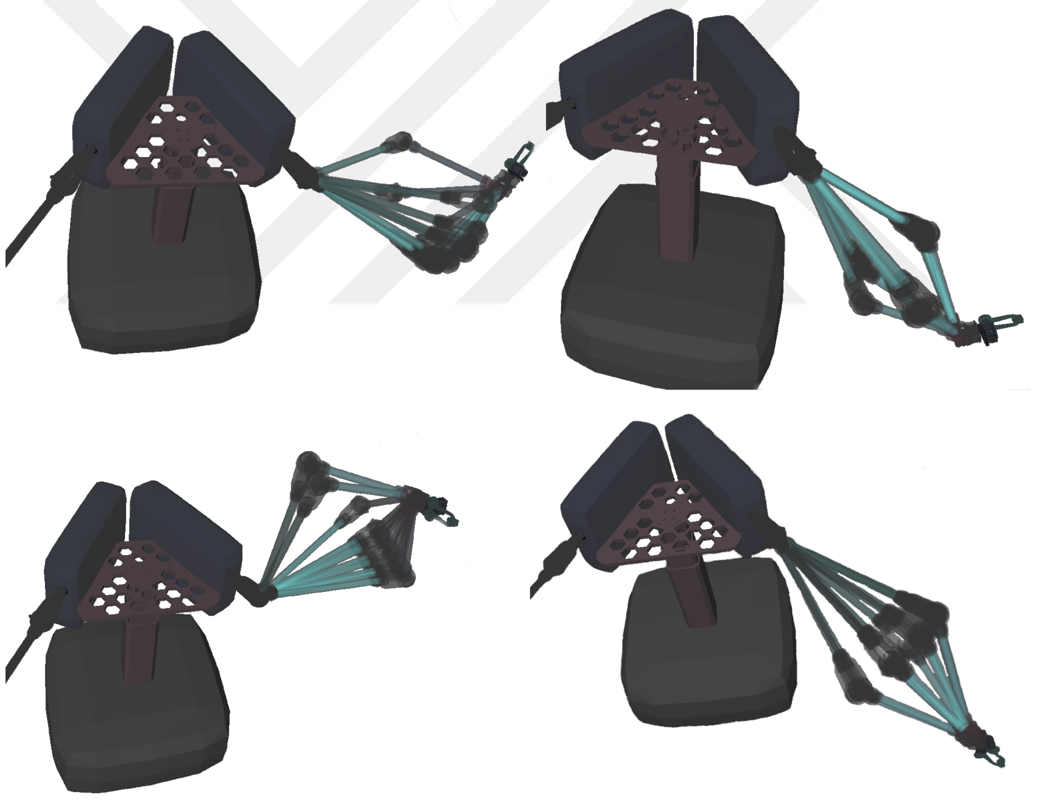


Figure 3.2.5: IK solution examples

3.2.4 Kinematic Reachability

The kinematic reachability database [61] contains IK solutions as quaternians and translation coordinates as well as number and density of these IK solutions for each point in the space. Basically, kinematic reachability database defines the workspace of a manipulator. It is generated by preprocessing the reachable volume and caching this information as a look-up data. This preprocessing stage is computationally intensive, but needs to be performed only once for each manipulator. The reachability database is useful since it can help robots to eliminate bad grasps quickly and position themselves wisely with the highest reachability (i.e., having multiple IK solutions) before performing any manipulation tasks.

Figure 3.2.6 illustrates generated kinematic reachability database for left arm of the robot CoCoA.

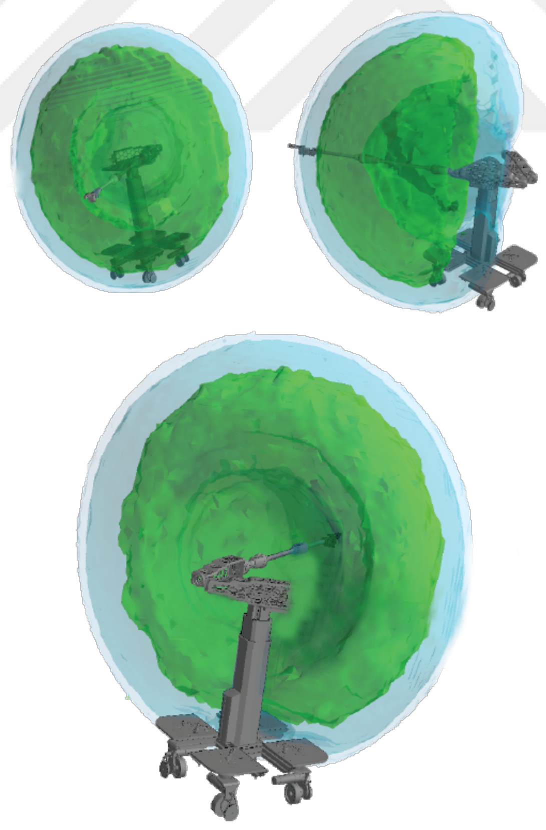


Figure 3.2.6: Kinematic reachability database for the robot CoCoA

3.2.5 Inverse Reachability

The inverse reachability database [62] is utilized to compute feasible configurations of the robot base such that the robot can perform a specific grasping action. The inverse reachability database uses the reachability space to sample for base locations and caches feasibility of such reaches for later use.

Figure 3.2.7 illustrates five different base locations for the left arm of CoCoA to reach the target object. The red square represents the target object.

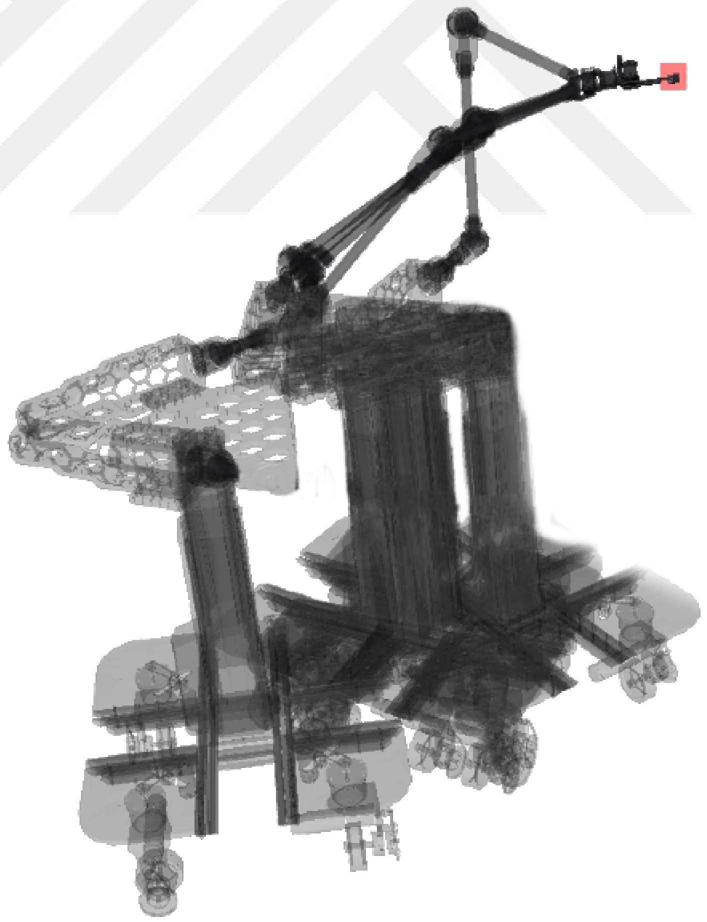


Figure 3.2.7: Inverse reachability database for robot CoCoA

Figure 3.2.8 illustrates relations and dependencies between OPENRAVE 0.9 databases/modules.

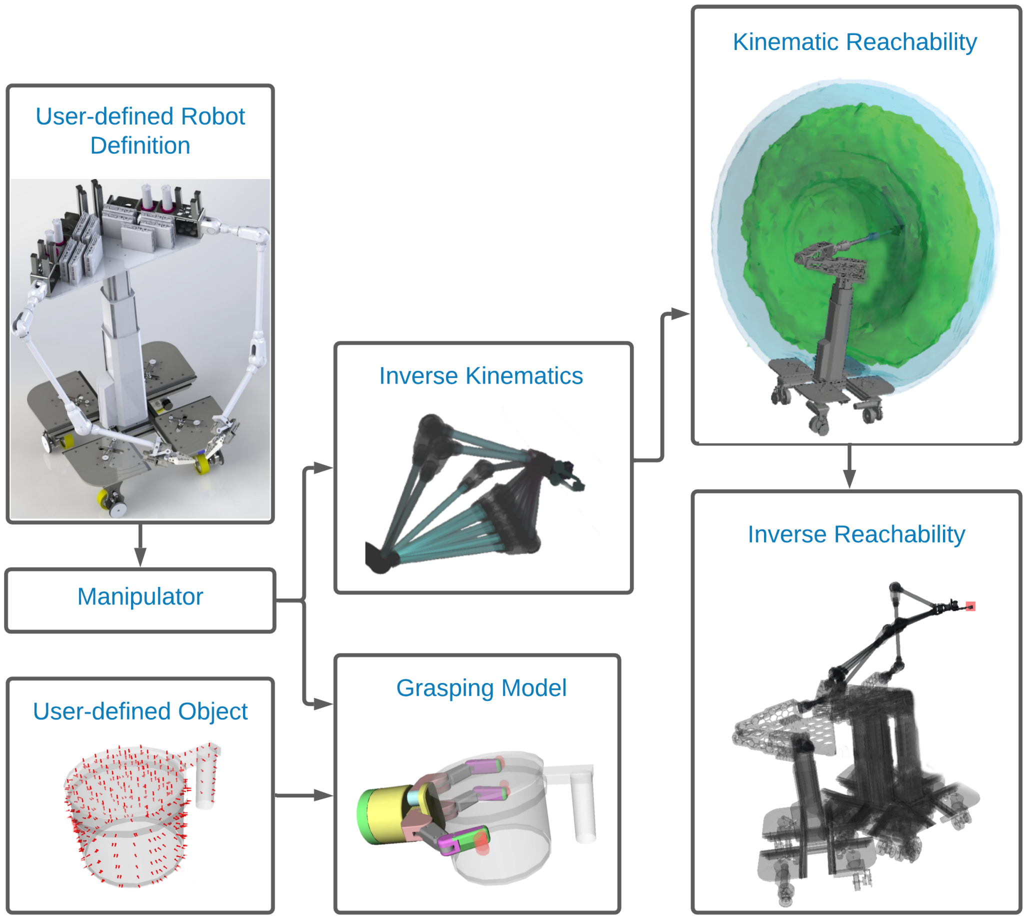


Figure 3.2.8: Database relations

3.2.6 Using Feasibility Tools to Check Feasibility of Actions

Combination of modules and databases defined in Section 3.2.1 can be used to implement feasibility checks for different robotic actions [63, 64]. Following examples illustrate usage of these tools to check feasibility of actions *move*, *pickUp* and *placeOn* respectively.

move action feasibility checks: The *move* action describes navigation of a robot from its current location to another location among obstacles and other robots. Therefore, it is essential to check whether there is a feasible (i.e., collision-free) path between these two locations. For this feasibility check, we use a rapidly exploring random tree (RRT) based motion planner, considering the presence of multiple robots in the environment.

pickUp action feasibility checks: For a robot to pick up an object with one of its manipulator, several feasibility checks are required (Algorithm 1). First, it is required to check whether there is a feasible grasping pose for the target object. For that, grasping database for specified target object and manipulator pair is generated (line 2). Second, it is required to find a base position that has an inverse kinematic solution for the selected grasp pose. For that, we utilize inverse reachability database to identify possible base positions for given grasp pose. This procedure continues until a possible base pose is found or all the possible grasp poses are exhausted (line 7). Third, we check if a collision-free trajectories exist for the manipulator to reach the object location, using the RRT-Connect motion planner of OPENRAVE and the collision checker of ODE (line 11).

Algorithm 1 PICKUP CHECK

```
Input: A target object to be picked up by a robot using manipulator manip.
Output: If action is feasible returns True; False otherwise.
 1: Feasible=False;
    //Generate a possible grasp pose set (grasping database).
 2: GraspSet ← GraspGenerator(Object);
 3: NumGrasps ← Length(GraspSet);
 4: i=0;
 5: while NumGrasps > i and Feasible == False do
 6:     TestGrasp ← GraspSet[i];
    // Generate a base pose for a grasp (inversereachability, convexdecomposition, in-
    versekinematic databases)
 7:     BasePose ← BasePoseGenerator(TestGrasp, Robot, Manip);
 8:     if BasePose not exist then
 9:         i=i+1;
10:     else
11:         Path ← PathGenerator(TestGrasp, BasePose);
12:         if Path not exist then
13:             i=i+1;
14:         else
15:             Feasible=True
16:
    return Feasible
```

placeOn action feasibility checks: For a robot to place an object that it holds with one of its manipulator to a target location requires some feasibility checks (Algorithm 2). First, the specified target area is uniformly discretized to obtain possible target object locations (line 2). After that, the algorithm picks a possible target object location, and tries to find a proper base position such that the object at hand can be placed on the target object location (line 7). If there exists such a base position, then an RRT-based motion planner is utilized to check whether there is a feasible path between the initial and final manipulator configurations by utilizing collision checker of ODE (line 11).

Algorithm 2 PLACEON CHECK

```
Input: A target location area to place the object TargetArea, end-effector pose
    GraspPose.
Output: If action is feasible returns True; False otherwise.
 1: Feasible=False;
    //Sample the target area
 2: TargetLocations ← Sampler(TargetArea);
 3: NumLocations ← Length(TargetLocations);
 4: i=0;
 5: while NumLocations > i and Feasible == False do
 6:     TargetLocation ← TargetLocations[i]
    //Find a proper base position (inversereachability, convexdecomposition, in-
    versekinematic databases)
 7:     BasePose ← BasePoseGenerator(TargetLocation, GraspPose);
 8:     if BasePose not exist then
 9:         i=i+1
10:     else
11:         Path ← PathGenerator(TargetLocation, GraspPose, BasePose)
12:         if Path not exist then
13:             i=i+1
14:         else
15:             Feasible = True
16:
    return Feasible
```

Chapter 4

Reasoning Problems for Execution Monitoring

A reliable execution monitoring algorithm requires detecting discrepancies between the expected and the observed states, diagnosing the cause of and generating explanations for these discrepancies, and implementing a means of rational recovery when the discrepancies affect the validity of the rest of the plan. In this chapter, we have formulated all the reasoning tasks required for plan execution using ASP in details.

For convenience, the notation used in this chapter and the rest of the dissertation is presented in Table 4.

Table 4.1.1: Notation used in the dissertation

Notation	**Description**
$\mathscr{D}^k$	planning domain description in ASP
$\mathscr{D}^t_{diag}$	diagnosis domain description in ASP
$\mathscr{D}_{state}$	ASP description of valid states of a robotic domain
$\mathscr{D}^{k-t}_R$	replanning domain description in ASP
$P_{<t}$	sequence of actions executed at the initial state s_0 until time step t
M	set of monitored fluents
O^M_t	set of earlier and current observed states o^M_i of monitored fluents in M at time step $i{\leq}t$
k	maximum length of a plan
$\mathscr{R}$	set of pairs of all robots and their components that may get broken
disables	relation $2^{\mathscr{R}} \times \mathscr{A} \times \mathscr{F}$ that describes which actions are affected when a set of robotic parts are broken
$X_{<t}$	set of triples (r,p,i) $(i<t-1)$ that describes the parts of p of robots r $((r,p)\in\mathscr{R})$ that are earlier diagnosed as broken
s_0	initial state
s_g	goal state
s_e	expected state
s_v	valid full state with respect to s_t and o^M_t

4.1 Defining Planning Problems

Intuitively, a planning problem in a dynamic domain, characterized by an initial state s_0, a goal state s_g, and a maximum length k of a plan, asks for a path P in the transition system modeling that domain such that the path starts at a node that denotes s_0, ends at a node that denotes s_g, and the length of the path is less than k. The following provides a formal definition of a plan when the transition system of a dynamic domain is formalized in ASP by a set $\mathscr{D}^k$ of rules as described in Chapter 3, over a set $\mathscr{F}$ of fluent constants and a set $\mathscr{A}$ of action constants, where the time steps range over $0..k$.

Let us characterize a state $s \subseteq \mathscr{F}$ by a conjunction F_s of fluent atoms that hold at that state and the negations of fluent atoms that do not hold at that state. For every action $A \subseteq \mathscr{A}$, let us denote by E_A the conjunction of action atoms in A and negations of action atoms in $\mathscr{A} - A$; E_A will be used to express that only the elementary actions in A are executed.

Definition 1. *A planning problem, $\mathscr{P}$ is characterized by a tuple $\langle \mathscr{D}^k, s_0, s_g, k \rangle$ where*

- *$\mathscr{D}^k$ is a planning domain description in ASP,*
- *s_0 is an initial state,*
- *s_g is a goal state,*
- *k is the maximum length of a plan.*

A solution of $\mathscr{P}$ (called a plan*) is a sequence $\langle A_0, \ldots, A_{n-1} \rangle$ $(n<k)$ of actions such that $\mathscr{D}^k$ when combined with the following rules (called a* planning query*)*

has an answer set:

$$\begin{aligned} &\leftarrow \textit{not } F_{s_0}(0) \\ &\leftarrow \textit{not } F_{s_g}(k) \\ &\leftarrow \textit{not } E_{A_i}(i) \quad (0 \leq i \leq n-1) \end{aligned} \tag{4.1}$$

The first and second constraints in (4.1) ensure that the plan starts at the given initial state and reaches the given goal state. The third constraint ensures that only the actions of the plan are executed.

Based on the definition above, plans of actions can be computed using the answer set solver CLINGO.

4.2 Diagnostic Reasoning and Explanations

During the plan execution, various discrepancies may occur between the expected state and the observed state and these discrepancies may result in plan failures. In the absence of knowledge of what went wrong, the execution of the newly computed plan is also likely to fail in some cases and the overall plan executed to reach the goal is expected to be longer. Diagnostic reasoning is required to identify and recover from such failures as early as possible.

In this dissertation, we assume that discrepancies may only occur due to malfunctioning of robot components. Then, a discrepancy can be diagnosed by a set of broken robot components. This assumption is reasonable in human-free environments where robots are highly involved.

To be able to perform diagnostic reasoning, the action domain description $\mathscr{D}^k$ used for planning has to be updated, since it formalizes states and transitions under the assumption that no anomalies occur.

4.2.1 Modifying the Action Domain Description into Diagnosis Domain Description

An action domain description $\mathscr{D}^k$ does not describe what will happen if a component of a robot is broken: if a robot part is broken then some precondition of an action in the plan may not hold anymore, and thus the plan simply fails without providing further knowledge. Therefore, for the purpose of diagnosing the reasons of such failures, we systematically transform the planning domain description $\mathscr{D}^k$ into a diagnosis domain description. Our method is based on the transformation suggested by [43] with some modifications.

Let $\mathscr{F}$ be a set of fluent constants, (that describe properties of the robotic domain that change over time), $\mathscr{A}$ be a set of action constants, (that describe robotic actuation actions), M be a set of fluent constants ($M \subseteq \mathscr{F}$) that are monitored.

Let $\mathscr{R}$ denote the set of pairs of all robots and their components that may get broken. For instance, $\mathscr{R} = \{(robot1, base), (robot2, arm)\}$ describes that the base of Robot 1 and the arm of the Robot 2 may be broken.

Let *disables* be a relation $2^{\mathscr{R}} \times \mathscr{A} \times \mathscr{F}$ that describes which actions are affected when a set of robotic parts are broken. For instance, $disables(X, a, F)$ represents that the effect of action a (of robot X) on F is disabled, if X are broken.

Suppose that a relevant discrepancy is detected at time step $t \leq k$, and we want to find a diagnosis for it. We transform $\mathscr{D}^k$ to a new description $\mathscr{D}^t_{diag}$ as follows, subject to the constraint that the time steps range over $0..t$.

Stage 1: A fluent constant of the form $broken(r, p)$ is introduced for each pair $(r, p) \in \mathscr{R}$, and the following rules are added to $\mathscr{D}^k$.

$$\begin{aligned}
&\neg broken(r,p,i) \leftarrow not\ broken(r,p,i)\\
&broken(r,p,i{+}1) \leftarrow not\ \neg broken(r,p,i{+}1), \neg broken(r,p,i)\\
&broken(r,p,i{+}1) \leftarrow broken(r,p,i)\\
&\{broken(r,p,0)\}.
\end{aligned}$$

These rules express that every robotic part (r,p) is assumed to be not broken by default, but may get broken at any time step i. If a robotic part gets broken, it remains broken. Moreover, different from [43], a robotic part may be broken initially.

Stage 2: A fluent constant $pre(A)$ is introduced for each action $A \subseteq \mathscr{A}$ to identify preconditions of A. By default, the value of $pre(A)$ is true for every action A; so the following rule is added to $\mathscr{D}^k$:

$$pre(A,i) \leftarrow not\ \neg pre(A,i).$$

Every precondition rule (3.2) of A in $\mathscr{D}^k$ is replaced by

$$\neg pre(A,i) \leftarrow F(i), not\ G(i).$$

Every effect rule of A (3.1) in $\mathscr{D}^k$ is replaced by

$$E(i{+}1) \leftarrow A(i), F(i),\ pre(A,i).$$

These rules express that, if the preconditions of an action A holds, then its effects are observed as expected.

Stage 3: Let A_R denote the robots that take place in the execution of action $A \in \mathscr{A}$ which may get broken (i.e., for every $r \in A_R$ there is a pair (r,p) in $\mathscr{R}$). The following rule is introduced to $\mathscr{D}^t_{diag}$ for each effect rule (3.1) in $\mathscr{D}^k$ such that $disables(A_R, A, F)$ holds:

$$E(i{+}1) \leftarrow A(i), F(i), \mathbf{,}_{r \in A_R, (r,p) \in \mathscr{R}} \neg broken(r,p,i).$$

These rules express that direct effect of actions are observed only if the relevant robotic parts are not broken.

4.2.2 Min-Cardinality Diagnosis

Once a relevant discrepancy is detected at time step t after executing the part $P_{<t}{=}\langle A_0, \ldots, A_{t-1}\rangle$ of the plan at the initial state s_0, diagnostic reasoning can be performed as follows:

Definition 2. *A diagnosis problem, DP, is characterized by a tuple* $\langle \mathscr{D}^t_{diag}, \mathscr{R}, s_0, P_{<t}, O^M_t, X_{<t}\rangle$ *where*

- $\mathscr{D}^t_{diag}$ *is the diagnosis domain description in ASP,*
- $\mathscr{R}$ *is the set of pairs of all robots and their components that may get broken,*
- s_0 *is an initial state,*
- $P_{<t}{=}\langle A_0, \ldots, A_{t-1}\rangle$ *is the sequence of actions executed at the initial state* s_0 *until time step t,*
- O^M_t *is a set of earlier and current observed states* o^M_i *of monitored fluents in M at time step* $i{\leq}t$*,*
- $X_{<t}$ *is a set of triples* (r,p,i) $(i < t-1)$ *that describes the parts p of robots r* $((r,p){\in}\mathscr{R})$ *that are earlier diagnosed as broken.*

A solution of DP is a set X_t of triples (r,p,j) that describe robotic components $(r,p) \in \mathscr{R}$ that are assumed to be broken at time step $j<t$ such that $\mathscr{D}^t_{diag}$ when combined with the following rules (called a diagnostic query*) has an answer set:*

$$\begin{aligned}
&\leftarrow not\ F_{s_0}(0) \\
&\leftarrow not\ E_{A_i}(i) \quad (0 \le i \le t-1) \\
&\leftarrow not\ F_{o_i^M}(i) \quad (o_i^M \in O_t^M) \\
&\leftarrow not\ broken(r,p,i) \quad ((r,p,i) \in (X_t \cup X_{<t})) \\
&\leftarrow not\ \neg broken(r,p,l) \\
&\qquad ((r,p) \in \mathscr{R}, l<t, (r,p,l) \notin (X_t \cup X_{<t})).
\end{aligned} \tag{4.2}$$

If DP has a solution X_t, then we say that $X_t \cup X_{<t}$ is a diagnosis *for the relevant discrepancy detected at time step t after executing $P_{<t}$.*

The diagnostic query (4.2) checks if every observed state in O_t^M can be reached from the initial state s_0 by executing the plan $P_{<t}$ if the parts of the robots in X_t and $X_{<t}$ were broken at specified times. If the program $\mathscr{D}^t_{diag}$ combined with rules (4.2) has an answer set, then $X_t \cup X_{<t}$ is a potential cause for the detected discrepancy.

Definition 2 can be used to implement three different approaches to diagnostic reasoning.

- *Augmented Diagnosis*: A diagnosis is generated about the broken parts based on the current observation, while imposing the validity of all of the previous hypotheses. This approach can be implemented by the following settings: $X_{<t}$ is set to the previously diagnosed broken parts before time step t and O_t^M contains only the last observation o_t^M.

- *Reset Diagnosis*: A diagnosis is generated about the broken parts based on the current observation, without considering any of the previous hypotheses. This approach can be implemented by the following settings: $X_{<t} = \emptyset$ and O_t^M contains only the last observation o_t^M (i.e., $O_t^M = \{o_t^M\}$).

- *Revised Diagnosis*: A diagnosis is generated about the broken parts based on the current observation and the previous observations, while updating the previous hypothesis as necessary. This approach can be implemented by the following settings: $X_{<t} = \emptyset$ and O_t^M contains the last observation o_t^M as well as the previous observations o_i^M $(i < t)$.

In general, more than one diagnosis can be generated for a discrepancy. In practice, focusing on diagnoses that include the minimum number of broken parts may be more reasonable, since malfunctioning of multiple robot parts at the same time is unlikely. We can find a min-cardinality diagnosis by adding the following weak constraints to the diagnostic query (4.2).

$$:\sim broken(r,p,i) \quad (i{<}t, (r,p) \in R).[1@2, r, p] \tag{4.3}$$

The rules (4.2) $\cup$ (4.3) define a *min-cardinality diagnostic query*.

4.2.3 Most-Probable Min-Cardinality Diagnosis

For a plan utilizing several robots, where each of them has several components, there may be more than one (min-cardinality) diagnosis for a detected relevant discrepancy. In such cases, our execution monitoring algorithm can utilize a priori knowledge about the likelihood of each robotic component getting broken to identify the most-probable diagnosis out of all (min-cardinality) diagnoses.

We express such a priori knowledge by atoms of the form $prob_b(r,p,w)$ where w is a positive integer that describes the likelihood of a component p of a robot

r getting broken. Then, we add the following weak constraints to the min-card diagnostic query (4.2) $\cup$ (4.3) to find a most-probable min-cardinality diagnosis:

$$:\sim broken(r,p,i), prob_b(r,p,w) \;\; (i{<}t, (r,p) \in R).\; [-w@1, r, p]$$

Note that the priority of the weak constraint above is less than that of (4.3) to ensure that a most-probable diagnosis is found among the min-cardinality diagnoses.

4.2.4 Explanation Generation

While diagnoses explain the reasons of relevant discrepancies in terms of broken robotic components, further explanations can be generated to include the actions whose effects have not been observed as expected due to these diagnoses. For that purpose, we can extend the diagnostic reasoning problem as follows.

For every action constant $A \in \mathscr{A}$, let $A_R \subseteq \mathscr{R}$ be the set of all pairs (r,p) of robots r and their parts p such that r takes part in executing the action A and that $disables(A_R, A, F)$ holds for some fluent constant $F \in \mathscr{F}$.

Definition 3. *Let $DP = \langle \mathscr{D}^t_{diag}, \mathscr{R}, s_0, P_{<t}, O^M_t, X_{<t} \rangle$ be a diagnosis problem. A* diagnosis with an explanation *for DP is a set X^E of tuples (r,p,A,i) where $(r,p) \in A_R$ and $i{<}t$, such that $\mathscr{D}^t_{diag}$ when combined with the diagnostic query* (4.2) *where $X_t \cup X_{<t} = \{(r,p,i) | (r,p,A,i) \in X^E\}$, and the following rules has an answer set:*

$$\begin{aligned} explains(r,p,A,i) &\leftarrow A(i), broken(r,p,i) \\ &((r,p,A,i) \in X^E, i{<}t). \end{aligned} \tag{4.4}$$

The atoms of the form $explains(r,p,A,i)$ that appear in an answer set for $\mathscr{D}^t_{diag} \cup (4.2) \cup (4.4)$ express which actions have failed because of which broken components.

4.3 Discrepancies and Relevancy

Diagnostic reasoning is applied if a discrepancy is detected between the expected and the observed states and found relevant for the rest of the plan. Let us define these concepts more formally.

4.3.1 Predicting the Expected State

A discrepancy (if there exists one) can be detected by comparing the expected state and the observed state after the execution of some part of a plan. We can predict an expected state by simulation of the part of the plan considering "believed-to-be-broken" robot components, while an observed state can be obtained by monitoring fluents with the help of sensors during the execution of the plan.

Definition 4. *A prediction problem, PP, is characterized by a tuple* $\langle \mathscr{D}^t_{diag}, s_0, P_{<t}, O^M_t, X_{<t} \rangle$ *where*

- $\mathscr{D}^t_{diag}$ *is the diagnosis domain description in ASP,*
- s_0 *is an initial state,*
- $P_{<t} = \langle A_0, \ldots, A_{t-1} \rangle$ *is the sequence of actions executed at the initial state* s_0 *until time step* t*,*
- O^M_t *is a set of earlier and current observations* o^M_i *of monitored fluents in* M *at time step* $i \leq t$*,*
- $X_{<t}$ *is a set of triples* (r, p, i) $(i < t-1)$ *that describes the parts* p *of robots* r $((r, p) \in \mathscr{R})$ *that are earlier diagnosed as broken.*

A solution of PP (called an expected state*) is a state s_t, such that $\mathscr{D}^t_{diag}$ when combined with the following rules has an answer set:*

$$
\begin{aligned}
&\leftarrow not\ F_{s_0}(0)\\
&\leftarrow not\ F_{s_t}(t)\\
&\leftarrow not\ E_{A_i}(i) \quad (0 \le i \le t-1)\\
&\leftarrow not\ F_{o^M_i}(i) \quad (o^M_i \in O^M_t)\\
&\leftarrow not\ broken(r,p,i) \quad ((r,p,i) \in X_{<t})\\
&\leftarrow not\ \neg broken(r,p,i)\\
&\qquad ((r,p) \in \mathscr{R}, i < t-1, (r,p,i) \notin X_{<t})
\end{aligned}
\tag{4.5}
$$

The first and second constraints of (4.5) ensure that the plan starts at the given initial state and reaches the expected state. The third constraint ensures that only the actions of the plan are executed. The fourth constraint takes into account the earlier and current observations. The fifth and sixth constraints indicate that only the (r,p) pairs in $X_{<t}$ are assumed to be broken and the rest of the (r,p) pairs are functioning properly.

Since the sequence $P_{<t} = \langle A_0,\dots,A_{t-1}\rangle$ of actions executed at the initial state s_0 until time step t is given, it is known that actions in $P_{<t}$ are already executed. Note that the planning domain description $\mathscr{D}^k$ is not suitable for prediction since there are precondition rules that are implemented as constraints. However, the diagnosis domain description, $\mathscr{D}^t_{diag}$, allows actions to be executed even when the effects of actions are not observed. Therefore, diagnostic domain description $\mathscr{D}^t_{diag}$ is used for state prediction.

4.3.2 Discrepancies

A discrepancy (if there exists one) is detected by comparing the expected state with the current observations.

Let $\mathscr{D}_{state}$ be the union of the state constraints and the nondeterministic choice rules for initial values of fluents in $\mathscr{D}^0$ for time step 0. The answer sets for $\mathscr{D}_{state}$ characterize the states of the transition diagram defined by $\mathscr{D}^k$.

Definition 5. *Let s_e be the expected state, and o^M be a set of observations of monitored fluents in M. There is a* discrepancy *between s_e and o^M if the program $\mathscr{D}_{state}$ when combined with the following rules does not have an answer set:*

$$\begin{gathered} \leftarrow not\ F_{s_e}(0) \\ \leftarrow not\ F_{o^M}(0). \end{gathered}$$

4.3.3 Inferring the Current State from Partial Observations

To check the relevancy of a discrepancy for the rest of the plan execution, and then to be able to replan if needed, we need to infer a valid full state from the partially observed current state, while respecting the state constraints of the robotic domain. Meanwhile, among all possible current states, we prefer the ones that are closest to the expected state.

For every state s, let us denote by $l(s)$ the set of fluent literals that hold at state s.

Definition 6. *A valid state generation problem, VSG, is characterized by a tuple $\langle \mathscr{D}_{state}, s_e, o^M \rangle$ where*

- *$\mathscr{D}_{state}$ is an ASP description of valid states of a robotic domain,*
- *s_e is an expected state,*

- o^M *is an observed state of monitored fluents in M.*

A solution of VSG is a valid full state s_v *with respect to* s_e *and* o^M*, such that* $\mathscr{D}_{state}$ *when combined with the following rules has an answer set:*

$$\begin{aligned}
&\leftarrow not\ F_{s_v}(0) \\
&\leftarrow not\ F_{o^M}(0) \\
&:\sim l_e(0).\quad [-1@1] \quad (l_e \in l(s_e)).
\end{aligned} \tag{4.6}$$

The first two constraints of (4.6) ensure that the state s_v is an inferred valid state where the current observations o^M hold. The weak constraints maximize the number of common fluent literals between the expected state s_e and the inferred valid state s_v, by minimizing their differences.

4.3.4 Determining the Relevancy of a Discrepancy

Sometimes observed discrepancies may be irrelevant to the rest of the plan, i.e., they may not prevent the rest of the plan from reaching a goal state. Therefore, when a discrepancy is detected, the monitoring agent takes the relevancy of a discrepancy into consideration before performing any diagnosis or replanning tasks.

Definition 7. *A discrepancy relevancy check problem, RC, is a decision problem characterized by a tuple* $\langle \mathscr{D}^t_{diag}, s_t, o^M_t, s_v, P_{\geq t}, s_g, X_{<t} \rangle$ *where*

- $\mathscr{D}^t_{diag}$ *is the diagnosis domain description in ASP,*
- s_t *is the expected current state at time step t (i.e., a solution for the prediction problem* $\langle \mathscr{D}^t_{diag}, s_0, P_{<t}, O^M_t, X_{<t} \rangle$*),*
- o^M_t *is a set of current observations of monitored fluents in M at time step t such that there is a discrepancy between* s_t *and* o^M_t*,*

- *s_v is a valid full state with respect to s_t and o_t^M,*
- *s_g is a goal state,*
- *$P_{\geq t} = \langle A_t, \ldots, A_{n-1} \rangle$ is the rest of the plan to be executed,*
- *$X_{<t}$ is a set of triples (r, p, i) $(i < t-1)$ that describes the parts p of robots r $((r, p) \in \mathscr{R})$ that are earlier diagnosed as broken.*

The discrepancy relevancy check problem RC returns True (i.e., the discrepancy between s_t and o_t^M is relevant for the execution of $P_{\geq t}$ at s_v, subject to $X_{<t}$) if the goal state s_g is not the expected state for the prediction problem $\langle \mathscr{D}_{diag}^{n-t}, s_v, P_{\geq t}, \{\}, \{(r, p, 0) | (r, p, i) \in X_{<t}\} \rangle$.

Otherwise, RC returns False.

4.4 Guided Replanning and Repairs

After a relevant discrepancy is detected (Definition 7) at time step $t \leq n$ and a possible diagnosis X is generated for it (Definition 2), we can perform replanning to reach the goal respecting the upper bound k on the total plan length. Replanning can be guided by the generated diagnosis leading to more meaningful plans by ensuring that robots do not perform any actions using parts that are diagnosed as broken.

4.4.1 Guided Replanning

To guide replanning by diagnostic reasoning, we obtain a replanning domain description from the planning domain description as follows.

For every action constant $A \in \mathscr{A}$, let $A_R \subseteq \mathscr{R}$ be the set of all pairs (r,p) of robots r and their parts p such that r takes part in executing the action A and that $disables(A_R, A, F)$ holds for some fluent constant $F \in \mathscr{F}$. The replanning domain description $\mathscr{D}_R^{k-t}$ is obtained from $\mathscr{D}^{k-t}$ by adding the following constraints for every time step $i \leq k-t-1$:

$$\leftarrow A(i), broken(r,p,i) \quad ((r,p) \in A_R). \tag{4.7}$$

Definition 8. *A replanning problem, RP, is characterized by a tuple* $\langle \mathscr{D}_R^{k-t}, s_t, s_g, X_t, k-t \rangle$ *where*

- $\mathscr{D}_R^{k-t}$ *is the replanning domain description in ASP,*
- X_t *is a diagnosis for the current detected relevant discrepancy,*
- s_t *is the current expected state (i.e., a solution for the prediction problem* $\langle \mathscr{D}_{diag}^t, s_0, P_{<t}, O_t^M, X_t \rangle$*),*
- s_g *is a goal state,*
- $k-t$ *is the maximum length of a plan to reach* s_g *from* s_t*.*

A solution of RP is a sequence $\langle A_0, \ldots, A_{m-1} \rangle$ *of actions* ($m \leq k-t-1$) *(called a* replan*) such that* $\mathscr{D}_R^{k-t}$ *when combined with the following rules has an answer set:*

$$\begin{aligned}
&broken(r,p,0) \leftarrow \quad ((r,p,i) \in X_t) \\
&\leftarrow not\ F_{s_t}(0) \\
&\leftarrow not\ F_{s_g}(m) \\
&\leftarrow not\ E_{A_i}(i) \quad (i \leq m-1).
\end{aligned} \tag{4.8}$$

4.4.2 Repairs

If there does not exist a solution to the replanning problem *RP*, then we can consider repairing some of the broken components of the robots before we replan. To find a solution to the replanning problem *RP* while repairing a set Y of minimum number of broken parts, we replace the constraints (4.7) in $\mathscr{D}_R^{k-t}$ with the following weak constraints

$$:\sim A(i), broken(r,p,i).[1@3,r,p] \quad ((r,p) \in \mathscr{R})$$

Let us denote the updated description by $\mathscr{D}_{Rfix}^{k-t}$.

Definition 9. *Let* $RP{=}\langle \mathscr{D}_R^{k-t}, s_{e_t}, s_g, X_t, k-t)\rangle$ *be a replanning problem. A* replan with repairs *consists of a sequence* $P{=}\langle A_0,\ldots,A_{m-1}\rangle$ *of actions* ($m{\le}k-t-1$) *such that* $\mathscr{D}_{Rfix}^{k-t}$ *when combined with* (4.8) *has an answer set Z, and a set* $Y \subset \mathscr{R}$ *of parts to be repaired such that* $Y = \{(r,p) \in \mathscr{R} : A(i) \in Z, broken(r,p,i) \in Z, 0{\le}i{\le}m-1\}$.

Some broken parts can be repaired more easily or on time, considering the available resources (e.g., consider a planetary rover). Such a priori knowledge can be useful to identify which parts are more preferable to repair while replanning. Suppose that atoms of the form $pref_x(r,p,w)$ where w is a positive integer, express the user's preference of repairing a component p of a robot r considering the available resources. Then, the description $\mathscr{D}_{Rfix}^{k-t}$ can be obtained from $\mathscr{D}_R^{k-t}$ by replacing the constraints (4.7) with the following weak constraints

$$:\sim A(i), broken(r,p,i), pref_x(r,p,w).[-w@3,r,p] \quad ((r,p) \in \mathscr{R})$$

Replanning with this weak constraint will involve repairing a most-preferable min-cardinality set of broken components.

Chapter 5

Plan Execution Monitoring with Diagnostic Reasoning

Our plan execution monitoring framework is based on formally representing a dynamic robotic domain, by means of logical formulas using the nonmonotonic formalism of Answer Set Programming (ASP) [54] embedded with external feasibility checks. Utilizing the reasoning mechanisms of ASP, it is capable of a rich set of hybrid reasoning tasks. Let us briefly give the overall idea of our framework by underlining these capabilities.

Given an initial state and goal conditions, the planning agent starts with *computing a hybrid plan of actions* taking into account the logical descriptions as well as the feasibility checks of actions. Robots start executing the actions in this plan, and occasionally perform (partial) observations about the environment from time to time.

Whenever a new observation is received, the monitoring agent verifies that the plan has evolved as expected, by first *predicting the expected state* and then *checking the consistency of the partially observed current state against the ex-*

pected state.

If a discrepancy is detected, then the monitoring agent checks the relevancy of this discrepancy to the rest of the plan, by first *inferring a valid full state description for the partially observed current state*, and then *checking the validity of the rest of the plan from the current observed state*. If no discrepancy is detected or the detected discrepancy is not relevant, the execution of the plan continues.

If the detected discrepancy is relevant, then the diagnosis agent identifies possible causes of this discrepancy by *diagnosing broken parts of the robots*, and *generating further explanations for the discrepancy in terms of failed actions*. For an observed discrepancy, there can be more than one diagnoses or explanations obtained by such *hypothetical reasoning*, so the diagnosis agent finds a min-cardinality diagnosis with more likelihood of failure and an explanation for it.

Once such a diagnosis and explanation are computed, the replanning agent *finds a new hybrid plan from the current state (inferred from the partially observed state) to the goal, guided by the constraints (obtained from inferred diagnoses)*. If there exists no such plan, the replanning agent *infers a repair by identifying the minimum number of broken parts that can be repaired such that the goal can be achieved by guided replanning*.

Our plan execution monitoring framework is autonomous, yet it can be used in an interactive mode as well. When the diagnosing agent computes a set of most-probable min-cardinality diagnoses with explanations, the monitoring agent may consult an expert to pick one of them.

All of the reasoning tasks are performed over a hybrid formulation of the robotic domain, that combines high-level logical reasoning with low-level feasibility checks, possibly based on probabilistic methods.

Based on the methods (Chapter 4) for hybrid planning, diagnostic reasoning and explanation generation, discrepancy detection under uncertainty and checking for their relevancy, and guided replanning and repairs, we propose a generic execution monitoring framework that is presented in Algorithm 3.

5.1 Autonomous Execution Monitoring

Our autonomous execution monitoring algorithm takes as input a robotic domain description $\mathscr{D}^k$, an initial state s_0, a goal state s_g, an upper bound k on time steps, a set $\mathscr{R}$ of pairs of robots and their components that may get broken, a relation *disables* that describes which effects of actions are affected if some robots and their components in $\mathscr{R}$ are broken, and a set M of monitored fluents (denoting what is being observed). The algorithm returns success if a goal state is reached, and returns a failure report otherwise.

First, Algorithm 3 tries to find a plan P that leads to the goal state (line 1); if such a plan does not exist, P is null, and the algorithm returns failure. Suppose that a plan P exists. Then the algorithm starts running the plan and makes observations over the monitored fluents in M from time to time.

Whenever a new observation becomes available, the algorithm obtains the current values o_t^M of the monitored fluents in M (line 9), and updates the set of observations O_t^M made so far. Then the algorithm infers the full expected state s_e reached from s_0 by the part $P_{<}t$ of the plan executed so far (line 11), and checks whether there is a discrepancy between the expected state s_e and the partially observed current state o_t^M (line 12).

If there exists a discrepancy, then Algorithm 3 generates a full valid state s_v that is consistent with the observations o_t^M and that utilizes knowledge from ex-

Algorithm 3 Autonomous Execution Monitoring with Diagnostic Reasoning and Repair

Input: $\mathscr{D}^k$, $\mathscr{R}$, *disables*, s_0, s_g, k, M, $prob_b$, $pref_x$.

Output: Return success if the goal state s_g is achieved, return a failure report otherwise.

```
    // Find a plan $P = \langle A_0, \dots, A_{n-1} \rangle$ $(n < k)$ (Definition 1).
 1: $P \leftarrow Plan(\mathscr{D}^k, s_0, s_g, k)$;
 2: **if** $P$ is not null **then**
 3:     $makespan = |P|$;
    // Monitor the execution of $P$.
    // Initially, the set $O_0^M$ of observations and the diagnosis $X_0$ are empty.
 4:     $t = 0,\ O_t^M = \{\},\ X_t = \{\}$;
 5:     **while** $t < makespan$ **do**
 6:         Execute $A_t$;
 7:         $t = t + 1$;
    // Observe the monitored fluents $M$ from time to time.
 8:         **if** a new observation is made at step $t$ **then**
 9:             $o_t^M \leftarrow$ Obtain the set of current values of $M$;
10:             $O_t^M \leftarrow$ Add $o_t^M$ to the set of earlier observations $O_{t-1}^M$;
    // Infer the expected state $s_e$ at step $t$, considering the plan $P_{<t} = \langle A_0, \dots, A_{t-1} \rangle$ executed so far and the set $X_{<t} = \cup_{i<t} X_i$ of earlier diagnoses (Definition 4).
11:             $s_e \leftarrow Predict(\mathscr{D}_{diag}^t, s_0, P_{<t}, O_t^M, X_{<t})$;
    // Check whether there is a discrepancy between the expected state $s_e$ and the current observations $o_t^M$ (Definition 5).
12:             $discrepancy \leftarrow Discrepancy(\mathscr{D}_{state}, s_e, o_t^M)$;
13:             **if** $discrepancy$ **then**
    // Infer a valid full state $s_v$ from the partially observed current state $o_t^M$, that is closest to the expected state $s_e$ (Definition 6).
14:                 $s_v \leftarrow ValidState(\mathscr{D}_{state}, s_e, o_t^M)$;
    // Check whether the detected discrepancy is relevant to the rest $P_{\geq t} = \langle A_t, \dots, A_{makespan-1} \rangle$ of the plan (Definition 7).
15:                 $relevant \leftarrow Relevancy(\mathscr{D}_{diag}^t, s_e, o_t^M, s_v, P_{\geq t}, s_g, X_{<t})$;
16:                 **if** $relevant$ **then**
    // Find a most-probable min-cardinality diagnosis $X^E$ with an explanation, for the detected relevant discrepancy (Definition 3), with respect to $prob_b$.
17:                     $X^E \leftarrow Diagnose(\mathscr{D}_{diag}^t, \mathscr{R}, s_0, P_{<t}, O_t^M, X_{<t}, prob_b)$;
18:                     $X_t = \{(r, p, i) | (r, p, A, i) \in X^E\}$;
    // Infer the expected state $s_t$ at step $t$, considering the plan $P_{<t}$ executed so far and the current diagnosis $X_t$.
19:                     $s_t \leftarrow Predict(\mathscr{D}_{diag}^t, s_0, P_{<t}, O_t^M, X_t)$;
```

```
    // Replan with a most-preferable min-cardinality set $Y \subset \{(r,p)|(r,p,i) \in X_t\}$
    of repairs with respect to $pref_x$: take $s_t$ as the initial state and $k-t$ as the upper
    bound on makespan (Definition 9).
20:             $P,\ Y \leftarrow RePlan(\mathscr{D}_R^{k-t}, s_t, s_g, X_t, k-t, pref_x)$;
21:             if $P$ is not null then
22:                 $makespan = t + |P|$;
    // Shift time indices of each action $A_i$ in $P$ by $t$.
    // Generate a report with explanations and repair requests.
23:                 Repair the broken parts in $Y$;
24:             else
25:                 return Failure;
26:     return Success;
27: else
28:     return Failure;
```

pected state s_e as much as possible (line 14), and then checks the relevancy of the detected discrepancy by checking whether the rest of the plan $P_{\geq t}$ reaches the goal state s_g from s_v (line 15).

If the discrepancy is found relevant, then Algorithm 3 generates a most-probable min-cardinality diagnosis with explanations X^E (line 17) utilizing the given knowledge $prob_b$ about the likelihood of components getting broken. Next, the algorithm infers the current state by computing an expected state s_t at step t reached by the plan $P_{<t}$ executed so far and considering the current diagnosis X_t included in X_E. Then, the algorithm continues by replanning at s_t guided by this diagnosis, and possibly repairing a min-cardinality set Y of components (line 20) utilizing the user's preferences and experiences $pref_x$.

Once all the actions in the plan are executed, the goal is reached with a success.

5.2 Interactive Execution Monitoring

Interactive execution monitoring is important to involve the expertise of the user in the loop. Thanks to the modular and formal aspects of our execution monitoring framework, we can revise Algorithm 3 easily to make it more interactive.

For instance, instead of generating a diagnosis with explanations and automatically replanning with respect to this diagnosis, the algorithm can generate a set of most-probable min-cardinality diagnoses with explanations, and then consult the user as to which one to consider. Meanwhile, the likelihood of each robotic component getting broken is updated based on the selected diagnoses, utilizing the user's expertise and experiences.

Instead of generating a single replan P with repairs Y, the algorithm can generate a set of replans with the minimum number of preferable repairs, and then consult the user as to which replan to consider to reach the goal. Providing such alternatives is important as the repairs may require different amounts of time and energy, depending on the available resources.

5.3 Case Study with a Collaborative Service Robotics Domain

Let us demonstrate the use of our execution monitoring algorithm (Algorithm 3) with a service robotics application, where two bi-manual mobile service robots collaborate to set up a dinner table in a kitchen.

This robotic domain is interesting since i) it involves both manipulation tasks (e.g., via pick and place actions) and navigation tasks (e.g., via move actions), ii) the executability of manipulation and navigation actions requires feasibility checks (e.g., via the reachability checks and motion planning queries), and iii) the locations of the objects and the robots in the kitchen may be unobservable depending on their whereabouts.

5.3.1 ASP Representation of the Kitchen Domain

We consider the following set of atoms to describe the domains of variables used in our ASP formulation:

- $rob(r)$: r is a robot,
- $obj(o)$: o is an object that the robots can manipulate in the environment,
- $thg(th)$: th is a thing (i.e., robots and objects),
- $manip(m)$: m is a (e.g., left or right) manipulator of a robot,
- $robloc(l)$: l is a location that a robot can be at,
- $objloc(l)$: l is a possible location for an object,
- $comloc(l)$: l is a possible location both for an object and for a robot,
- $loc(l)$: l is a possible location for a thing,
- $time(i)$: $i \in \{0, \ldots, k\}$ where k is a given upper bound on the makespan,
- $atime(i)$: $i \in \{0, \ldots, k-1\}$ where k is a given upper bound on the makespan.

We also denote by $man(m,r)$ a manipulator m of a robot r.

We consider the following set of fluents and actions:

- $at(th,l,i)$: thing th is located at l at time step i,

- $move(r,l,i)$: the robot r moves to location l at time step i,
- $pickUp(r,m,o,i)$: the robot r picks up the object o with its manipulator m at time step i,
- $placeOn(r,m,l,i)$: the robot r places the object at its manipulator m onto a location l at time step i.

First, we define the states (i.e., nodes) of the transition system. Initially, things can be places arbitrarily:

$$\{at(th,l,0)\} \leftarrow thg(th), loc(l).$$

At every time step, a thing should be located at some location (existence), but cannot be located at two different locations (uniqueness).

$$\leftarrow \{at(th,l,i){:}loc(l)\}0, thg(th), time(i).$$
$$\leftarrow 2\{at(th,l,i){:}loc(l)\}, thg(th), time(i).$$

Moreover, the things should be at their relevant locations:

$$\leftarrow at(o,l,i), obj(o), loc(l), not\ objloc(l), time(i).$$
$$\leftarrow at(r,l,i), rob(r), loc(l), not\ robloc(l), time(i).$$

We should also ensure that a robot cannot hold more than one object in its manipulator:

$$\leftarrow 2\{at(o,man(r,m),i){:}obj(o)\}, rob(r), manip(m), time(i).$$

We define the transitions (i.e., edges) of the transition system. Consider the edges that describe nonoccurrences of actions. We express that a thing remains to be where it is, unless some action causes a change of its location as follows:

$$\{at(th,l,i{+}1)\} \leftarrow at(th,l,i), thg(th), loc(l), atime(i).$$

The other edges describe occurrences of actions. Actions may occur at any time:

$$\begin{aligned}
&\{move(r,l,t)\} \leftarrow rob(r), robloc(l), atime(i).\\
&\{pickUp(r,m,o,i)\} \leftarrow rob(r), manip(m), obj(o), atime(i).\\
&\{placeOn(r,m,l,i)\} \leftarrow rob(r), manip(m), objloc(l), atime(i).
\end{aligned}$$

However, an action cannot occur at a state that does not satisfy its preconditions. For instance, a robot *r* cannot move to a location *l* if it is already there:

$$\leftarrow move(r,l,i), at(r,l,i), rob(r), robloc(l), atime(i).$$

A robot *r* cannot pick up an object *o* if they are at different locations, if another robot *r1* is holding *o* but the robot is at a different location, or if the object *o* is on the table but *r* is not around the table:

$$\begin{aligned}
&\leftarrow pickUp(r,m,o,i), at(o,l,i), at(r,l1,i), rob(r), obj(o),\\
&\qquad manip(m), robloc(l1), comloc(l), atime(i) \quad (l \neq l1).\\
&\leftarrow pickUp(r,m,o,i), at(o,man(r1,m1),i), at(r,l,i), at(r1,l1,i),\\
&\qquad rob(r), rob(r1), manip(m), manip(m1), obj(o),\\
&\qquad robloc(l), robloc(l1), atime(i) \quad (l \neq l1).\\
&\leftarrow pickUp(r,m,o,i), \{at(r,TableLeft,i); at(r,TableRight,i)\}0,\\
&\qquad at(o,Table,i), rob(r), manip(m), obj(o), atime(i).
\end{aligned}$$

Similarly, a robot r cannot place an object onto some location l with its manipulator m if the manipulator is not holding any object, if the robot r is not at l, if $l{=}man(r1,m1)$ and the robots r and $r1$ are not at the same location, or if $l{=}Table$ but the robot r is not around the table:

$$
\begin{aligned}
\leftarrow\ & placeOn(r,m,l,i), \{at(o,man(r,m),i){:}obj(o)\}0, rob(r),\\
& \quad manip(m), objloc(l), atime(i).\\
\leftarrow\ & placeOn(r,m,l,i), at(r,l1,i), rob(r), manip(m),\\
& \quad comloc(l), robloc(l1), atime(i) \quad (l \neq l1).\\
\leftarrow\ & placeOn(r,m,man(r1,m1),i), at(r1,l1,t), at(r,l,t),\\
& \quad rob(r), rob(r1), manip(m), manip(m1), robloc(l),\\
& \quad robloc(l1), atime(i) \quad (l \neq l1).\\
\leftarrow\ & placeOn(r,m,Table,i), rob(r), manip(m), atime(i),\\
& \quad \{at(r,TableLeft,i); at(r,TableRight,i)\}0.
\end{aligned}
$$

Furthermore, we consider some noconcurreny constraints to ensure that navigation and manipulation actions do not occur at the same time:

$$
\begin{aligned}
\leftarrow\ & move(r,l,i), pickUp(r,m,o,i), rob(r), robloc(l),\\
& \quad manip(m), obj(o), atime(i).\\
\leftarrow\ & move(r,l,i), placeOn(r,m,l1,i), rob(r), manip(m),\\
& \quad robloc(l), objloc(l1), atime(i).
\end{aligned}
$$

Since robots are operating in a continuous space, for the feasibility of actions, we embed the relevant low-level feasibility checks into the domain description. For instance, a robot r cannot move from a location l to another location $l1$ if there is no collision-free trajectory that it can follow. A robot r cannot pick up an object o if the robot cannot reach and grasp the object without any collisions. Similarly, a robot r cannot place an object onto a location if the robot cannot reach the location or place the object without any collisions.

$$\begin{aligned}
&\leftarrow at(r,l1,i), move(r,l,i), not\ \&moveIsFeasible[r,l,l1](),\\
&\quad rob(r), robloc(l), robloc(l1), atime(i).\\
&\leftarrow at(r,l,i), pickUp(r,m,o,i), rob(r), objloc(l), manip(m),\\
&\quad obj(o), not\ \&pickUpIsFeasible[r,l,o,m](), atime(i).\\
&\leftarrow at(r,l,i), placeOn(r,m,l,i), rob(r), objloc(l), manip(m),\\
&\quad not\ \&placeOnIsFeasible[r,l,m](), atime(i).
\end{aligned} \tag{5.1}$$

Let us now define what changes after execution of an action. For instance, when a robot r moves to a location l at step i, its location becomes l at the next time step:

$$at(r,l,i{+}1) \leftarrow move(r,l,i), rob(r), robloc(l), atime(i).$$

When a robot r picks up an object o with its manipulator, the location of object becomes the manipulator at the next time step:

$$\begin{aligned}
&at(o,man(r,m),i{+}1) \leftarrow pickUp(r,m,o,i), rob(r), manip(m),\\
&\quad obj(o), atime(i).
\end{aligned}$$

When a robot r places the object o that it is holding with its manipulator to a location l, the location of the object becomes l at the next time step:

$$\begin{aligned}
&at(o,l,i{+}1) \leftarrow placeOn(r,m,l,i), at(o,man(r,m),i),\\
&\quad rob(r), objloc(l), manip(m), atime(i).
\end{aligned}$$

5.3.2 Importance of Feasibility Checks

As described in Chapter 4, our methods for generating (re)plans, predictions, diagnoses, explanations and repairs utilize the robotic domain descriptions ($\mathscr{D}$ and $\mathscr{D}_{diag}$), which embed low-level feasibility checks described in Chapter 3 into action descriptions. In this section, we present a scenario (Figure 5.3.1) to show the importance of employing such hybrid reasoning that integrates low-level feasibility checks into high-level reasoning.

Consider the kitchen domain described above, with two robots *R1* and *R2*. Suppose that the base of *R2* is diagnosed as broken. Currently, at time step t, *R1* is in *Room1*, *R2* is in *Room2*, and there is a *Spoon* on *Table1* in *Room2*. The goal is placing *Spoon* on *Table2* in *Room2*. The rest of the plan will be executed as follows: *R1* moves to *Table1*, picks up the *Spoon*, moves to *Table2* (i.e., the other table in *Room2*), places *Spoon* on *Table2*, and moves to *Room1*.

Suppose that the first observation o_t^M is done at time step $t+4$: *Spoon* is on *Table1*. To check for a discrepancy, the expected state at $t+4$ needs to be predicted. Since the feasibility checks are embedded in the preconditions of actions in the domain description as described above, the expected state s_e at $t+4$ is predicted as follows: *R1* is in *Room1*, *R2* is in *Room2*, and *Spoon* is on *Table1*. Then there is no discrepancy between s_e and o_t^M.

If the feasibility checks were not embedded in the preconditions of actions in the domain description as described above, the expected state s_e at $t+4$ would be predicted as follows: *R1* is in *Room2*, *R2* is in *Room2*, and *Spoon* is on *Table2*. Then there is a discrepancy between s_e and o_t^M.

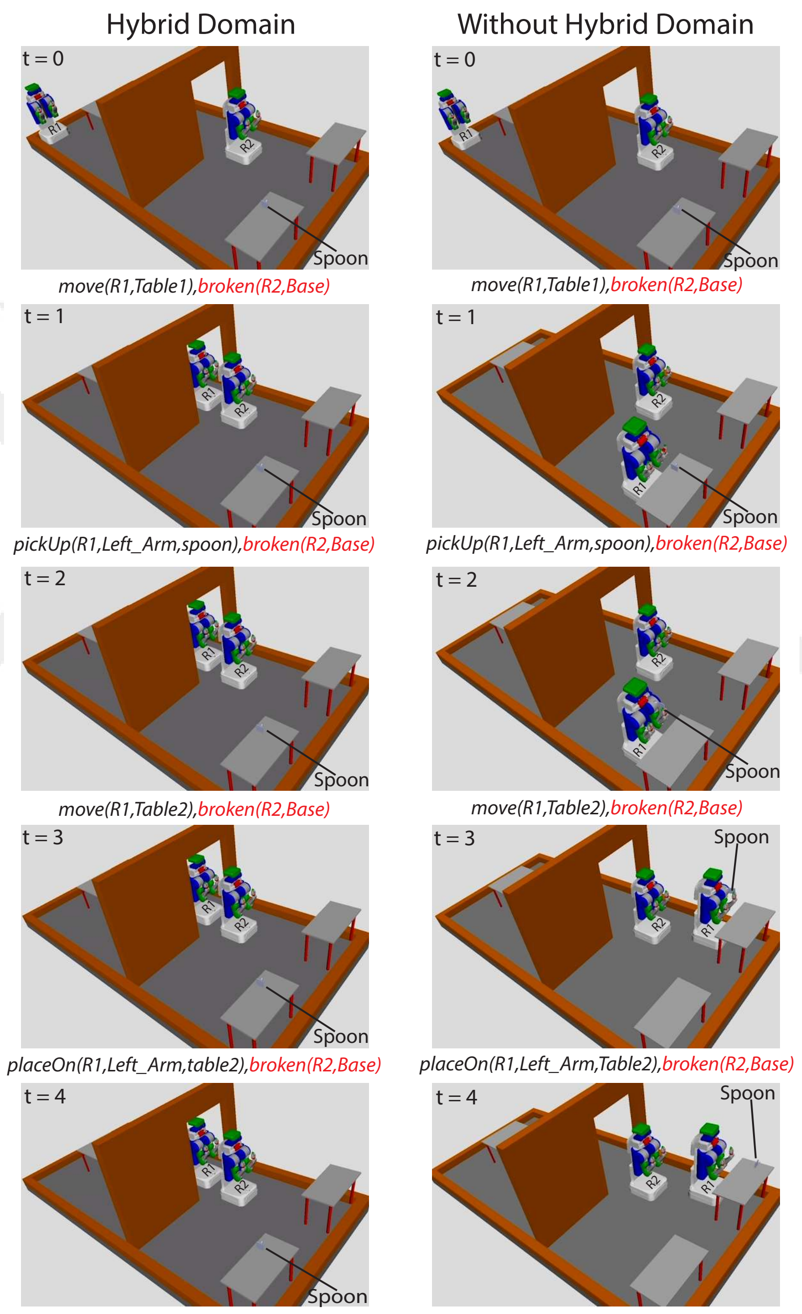

Hybrid Domain
Without Hybrid Domain
t = 0
R1
R2
Spoon
t = 0
R1
R2
Spoon
move(R1,Table1),broken(R2,Base)
move(R1,Table1),broken(R2,Base)
t = 1
R1
R2
Spoon
t = 1
R2
R1
Spoon
pickUp(R1,Left_Arm,spoon),broken(R2,Base)
pickUp(R1,Left_Arm,spoon),broken(R2,Base)
t = 2
R1
R2
Spoon
t = 2
R2
R1
Spoon
move(R1,Table2),broken(R2,Base)
move(R1,Table2),broken(R2,Base)
t = 3
R1
R2
Spoon
t = 3
Spoon
R2
R1
placeOn(R1,Left_Arm,table2),broken(R2,Base)
placeOn(R1,Left_Arm,Table2),broken(R2,Base)
t = 4
R1
R2
Spoon
t = 4
Spoon
R2
R1

Figure 5.3.1: Plan execution with/without hybrid domain. Each sub-figure represents world state at different time steps.

The left column of Figure 5.3.1 shows how the world states change over time steps $t = 0, 1, 2, 3, 4$ during the execution of this plan, when hybrid representation and reasoning are utilized. Since the collision checks are performed for the first move action, *R1* will move closer to *R2*, but cannot go any further because there is no feasible path from its current location to *Table1*. Therefore, at the end of execution of the plan, the *Spoon* will still be on the *Table1*.

The right column of Figure 5.3.1 shows the world states during the execution of the given plan when hybrid representation and reasoning are not utilized (i.e., none of the feasibility checks are embedded in the robot domain description). Since no collision check is performed, the first move action of *R1* to reach the *Table1* at time step 0 seems feasible. Then, at the end of execution of the plan, the *Spoon* will be on the *Table2*.

These two predicted states generated by with/without hybrid reasoning demonstrate that not using feasibility checks may lead to incorrect predictions, as false positives cannot be eliminated.

5.3.3 Execution Monitoring of a Plan – A Sample Scenario

We introduce an illustrative example in the kitchen domain, to show how our execution monitoring algorithm (Algorithm 3) can be used autonomously or interactively, emphasizing the differences among the three diagnostic reasoning approaches explained in Section 4.2.2.

Consider the kitchen domain with two robots (*R1* and *R2*), two distinct objects (*Knife* and *Spoon*), and three distinct locations (*ShelfA*, *ShelfB*, and *Table*). Assume that the world is partial observable: only the objects located on the table can be observed. Let's also assume that the bases of *R1* and *R2* are more likely to get broken compared to other components. Initially, at s_0, *Table* is empty, *Knife* is on

ShelfA, and *R1* is near *ShelfA*, while *Spoon* is on *ShelfB* and *R2* is near *ShelfB*. At the goal state s_g, both *Knife* and *Spoon* are on *Table*.

The algorithm solves this planning problem instances by utilizing the kitchen domain description above:

$$\begin{aligned}&P{=}\langle \mathit{pickUp(R1,LeftArm,Knife,0)},\ \mathit{pickUp(R2,LeftArm,Spoon,0)},\\ &\quad \mathit{move(R1,TableLeft,1)},\ \mathit{move(R2,TableRight,1)},\\ &\quad \mathit{placeOn(R1,LeftArm,Table,2)},\ \mathit{placeOn(R2,LeftArm,Table,2)}\rangle.\end{aligned}$$

Then, the robots start executing the plan P. The left column of Figure 5.3.3 illustrates the expected world states during a successful execution of the plan P, while the right column shows the observations made during a real execution of P. Figure 5.3.3 shows the dynamic simulations for both the expected and real executions of the plan P, for better understandability.

While P is being executed, suppose that an observation o_t^M has been performed at time step $t = 3$: only *Spoon* is on *Table*. The expected state s_e at $t = 3$ is predicted as follows: $at(R1, \mathit{TableLeft})$, $at(R2, \mathit{TableRight})$, $at(\mathit{Knife}, \mathit{Table})$, $at(\mathit{Spoon}, \mathit{Table})$. Since *Knife* is not observed to be on *Table* as expected, there is a discrepancy between s_e and o_3^M.

Note that once a full valid state s_v is estimated (where *Spoon* is on *Table* but *Knife* is not on *Table*), this discrepancy is found relevant for the rest $P_{\geq} = \emptyset$ of the plan: executing $P_{\geq}$ at s_v does not lead the plan to the goal state s_g.

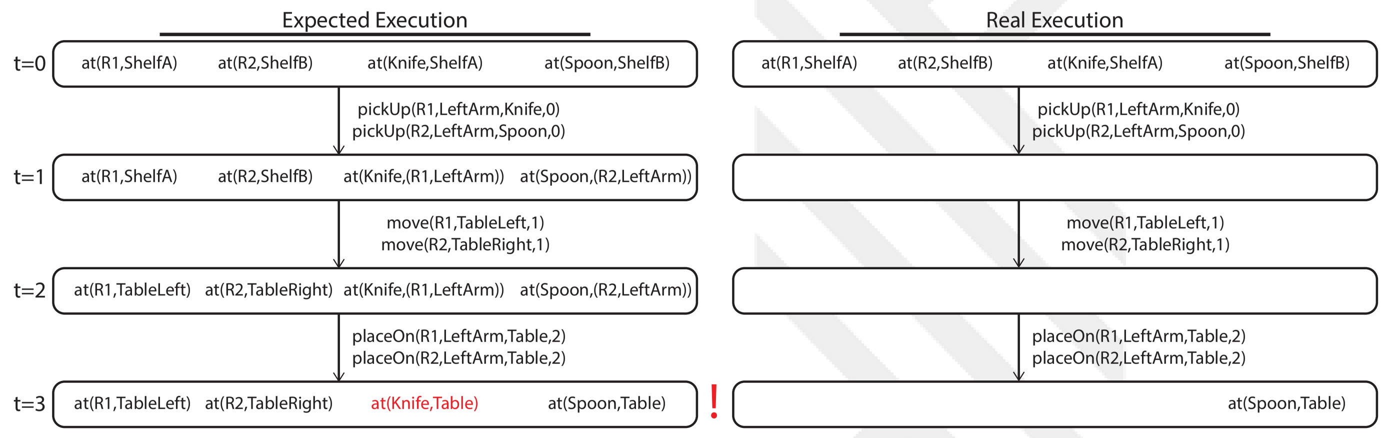


Figure 5.3.2: The expected execution (with expected world states) and a real execution (with observations) of the initial plan P at time step $t = 0$.

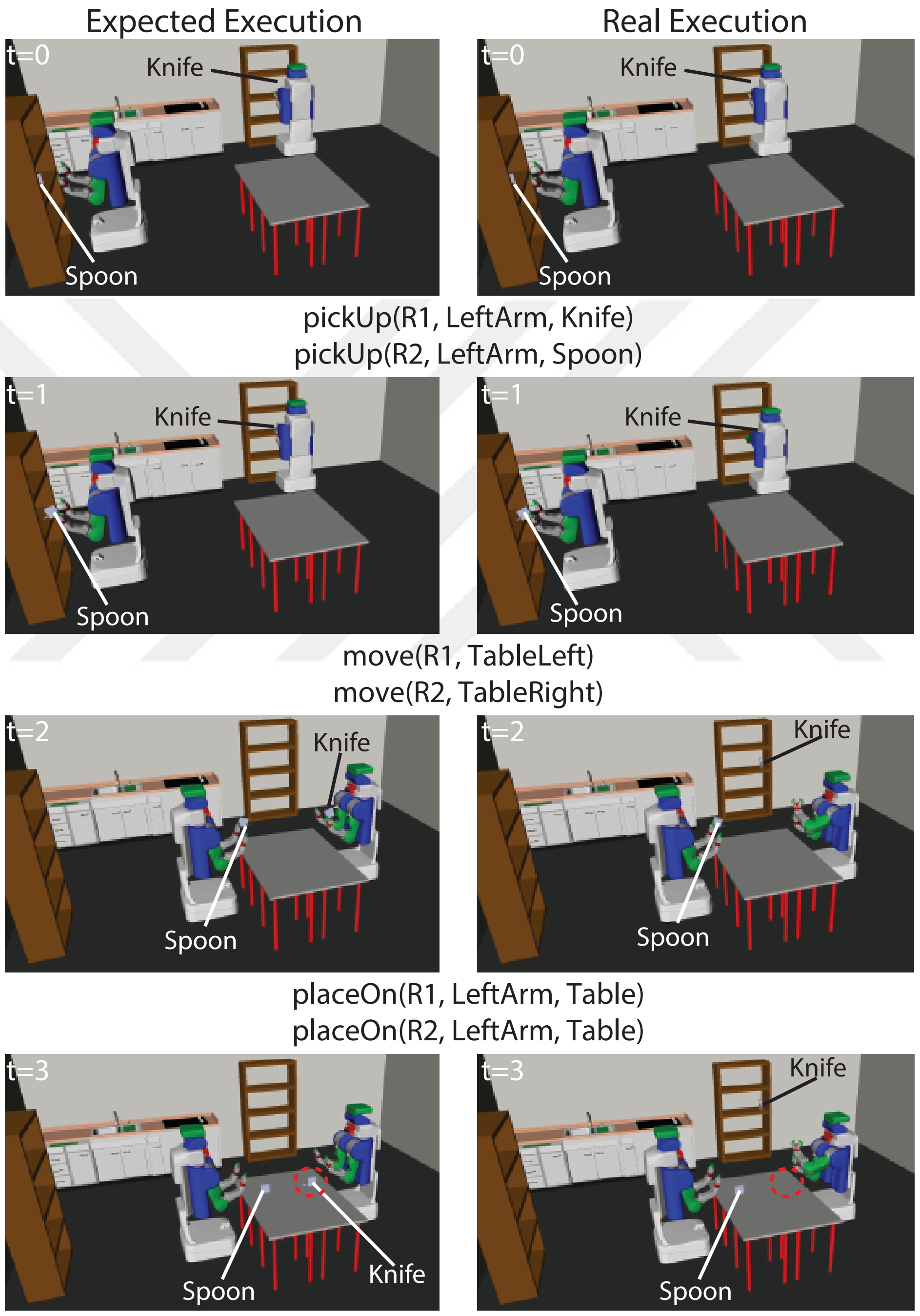

Expected Execution
Real Execution
t=0
Knife
Spoon
pickUp(R1, LeftArm, Knife)
pickUp(R2, LeftArm, Spoon)
t=1
move(R1, TableLeft)
move(R2, TableRight)
t=2
placeOn(R1, LeftArm, Table)
placeOn(R2, LeftArm, Table)
t=3

Figure 5.3.3: The dynamic simulations for the expected and the real executions of the initial plan P at time step $t = 0$ presented in Figure 5.3.3.

Then, according to the autonomous mode of the algorithm, considering that the bases are more likely to get broken, the following most-probable min-cardinality diagnosis is computed for the relevant discrepancy detected at time step $t = 3$

$$X_3 = \{(\mathit{R1}, \mathit{Base}, 1)\}$$

with an explanation: "*Base* of *R1* got broken at time step $t = 1$, so *R1* could not move to *TableLeft* at time step $t = 1$ as expected." Note that since this is the first diagnosis, all three diagnostic reasoning approaches (i.e., augmented, reset, revised) generate the same hypothesis.

In the interactive mode of this algorithm, the following set of 3 min-cardinality diagnoses is presented to the user

$$\{\{(\mathit{R1}, \mathit{Base}, 1)\}, \{(\mathit{R1}, \mathit{LeftArm}, 0)\}, \{(\mathit{R1}, \mathit{LeftArm}, 2)\}\}$$

with explanations:

- The base of *R1* got broken at time step $t = 1$, so *R1* could not move to *TableLeft* at time step $t = 1$ as expected.
- The left arm of *R1* got broken at time step $t = 0$, so *R1* could not pick up *Knife* at time step $t = 0$ as expected.
- The left arm of *R1* got broken at time step $t = 2$, so *R1* could not place *Knife* on *Table* at time step $t = 2$ as expected.

Then, the user picks one of them for the algorithm to proceed with replanning. For instance, considering that the bases are more likely to get broken, the user might pick the first diagnosis to proceed: $X_3 = \{(\mathit{R1}, \mathit{Base}, 1)\}$; or considering that the left arm of *R1* often breaks down early in a plan, the user may pick the second diagnosis: $X_3 = \{(\mathit{R1}, \mathit{LeftArm}, 0)\}$.

Suppose that the user picks $X_3 = \{(\mathit{R1}, \mathit{Base}, 1)\}$. With this diagnosis with explanation, the execution monitoring algorithm predicts the current expected state s_t as follows: $at(\mathit{R1}, \mathit{ShelfA})$, $at(\mathit{R2}, \mathit{TableRight})$, $at(\mathit{Knife}, \mathit{R1Hand})$, $at(\mathit{Spoon}, \mathit{Table})$. After that, the algorithm replans to reach the goal state s_g from s_3 as follows:

$$P{=}\langle \mathit{move(R2,ShelfA,3),\ pickUp(R2,LeftArm,Knife,4),}$$
$$\mathit{move(R2,TableLeft,5),\ placeOn(R2,LeftArm,Table,6)}\rangle.$$

Then, the robots start executing the new plan P at time step $t = 3$. The expected execution and a real execution of P are shown in the left and right columns of Figure 5.3.3, respectively, while Figure 5.3.5 illustrates their dynamic simulations.

Assume that the next observation o_t^M takes place at time step $t = 7$: only *Spoon* is on *Table*. The expected state s_e at $t = 7$ is predicted as follows: $at(\mathit{R1}, \mathit{ShelfA})$, $at(R2, \mathit{TableLeft})$, $at(\mathit{Knife}, \mathit{Table})$, $at(\mathit{Spoon}, \mathit{Table})$. Since *Knife* is not observed to be on *Table* as expected, there is a discrepancy between s_e and o_7^M. Furthermore, this discrepancy is relevant as the goal state is not reached by the execution of the rest of the plan.

Then, according to the interactive mode of the algorithm, a set of min-cardinality diagnoses is computed so that the user can pick the most probable one.

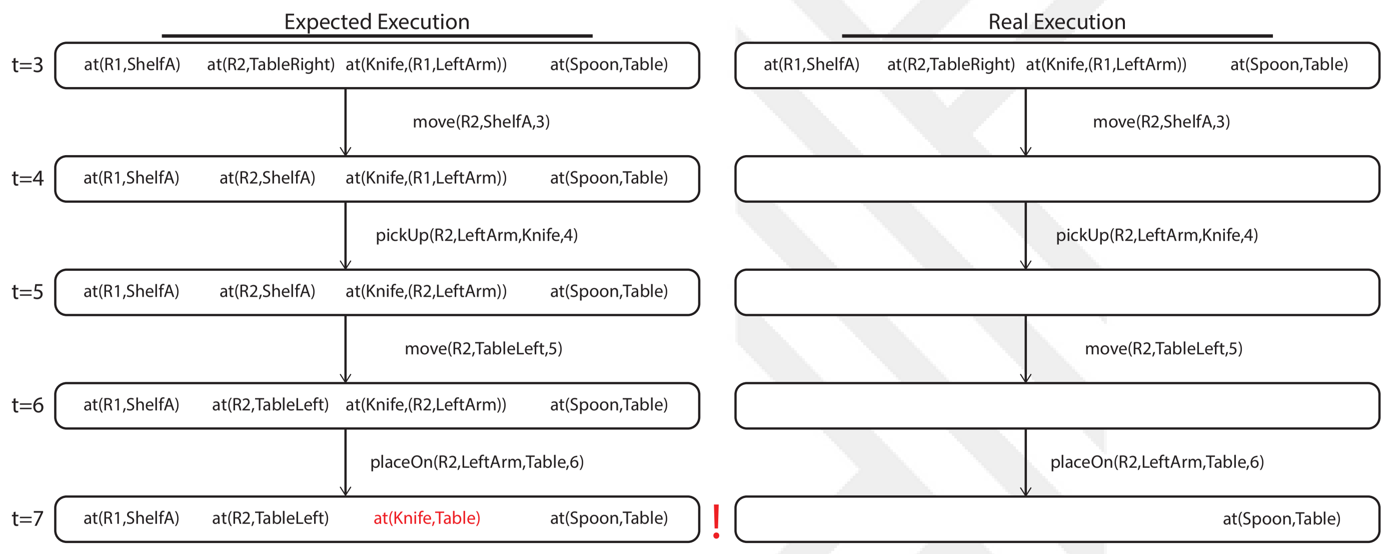


Figure 5.3.4: The expected execution (with expected world states) and a real execution (with observations) of the replan P computed at time step $t = 3$ considering the diagnosis $X_t = \{(R1, Base, 1)\}$.

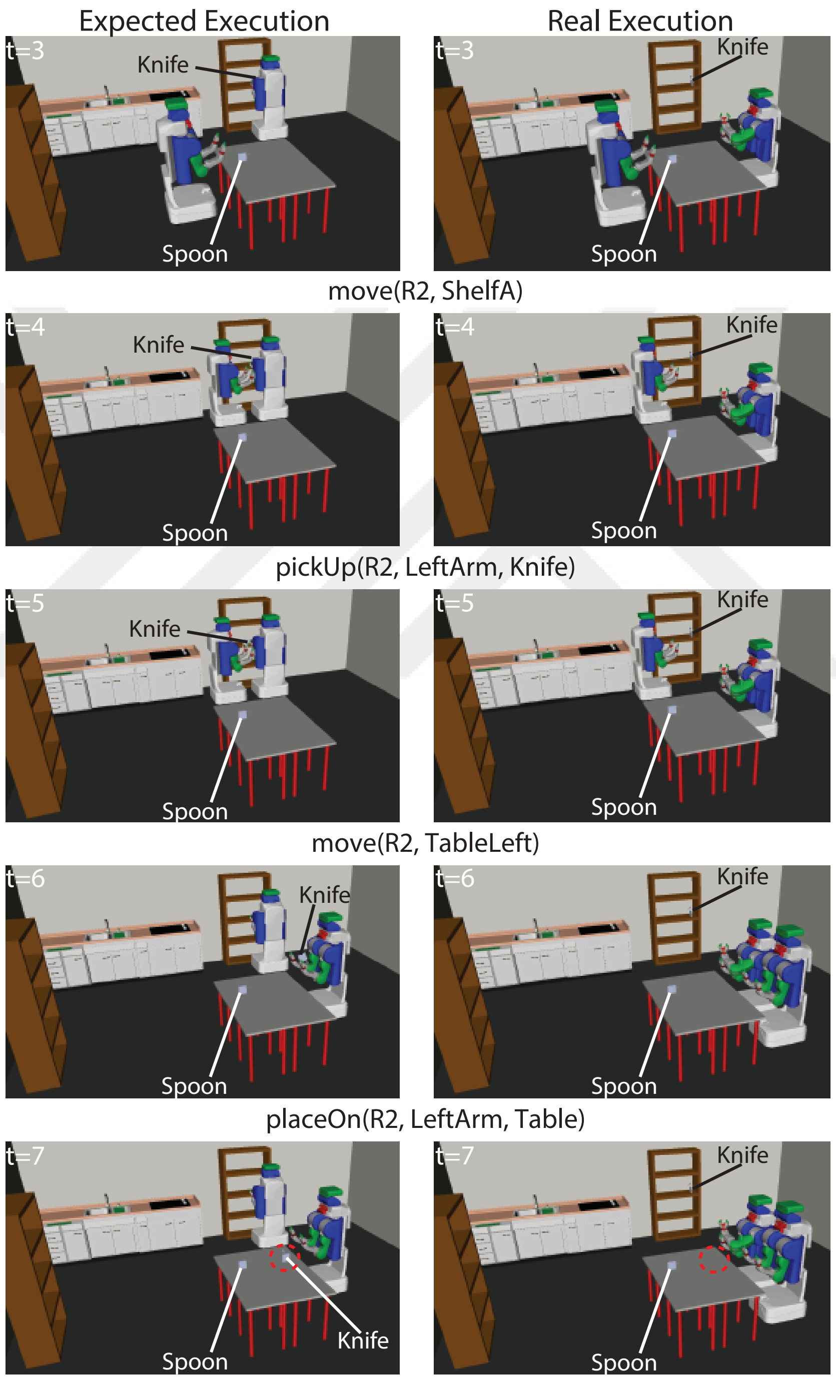


Figure 5.3.5: The dynamic simulations for the expected and the real executions of the replan P at time step $t = 3$ presented in Figure 5.3.3.

Augmented diagnoses: For instance, a set of 4 min-cardinality augmented diagnoses can be generated based on the current observation o^M_7, while imposing the previous diagnoses $X_{<7} = X_3 = \{(\mathit{R1}, \mathit{Base}, 1)\}$:

$$\{\{(\mathit{R1}, \mathit{Base}, 1), (\mathit{R2}, \mathit{Base}, 3)\}, \{(\mathit{R1}, \mathit{Base}, 1), (\mathit{R2}, \mathit{Base}, 5)\},$$
$$\{(\mathit{R1}, \mathit{Base}, 1), (\mathit{R2}, \mathit{LeftArm}, 4)\}, \{(\mathit{R1}, \mathit{Base}, 1), (\mathit{R2}, \mathit{LeftArm}, 6)\}\}.$$

Once these augmented diagnoses with their explanations are presented to the user, assume that $X_7 = \{(\mathit{R1}, \mathit{Base}, 1), (\mathit{R2}, \mathit{Base}, 3)\}$ is selected by the user as the min-cardinality diagnosis with the following explanation: "*Base* of *R1* got broken at time step $t = 1$, so *R1* could not move to *TableLeft* at time step $t = 1$ as expected; and since *Base* of *R2* got broken at time step $t = 3$, *R2* could not move to *ShelfA* at time step $t = 3$ as expected."

However, since the bases of both robots are diagnosed as broken, the robots cannot move around, and thus replanning fails. This example shows that although augmented diagnoses generously maintain the previous diagnoses, they put more constraints on replanning (as more number of parts are assumed to be broken) making it harder to find a replan.

Reset diagnoses: Alternatively, a set of 4 min-cardinality reset diagnoses can be generated based on the current observation o^M_7, but without considering any of the previous diagnoses:

$$\{\{(\mathit{R2}, \mathit{Base}, 3)\}, \{(\mathit{R2}, \mathit{Base}, 5)\}, \{(\mathit{R2}, \mathit{LeftArm}, 4)\},$$
$$\{(\mathit{R2}, \mathit{LeftArm}, 6)\}\}.$$

Suppose that the user selects the first min-cardinality diagnosis $X_7 = \{(\mathit{R2}, \mathit{Base}, 3)\}$ with explanation: "*Base* of *R2* is broken, so *R2* could not move to *ShelfA* at time step 3 as expected."

Considering this diagnosis, the algorithm predicts the expected current state s_t for time step $t=7$, and replans to reach the goal state s_g from s_7 as follows:

$$P{=}\langle \textit{move(R1,TableLeft,7), placeOn(R1,LeftArm,Table,8)}\rangle.$$

The expected execution and a real execution of the new plan P are shown in Figure 5.3.3, respectively.

Suppose that, at time step $t=9$, another observation o_t^M is made: only *Spoon* is on *Table*. The expected state s_e at $t=9$ is predicted as follows: $at(\mathit{R1},\mathit{TableLeft})$, $at(\mathit{R2},\mathit{TableRight})$, $at(\mathit{Knife},\mathit{Table})$, $at(\mathit{Spoon},\mathit{Table})$. There is a discrepancy between s_e and o_9^M, and it is relevant for the rest of the plan.

The following set of 2 min-cardinality reset diagnoses can be generated based on the current observation o_t^9:

$$\{\{(\mathit{R1},\mathit{Base},7)\},\{(\mathit{R1},\mathit{LeftArm},8)\}\}.$$

Assume that the user picks the reset diagnosis $X_9=\{(\mathit{R1},\mathit{Base},7)\}$. Note that the diagnosed broken component is identical with that of the first diagnosis X_3. This example shows that reset diagnoses may unnecessarily repeat themselves in terms of broken components, as they forget the previous diagnoses and the observations. As a result, replanning guided with a reset diagnosis may keep generating the same plan that may not reach the goal.

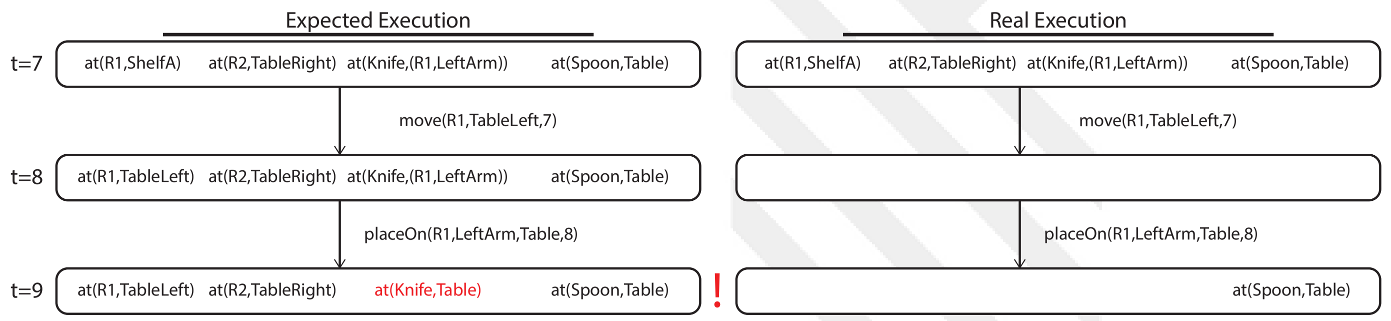


Figure 5.3.6: The expected execution (with expected world states) and a real execution (with observations) of the replan P computed at time step $t = 7$ considering the reset diagnosis $X_t = \{(R2, Base, 3)\}$.

Revised diagnosis: Alternatively, a set of 2 min-cardinality revised diagnoses can be generated for the detected relevant discrepancy at time step $t = 7$, considering both the current observation o_7^M and the previous observation o_3^M (i.e., $O_7^M = \{o_3^M, o_7^M\}$) while updating the previous diagnoses as follows:

$$\{\{(R1, LeftArm, 0)\}, \{(R1, LeftArm, 2)\}\}.$$

Assume that the user selects the revised diagnosis $X_7 = \{(R1, LeftArm, 0)\}$ with its explanation: "*LeftArm* of *R1* got broken at time step $t = 0$, so *R1* could not pick up *Knife* at time step $t = 0$ as expected."

Once the algorithm predicts the expected current state s_7 with respect to this diagnosis, it proceeds with replanning:

$$P = \langle move(R2,ShelfA,7),\ pickUp(R2,LeftArm,Knife,8),$$
$$move(R2,TableLeft,9),\ placeOn(R2,LeftArm,Table,10)\rangle.$$

The expected execution and a successful real execution of this new plan P at time step $t = 7$ are shown in the left and right columns of Figure 5.3.3, respectively. Figure 5.3.8 illustrates their dynamic simulations.

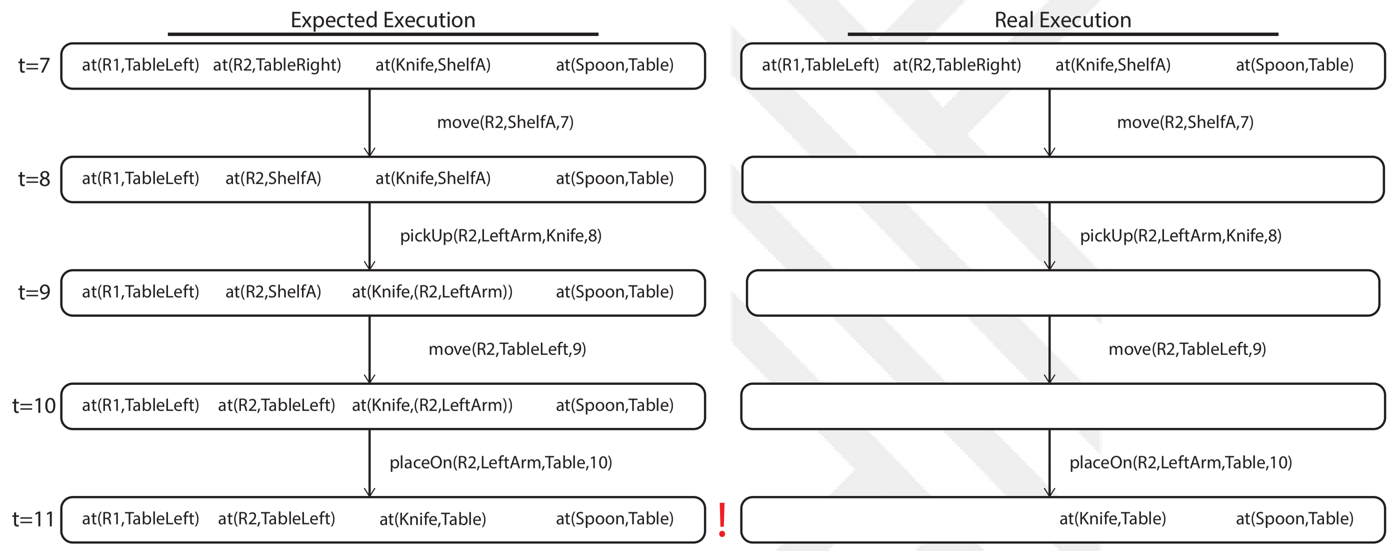

Expected Execution
Real Execution
t=7
at(R1,TableLeft) at(R2,TableRight) at(Knife,ShelfA) at(Spoon,Table)
move(R2,ShelfA,7)
t=8
at(R1,TableLeft) at(R2,ShelfA) at(Knife,ShelfA) at(Spoon,Table)
pickUp(R2,LeftArm,Knife,8)
t=9
at(R1,TableLeft) at(R2,ShelfA) at(Knife,(R2,LeftArm)) at(Spoon,Table)
move(R2,TableLeft,9)
t=10
at(R1,TableLeft) at(R2,TableLeft) at(Knife,(R2,LeftArm)) at(Spoon,Table)
placeOn(R2,LeftArm,Table,10)
t=11
at(R1,TableLeft) at(R2,TableLeft) at(Knife,Table) at(Spoon,Table)
!
at(R1,TableLeft) at(R2,TableRight) at(Knife,ShelfA) at(Spoon,Table)
move(R2,ShelfA,7)
pickUp(R2,LeftArm,Knife,8)
move(R2,TableLeft,9)
placeOn(R2,LeftArm,Table,10)
at(Knife,Table) at(Spoon,Table)

Figure 5.3.7: The expected and the real executions of the replan P computed at time step $t = 7$, with respect to the revised diagnosis $X_7 = \{(R2, Base, 3)\}$.

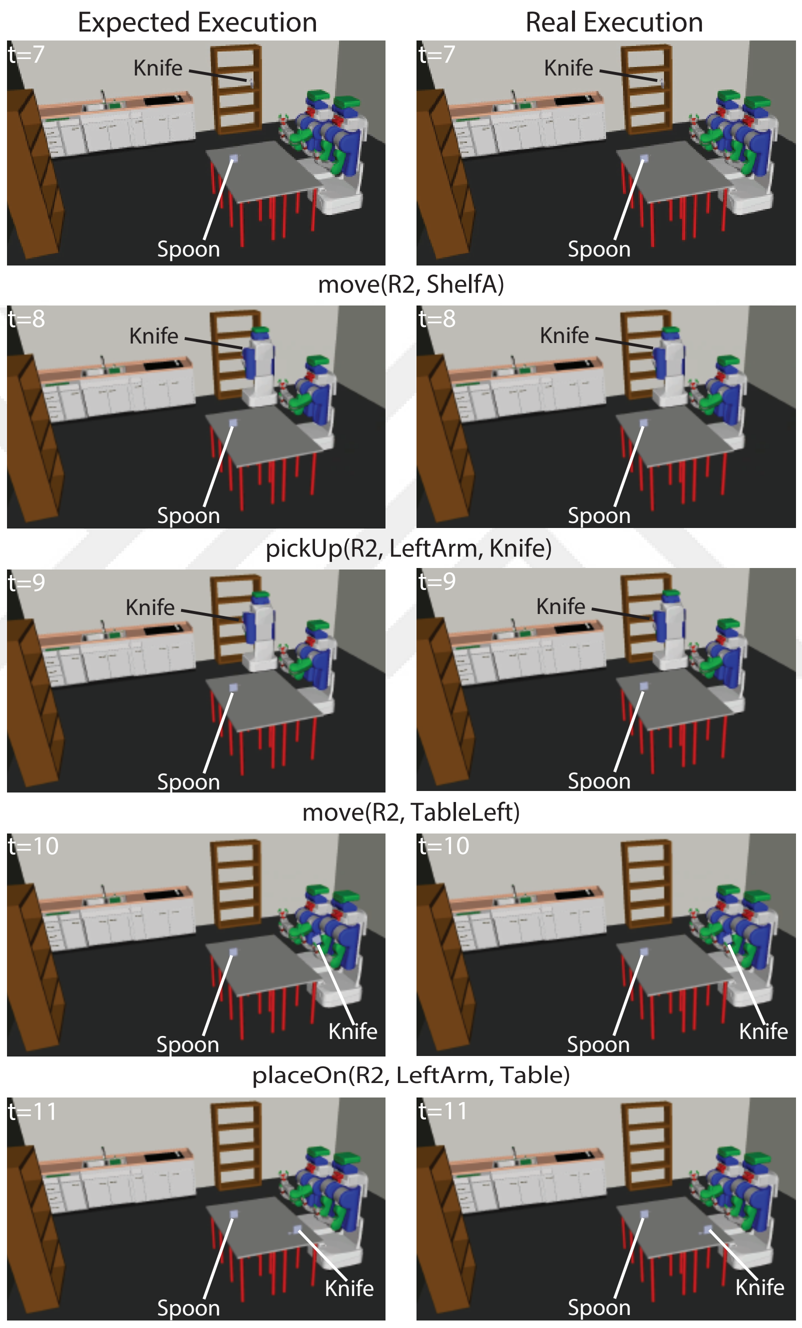


Figure 5.3.8: The dynamic simulations for the expected and the real executions of the replan presented in Figure 5.3.3.

Repairs: Sometimes the generated diagnoses prevent the algorithm find a replan, as we have observed with the augmented diagnosis computed for the relevant discrepancy detected at time step $t = 7$: since the bases of both robots are diagnosed as broken, the robots cannot move around, and thus replanning fails. In such cases, the algorithm may repair a minimum number of broken parts while replanning.

Suppose that the algorithm computes the following set of all min-cardinality diagnoses, and none of them leads to a replan:

$$\begin{aligned}&\{\{(\mathit{R1},\mathit{Base},1),(\mathit{R1},\mathit{LeftArm},3)\},\{(\mathit{R1},\mathit{Base},1),(\mathit{R2},\mathit{Base},3)\},\\ &\{(\mathit{R1},\mathit{LeftArm},3),(\mathit{R2},\mathit{RightArm},5)\},\{(\mathit{R1},\mathit{LeftArm},3),\\ &(\mathit{R2},\mathit{Base},3)\},\{(\mathit{R2},\mathit{RightArm},5),(\mathit{R1},\mathit{Base},1)\}\}.\end{aligned}$$

The algorithm may consult the user about the preference over repairing these parts. Suppose that the user knows that the parts of robot *R2* take usually more time to repair since it is an old model. Alternatively, the user may check all the computed diagnoses above and identify the parts that more often appear in diagnoses. Then this information (pref_x) can be utilized while replanning as explained in Section 4.4.2. For instance, the following replan is computed with repairs $\{(R1,\mathit{Base}),(R1,\mathit{LeftArm})\}$:

$$\begin{aligned}&P{=}\langle \mathit{move(R1,ShelfA,5),\ pickUp(R1,LeftArm,Knife,6),}\\ &\mathit{move(R1,TableLeft,7),\ placeOn(R1,LeftArm,Table,8)}\rangle.\end{aligned}$$

Since the parts *(R1,Base)* and *(R1,LeftArm)* are repaired, they are utilized in the new plan.

5.4 Experimental Evaluation

We have performed two groups of experiments. First, we have evaluated the computational performance of our execution monitoring algorithm (Algorithm 3) with different replanning approaches.

- *No guidance, full observability*: Whenever a relevant discrepancy is detected, the execution monitoring algorithm does not perform diagnostic reasoning, and replans from the current state under the full observability assumption, considering the low-level feasibility checks.
- *Guidance via diagnosis, partial observability*: Whenever a relevant discrepancy is detected, the execution monitoring algorithm computes a diagnosis for it (e.g., which broken components may cause this discrepancy). Then, the algorithm replans guided by this diagnosis. For instance, if a robotic component is diagnosed to be broken, then the algorithm does not allow the use of this broken part while replanning.

Second, we have experimented with three different approaches to diagnostic reasoning to better understand their usefulness within the execution monitoring algorithm.

- *Augmented diagnosis*: A diagnosis is generated about the broken parts based on the current observation, while imposing the validity of all of the previous hypotheses.
- *Reset diagnosis*: A diagnosis is generated about the broken parts based on the current observation, without considering any of the previous hypotheses.

- *Revised diagnosis*: A diagnosis is generated about the broken parts based on the current and the previous observations, while updating the previous hypothesis as necessary.

In these experiments, we have evaluated the validity of the following statements:

S1 Utilizing diagnostic reasoning in the execution monitoring algorithm increases the success of reaching the goal state while decreasing the number of replannings.

S2 Utilizing revised diagnosis (instead of reset or augmented diagnosis) in the execution monitoring algorithm increases the success of reaching the goal state while decreasing the number of replannings.

S3 The execution monitoring algorithm equipped with revised diagnosis (instead of reset or augmented diagnosis) is more scalable in terms of computation time as the size of a diagnosis increases.

5.4.1 Experimental Setup

We have considered the kitchen domain (Section 5.3.1) for our experimental evaluations. Recall that, in this domain, multiple bimanual mobile service robots collaboratively set-up a dinner table in a kitchen. The goal for the robots is to place all objects initially scattered around the kitchen on the table.

Each benchmark instance $I{=}\langle s_0, s_g, n, B\rangle$ is described by an initial state s_0, a goal state s_g, a makespan n (n <k), and a set B of pairs (b,i) of a number b of broken robotic parts and a time step i. Each pair describes how many parts get

broken and at what time step. No physical plan execution takes place during the evaluations. Instead, for each benchmark instance $I{=}\langle s_0, s_g, n, B\rangle$,

- an initial plan $P_I{=}\langle A_0, \ldots, A_{n-1}\rangle$ $(n < k)$ is computed to reach s_g from s_0 in k steps, and

- a set X_I of tuples (r, p, i) are randomly generated respecting B (i.e., for each (b, i) in B, the number of tuples (r, p, i) in X_I is exactly b) such that every component (r, p) plays a role in the actions A_i of the plan P (i.e., for every (r, p, i) in X_I, $disables(X, A_i, F)$ where $(r, p, i) \in X$).

Then, the "real world" states expected during the execution this plan P_I are computed by a dynamic simulation considering X_I. Along these lines, the observations over the monitored fluents are extracted from these "real world" states. For instance, Table 5.4.1 shows the instances that are used to compare different diagnosis approaches when two robotic components are broken. The first line indicates that in the "real world" dynamic simulation *rightArm* of *rob1* and *leftArm* of *rob2* are broken at time steps 4 and 2, respectively. The upper bound k on the total plan length is set to 60.

All experiments have been performed on a Windows server with six 3.5 GHz Intel® Xeon® E5-1650 v3 CPU cores and 8 GB memory. Reasoning problems have been solved using CLINGO 4.5.4. We have used multi-threading (limited to 4 threads) in these computations. We have considered a timeout of 100 seconds per planning instance, with the anytime option of CLINGO.

Table 5.4.1: 2 Broken Part Instance Set

Experiment Number	**Broken Part Set**	
1	broken(rob1,right,4)	broken(rob2,left,2)
2	broken(rob1,base,6)	broken(rob3,left,6)
3	broken(rob4,left,7)	broken(rob4,right,0)
4	broken(rob2,right,8)	broken(rob3,left,8)
5	broken(rob4,base,7)	broken(rob4,right,4)
6	broken(rob1,left,2)	broken(rob4,left,4)
7	broken(rob2,right,3)	broken(rob3,right,3)
8	broken(rob1,base,0)	broken(rob2,right,5)
9	broken(rob1,left,3)	broken(rob3,left,8)
10	broken(rob4,base,0)	broken(rob4,left,8)
11	broken(rob3,right,8)	broken(rob4,base,4)
12	broken(rob1,base,6)	broken(rob3,right,3)
13	broken(rob1,base,2)	broken(rob3,left,7)
14	broken(rob2,left,5)	broken(rob3,left,6)
15	broken(rob3,left,4)	broken(rob4,right,0)
16	broken(rob1,left,8)	broken(rob2,base,6)
17	broken(rob3,left,0)	broken(rob3,right,2)
18	broken(rob3,left,0)	broken(rob3,right,0)
19	broken(rob4,left,5)	broken(rob4,right,0)
20	broken(rob4,base,4)	broken(rob4,right,7)
21	broken(rob2,base,7)	broken(rob2,right,1)
22	broken(rob1,left,1)	broken(rob1,right,5)
23	broken(rob2,base,4)	broken(rob4,right,1)
24	broken(rob1,left,0)	broken(rob2,left,8)
25	broken(rob1,base,1)	broken(rob3,right,6)

5.4.2 Evaluation Criteria

We have evaluated statements S1, S2, and S3 from the following perspectives: success rate and computational efficiency.

For every instance $I=\langle s_0, s_g, n, B\rangle$ and an initial plan P_I, we say that I is *successful* if the goal state s_g is reached by execution monitoring of P_I by our algorithm without exceeding the total plan length k. Accordingly, the success rate is defined as follows:

$$\mathit{Success\ Rate} = \frac{\#\ \mathit{successful\ instances}}{\#\ \mathit{all\ instances}} \times 100.$$

For every instance $I=\langle s_0, s_g, n, B\rangle$, with an initial plan P_I and a set X_I of broken components, we quantify the computational efficiency of our algorithm by means of the following measures:

- *# replannings*: the number of replannings performed during execution monitoring,
- *total plan length*: the total length of the final plan,
- *diagnosis time*: CPU time in seconds, to perform all required diagnostic reasoning during execution monitoring,
- *replanning time*: CPU time in seconds, to perform all required replannings during execution monitoring, and
- *diagnosis accuracy*: the accuracy of the generated diagnosis X_t with respect to X_I.

Note that, during plan execution monitoring, replanning guided by a diagnosis may lead to the goal state even though the diagnosis is incorrect with respect to what is broken. However, proper identification of the broken robotic parts is desirable such that they can be repaired before the next plan execution. To evaluate the accuracy of the diagnoses generated by different diagnostic reasoning approaches, we have introduced a metric called *diagnosis accuracy*.

The diagnosis accuracy for *Revised diagnosis* is defined relative to the last revised diagnosis X_t made by the execution monitoring algorithm:

$$Accuracy_{Revised} = \frac{|X_t \bigcap X_I|}{max(|X_I|, |X_t|)} \times 100. \tag{5.2}$$

Here, the last diagnosis is considered since it utilizes all of the previous observations as well as the current observations. For instance, assume that

$$X_I = \{\textit{(R1,LeftArm,3),(R2,RightArm,5),(R3,Base,9)}\},$$
$$X_3 = \{\textit{(R1,LeftArm,3),(R1,RightArm,2),(R3,Base,9)}\}.$$

Then the diagnosis accuracy is 66.67%.

The diagnosis accuracy for *Augmented diagnosis* is defined in the same way:

$$Accuracy_{Augmented} = Accuracy_{Revised}.$$

The diagnosis accuracy for *Reset diagnosis* is defined relative to the all reset diagnoses X_i made by the execution monitoring algorithm:

$$Accuracy_{Reset} = \frac{|X_I \cap \bigcup X_i|}{max(|X_I|, |\bigcup X_i|)} \times 100 \tag{5.3}$$

Here, all reset diagnoses are considered since they are computed independent from each other. For instance, assume that

$$X_I = \{\textit{(R1,LeftArm,3),(R2,RightArm,5),(R3,Base,9)}\},$$
$$X_1 = \{\textit{(R1,LeftArm,3)}\},$$
$$X_2 = \{\textit{(R2,LeftArm,4)}\},$$
$$X_3 = \{\textit{(R1,LeftArm,3),(R2,RightArm,5),(R2,Base,8)}\}.$$

Then, since

$$\bigcup X_i = \{\textit{(R1,LeftArm,3),(R2,LeftArm,4),}$$
$$\textit{(R2,RightArm,5),(R2,Base,8)}\}$$

the diagnosis accuracy is 50%.

5.4.3 Evaluating Replanning Approaches

Whenever a relevant discrepancy is detected, replanning is required since the rest of the plan cannot reach the goal state. In general, there are two different approaches addressing replanning, with and without guidance. Guided replanning considers the generated explanation for the cause of the failure such that if some parts are diagnosed as broken, the newly generated plan does not rely on the components that are diagnosed as broken.

Algorithms that do not have such explanations for the cause of failure simply perform replanning from the current state, without considering the reason of the failure. Replanning to reach the goal state requires the full state information at the current time step. However, a typical replanning algorithm has no means to predict full current state under partial observability. On the other hand, our proposal to guided replanning with diagnosis generates the full state information by performing predictions considering the hypothetically broken parts as (Definition 4).

Therefore, for testing purposes the full state information is provided to the replanning method, whenever required. Note that guided replanning is not provided with the full state and relies on prediction to estimate current state.

To test the validity of S1, we have investigated the usefulness of two approaches to replanning, *No guidance with full observability* and *Guidance via diagnosis with partial observability*, by performing a set of experiments in the kitchen domain with 2 robots and 10 objects. We have generated 25 instances where 2 robotic parts are broken at different times. Similarly, another 25 instances have been generated where 3 robotic parts are broken. We have run our algorithm for each instance, as described in Section 5.4.1, and reported in Table 5.4.2 the average and the standard deviation (in parenthesis, over 25 instances) of the evaluation metrics described in Section 5.4.2.

Several interesting observations about these two replanning approaches can be made over these results:

- In terms of Success Rate, replanning guided by diagnosis outperforms replanning with no guidance.
- For each number of broken parts, the average number of replannings with guided replanning is much lower.
- The total plan length decreases when replanning is guided by diagnosis. This decrease becomes more significant as the number of broken parts increases.

These results are expected since replanning with no guidance does not consider causes of discrepancies and may try to use the broken robotic components in replans, resulting in new discrepancies. Therefore, with no guidance, the number of replannings and the total plan length increase with a large margin.

We need to emphasize that even though replanning with no guidance is provided with full observability while guided replanning is performed under partial observability, the guided replanning performs significantly better than replanning with no guidance in terms of the success rate, the number of replannings, and the total plan length.

These results also confirm our statement S1: diagnosis is useful for execution monitoring.

Table 5.4.2: Replanning with No Guidance (full observability) vs. Replanning Guided by Revised Diagnosis (partial observability)

	No Guidance			Guidance via Revised Diagnosis		
Number of broken parts	# Replannings	Total plan length	Success rate [%]	# Replannings	Total plan length	Success rate [%]
2	8.48 ± 2.29	26.44 ± 5.50	80	3.52 ± 0.92	17.76 ± 2.77	92
3	14.12 ± 3.37	35.96 ± 7.09	72	4.04 ± 1.30	22.48 ± 6.46	80

5.4.4 Evaluating Diagnostic Reasoning Approaches

To test the validity of S2 and S3, we have evaluated three diagnostic reasoning approaches, *Augmented diagnosis*, *Reset diagnosis* and *Revised diagnosis*, in terms of number of replannings, total plan length, and CPU time.

Scalability analysis with respect to the number of broken robot components

We have performed a set of experiments in the kitchen domain with 4 robots and 20 objects. We have generated 25 instances where 2 (resp. 3, 4, 5, 6) robotic parts are broken at different times. We have run our algorithm 3 times for each instance, as described in Section 5.4.1, and reported in Table 5.4.3 the average numbers (over 25 instances, 3 runs) for the evaluation metrics described in Section 5.4.2. The results of these experiments are also presented in Figure 5.4.1.

Figure 5.4.1 presents Success Rate of different diagnostic reasoning approaches, for each number of broken parts in terms of reaching the goal state. According to these results,

- Increasing the number of broken parts reduces Success Rate in general.
- The success Rate of *Augmented diagnosis* is quite low compared to the other two diagnostic reasoning approaches. For instance, with 6 broken parts, *Augmented diagnosis* can solve only 4% of the 25 instances, whereas *Reset diagnosis* can solve 20% of the instances and *Revised diagnosis* can solve almost 68% of the instances.

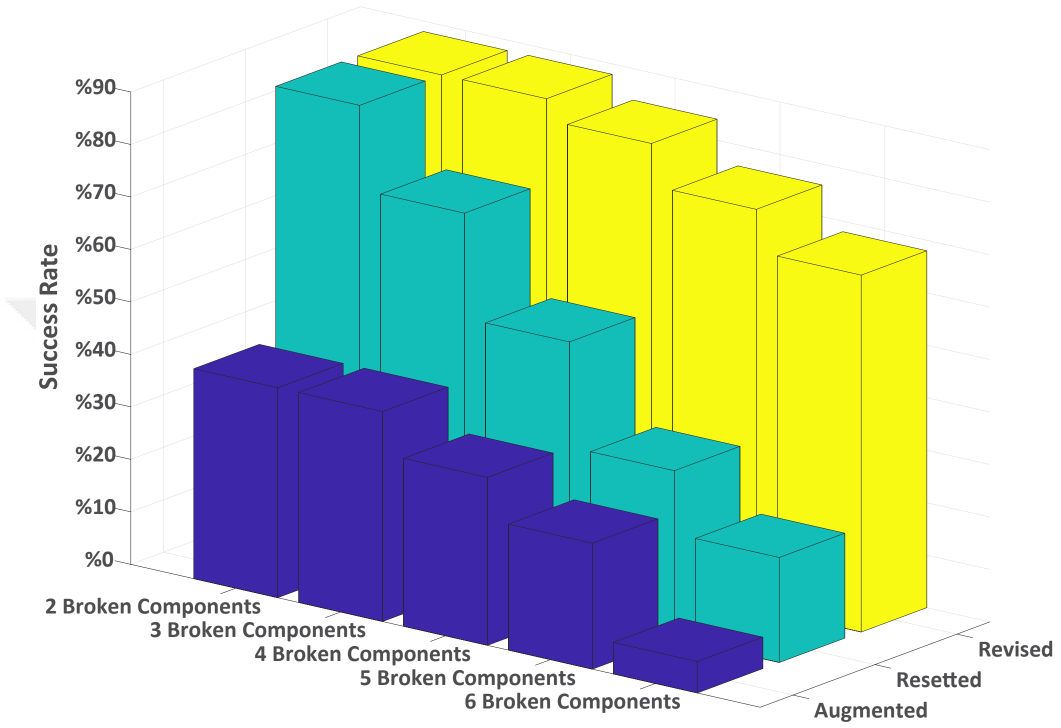


Figure 5.4.1: (Effect of number of broken parts on success rates of different diagnosis approaches.

The first observation above can be explained as follows: increasing the number of broken parts increases the number of possible most-probable min-cardinality diagnoses, and thus decreases the likelihood of selecting a correct diagnosis.

The second observation above can be explained as follows. When a relevant discrepancy is detected, *Augmented diagnosis* generates a hypothesis about the broken parts based on the current observations, while imposing the validity of the previous diagnoses even though they might not be correct. Therefore, an augmented diagnosis grows each time a relevant discrepancy is detected during plan execution monitoring: larger diagnosis puts more constraints on replanning, and thus reduces Success Rate. *Reset diagnosis* generates a hypothesis for a detected relevant discrepancy based on the current observations, but without relying on the previous assumptions or observations. Therefore, sometimes incorrect diagnoses can be made repeatedly, leading to incorrect replans; reducing Success Rate but not as much as *Augmented diagnosis*.

Revised diagnosis, on the other hand, considers the current observation and all of the previous observations such that the new hypothesis about the broken robot components is consistent with them, alleviating the disadvantages of the other diagnostic reasoning approaches.

Since the number of successful instances with *Augmented diagnosis* is very low, Table 5.4.3 compares *Reset diagnosis* and *Revised diagnosis* in terms of the number of replannings, the total plan length, and the diagnosis accuracy. The results can be summarized as follows:

- With *Revised diagnosis*, generally, the average number of replannings performed by the algorithm is significantly lower (e.g., 40% lower for instances with 5 broken parts).

- The scalability of the algorithm in terms of diagnosis time is better with *Revised diagnosis* (e.g., 30% lower for instances with 5 broken parts).

- The diagnosis accuracy is always higher for *Revised diagnosis* (e.g., 25% higher for instances with 2 broken parts).

Table 5.4.3 indicates that *Reset diagnosis* outperforms *Revised diagnosis*, when the number of broken parts is 6. However, this result might be misleading since the number of common successful instances is only 4. Therefore, in Table 5.4.4 we present the results for all instances even if the runs could not reach the goal state. The results confirm the conclusions from Table 5.4.3

- *Revised Diagnosis* can reduce the number of replannings up to 60% with respect to *Reset Diagnosis*.
- *Revised Diagnosis* can save up to 75% CPU time with respect to *Reset Diagnosis*.
- The diagnosis accuracy is higher for *Revised Diagnosis* (up to 40%) with respect to *Reset Diagnosis*.

Table 5.4.3: Scalability analysis with different number of broken parts (commonly solved instances): *Revised* vs. *Reset*

	2 Broken Parts		3 Broken Parts		4 Broken Parts		5 Broken Parts		6 Broken Parts	
	Reset	Revised	Reset	Revised	Reset	Revised	Reset	Revised	Reset	Revised
# Replannings	3.52 (1.47)	**2.76** (0.94)	4.75 (2.23)	**3.75** (1.12)	6.84 (1.77)	**5.69** (1.97)	9.28 (2.42)	**5.71** (1.79)	**6.75** (3.30)	9.75 (0.50)
Total plan length	26.09 (4.88)	**24.85** (4.75)	29.87 (7.54)	**27.75** (4.53)	39.23 (5.54)	**36.76** (6.52)	40.57 (6.60)	**36.85** (6.54)	**41.00** (8.36)	51.50 (7.32)
Diagnosis time [s]	7.65 (5.23)	**4.13** (1.54)	11.86 (9.75)	**7.78** (3.16)	20.38 (7.91)	**13.61** (4.14)	30.64 (12.76)	**21.72** (6.98)	**21.34** (15.52)	40.76 (9.55)
Replanning time [s]	67.72 (22.77)	**64.94** (28.14)	91.83 (28.49)	**81.04** (24.20)	128.71 (35.13)	**118.91** (48.53)	133.60 (31.87)	**104.69** (16.92)	**139.86** (62.35)	169.18 (54.19)
Diagnosis accuracy [%]	65.39 (18.88)	**90.40** (20.11)	66.66 (22.92)	**89.56** (15.93)	59.94 (13.02)	**86.53** (19.40)	63.57 (7.70)	**82.85** (17.99)	59.71 (20.97)	**74.95** (9.62)
# Successful instances	22	22	18	**22**	13	**21**	8	**19**	5	**17**
# Common successful instances	21		16		13		7		4	

Table 5.4.4: Scalability analysis with different number of broken parts (all instances): *Revised* vs. *Reset*

	2 Broken Parts		3 Broken Parts		4 Broken Parts		5 Broken Parts		6 Broken Parts	
	Reset	Revised	Reset	Revised	Reset	Revised	Reset	Revised	Reset	Revised
# Replannings	4.32 (3.86)	**2.80** (0.91)	8.72 (6.63)	**3.84** (1.14)	8.96 (4.65)	**5.60** (1.89)	14.20 (10.06)	**5.88** (2.55)	16.20 (7.87)	**6.96** (2.54)
Total plan length	27.88 (8.31)	**25.32** (4.62)	39.76 (14.84)	**29.00** (6.19)	42.24 (13.33)	**36.00** (9.63)	45.20 (15.89)	**39.28** (10.20)	53.08 (11.47)	**43.52** (8.84)
Diagnosis time [s]	12.16 (20.69)	**4.41** (1.99)	35.42 (39.20)	**7.67** (3.13)	43.33 (32.54)	**19.59** (16.95)	83.55 (76.47)	**19.73** (11.09)	116.94 (72.97)	**30.16** (16.37)
Replanning time [s]	68.14 (26.28)	**63.37** (27.91)	102.53 (34.42)	**86.59** (28.20)	137.01 (46.98)	**125.11** (48.16)	135.88 (79.00)	**117.74** (46.42)	154.78 (66.26)	**154.77** (50.21)
Diagnosis accuracy [%]	58.93 (25.43)	**82.00** (31.88)	55.92 (27.43)	**78.65** (34.53)	45.65 (22.18)	**76.00** (31.02)	25.44 (32.10)	**66.40** (40.71)	23.00 (28.84)	**54.66** (40.40)

Total plan length and diagnosis accuracy have been analyzed by two-way ANOVA (between-subjects) to determine any significant differences. Two-way ANOVA is applied as while the broken part sets are matched, the total execution of each scenario differs in each experiment. Levene's test of homogeneity of variances is implemented for two independent variables (i.e., diagnosis type and number of broken parts) to validate sphericity assumption of two-way ANOVA. For both total plan length and diagnosis accuracy sphericity assumption has not been violated.

The results of the two-way ANOVA for *total plan length* are presented in Table 5.4.5. These results indicate that different *diagnosis types* are not significantly different from each other ($F(1,112) = 0.023, p = 0.879$ and $\eta^2 = 0$) while different *broken part numbers* are significantly different from each other ($F(4,112) = 38.455, p < 0.001$ and $\eta^2 = 0.579$).

Table 5.4.5: Tests of between-subjects effects for dependent variable *total plan length*

Source	Type III Sum of Squares	df	Mean Square	F	Sig.	Partial Eta Squared
Diagnosis Type	0.796	1	0.796	0.023	0.879	0.000
Broken Part Number	5303.111	4	1325.778	38.455	<0.001	0.579
Diagnosis Type*Broken Part Number	313.046	4	78.262	2.270	0.066	0.075
Error	3861.318	112	34.476			

We have performed pairwise comparison of different number of broken parts utilizing Tukey HSP post-hoc test. The result of this test have shown that each different number of broken part is significantly different from the others.

The results of the two-way ANOVA for *diagnosis accuracy* are presented in Table 5.4.6. These results indicate that different *diagnosis types* are significantly different from each other ($F(1,112) = 30.513, p < 0.001$ and $\eta^2 = 0.214$)

while different *broken part numbers* are not significantly different from each other ($F(4,112) = 0.931, p = 0.931$ and $\eta^2 = 0.032$).

Table 5.4.6: Tests of between-subjects effects for dependent variable *diagnosis accuracy*

Source	Type III Sum of Squares	df	Mean Square	F	Sig.	Partial Eta Squared
Diagnosis Type	1.018	1	1.018	30.513	<0.001	0.214
Broken Part Number	0.124	4	0.931	0.931	0.449	0.032
Diagnosis Type * Broken Part Number	0.027	4	0.204	0.204	0.935	0.007
Error	3.737	112	0.033			

Figure 5.4.2 presents the CPU time for the diagnostic reasoning of *Revised diagnosis*. As expected,

- Increasing the number of broken parts exponentially increases the diagnosis time.

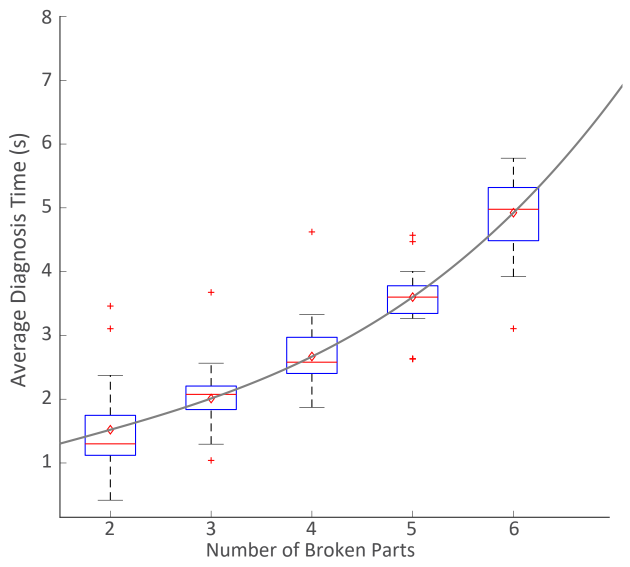


Figure 5.4.2: Average CPU time consumption for different number of broken parts during revised diagnosis.

Scalability analysis with respect to the number of objects in the environment

We have performed a set of experiments in the kitchen domain with 4 robots, and 15, 20, 25 objects. We have generated 25 instances where 4 robotic parts are broken at different times. We have run our algorithm 3 times for each instance, as described in Section 5.4.1, and reported in Table 5.4.7 the average numbers (over 25 instances, 3 runs) for the evaluation metrics described in Section 5.4.2. The results of these experiments are also presented in Figure 5.4.3.

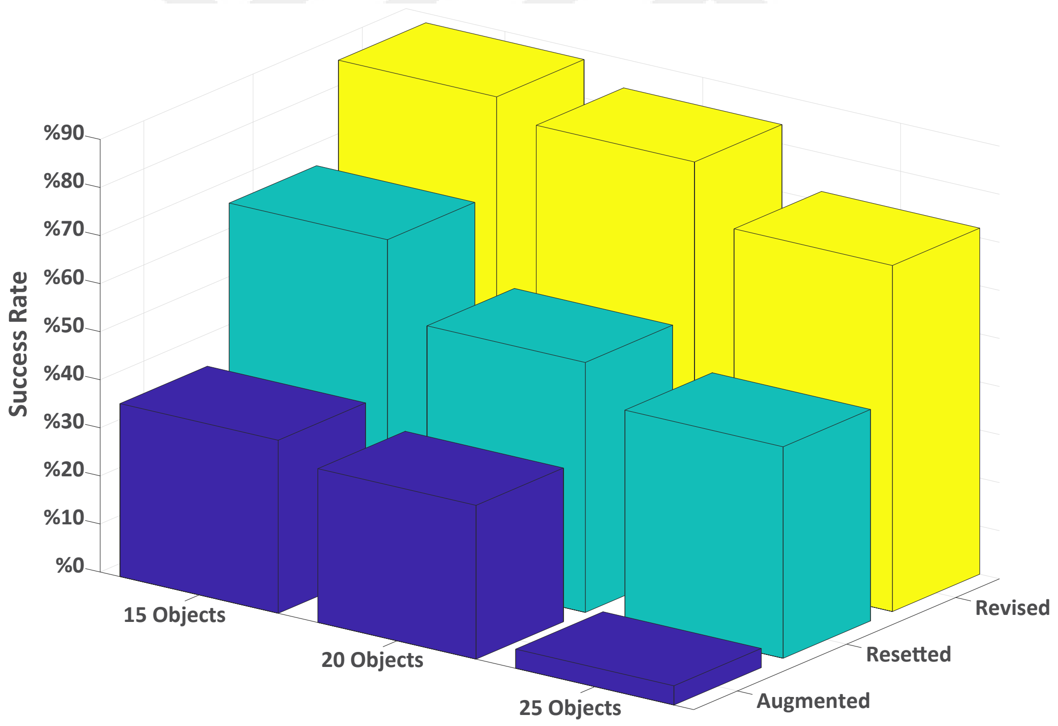


Figure 5.4.3: Effect of number of objects on success rate of different diagnosis approaches.

Figure 5.4.3 presents Success Rates of different diagnostic reasoning approaches. According to these results:

- The execution monitoring algorithm with *Revised Diagnosis* outperforms all other approaches by large margin in terms of Success Rate.

Table 5.4.7 compares *Reset diagnosis* and *Revised diagnosis* in terms of the number of replannings, the total plan length, and the diagnosis accuracy. The results can be summarized as follows:

- With *Revised diagnosis*, generally, the average number of replannings performed by the algorithm is significantly lower (e.g., 30% lower for instances with 25 objects).
- The scalability of the algorithm in terms of diagnosis time is better with *Revised diagnosis* (e.g., 55% lower for instances with 25 objects).
- The diagnosis accuracy is always higher for *Revised diagnosis* (e.g., 30% higher for instances with 25 objects).

Table 5.4.7: Comparison of Revised and Reset Approaches for Different Number of Objects (commonly solved instances)

	15 Objects		20 Objects		25 Objects	
	Reset	Revised	Reset	Revised	Reset	Revised
# Replannings	5.05 (2.10)	**4.29** (1.26)	6.84 (1.77)	**5.69** (1.97)	8.77 (3.19)	**5.88** (1.83)
Total plan length	30.05 (5.92)	**28.58** (5.55)	39.23 (5.54)	**36.76** (6.52)	43.88 (2.84)	**36.55** (6.00)
Diagnosis time [s]	9.51 (6.27)	**6.65** (2.92)	20.38 (7.91)	**13.61** (4.14)	43.51 (14.48)	**19.69** (9.22)
Replanning time [s]	**73.26** (33.73)	85.38 (30.88)	128.71 (35.13)	**118.91** (48.53)	237.95 (67.30)	**219.99** (33.79)
Diagnosis accuracy [%]	65.94 (17.50)	**85.29** (12.68)	59.94 (13.02)	**86.53** (19.40)	52.32 (7.33)	**83.33** (17.67)
# Successful instances	17	**22**	13	**21**	11	**18**
# Common successful instances	17		13		9	

Replanning number and diagnosis accuracy have been analyzed by two-way ANOVA (between-subjects) to determine any significant differences. Two-way ANOVA is applied as while the broken part sets are matched, the total execution of each scenario differs in each experiment. Levene's test of homogeneity of variances is implemented for two independent variables (i.e., diagnosis type and number of objects) to validate sphericity assumption of two-way ANOVA. For both replanning number and diagnosis accuracy sphericity assumption has not been violated.

The results of the two-way ANOVA for *replanning* are presented in Table 5.4.8. These results indicate that different *diagnosis types* ($F(1,72) = 11.647, p = 0.001$ and $\eta^2 = 0.139$) and *object numbers* ($F(2,72) = 11.312, p < 0.001$ and $\eta^2 = 0.239$) are significantly different from each other.

Table 5.4.8: Tests of between-subjects effects for dependent variable *replanning number*

Source	Type III Sum of Squares	df	Mean Square	F	Sig.	Partial Eta Squared
Diagnosis Type	46.811	1	46.811	11.647	<0.001	0.139
Object Number	90.931	2	45.465	11.312	0.001	0.239
Diagnosis Type * Object Number	13.795	2	6.898	1.716	<0.001	0.046
Error	289.377	72	4.019			

This results also revealed that there is a statistically significant interaction between the effects of *diagnosis type* and *object number* ($F(2,72) = 1.716, p < 0.001$ and $\eta^2 = 0.046$).

Tukey HSD post-hoc comparison of different objects numbers is presented in Table 5.4.9 which indicates that there are significant differences between $15-20$ objects and $15-25$ objects.

Table 5.4.9: Pairwise comparison of different object numbers for dependent variable of replanning number wrt Tukey

(Object Number	Object Number	Mean Difference	Std. Error	Sig.	95% Confidence Interval	
					Lower Bound	Upper Bound
15	20	-1.5928*	0.522	0.009	-2.84	-0.34
	25	-2.6569*	0.584	<0.001	-4.05	-1.25
20	15	1.5928*	0.522	0.009	0.34	2.84
	25	-1.0641	0.614	0.201	-2.55	0.40
25	15	2.6569*	0.584	<0.001	1.25	4.05
	20	1.0641	0.614	0.201	-0.40	2.53

The results of the two-way ANOVA for *diagnosis accuracy* are presented in Table 5.4.10. These results indicate that different *diagnosis types* are significantly different from each other ($F(1,72) = 50.831, p < 0.001$ and $\eta^2 = 0.414$) while different *object numbers* are not significantly different from each other ($F(2,72) = 1.523, p = 0.225$ and $\eta^2 = 0.041$).

Table 5.4.10: Tests of between-subjects effects for dependent variable *diagnosis accuracy*

Source	Type III Sum of Squares	df	Mean Square	F	Sig.	Partial Eta Squared
Diagnosis Type	1.198	1	1.198	50.831	<0.001	0.414
Object Number	0.072	2	0.036	1.523	0.225	0.041
Diagnosis Type * Object Number	0.045	2	0.022	0.0945	0.394	0.026
Error	1.696	72	0.024			

Chapter 6

Systematic Evaluation of Execution Monitoring Algorithms

Before execution monitoring algorithms are deployed on autonomous systems, comprehensive testing and simulation is needed to evaluate their performance and to understand their applicability. In general, three major components are required to implement a generic testing framework for execution monitoring: generation of discrepancies between the expected states and the observed states of the world, identification of the causes of relevant discrepancies, and replanning to reach the goals.

Major sources of discrepancies during plan execution can be loosely categorized as unexpected exogenous events, changes in the goals, or failure of robot parts. In general, unexpected exogenous events and changes in the goals may be detected directly by sensors used to monitor the world states, while determination of the broken parts that result in a discrepancy typically requires deeper reasoning.

With this motivation, we introduce a formal method for relevant discrepancy generation with respect to the plan being executed to evaluate performance of an execution monitoring algorithm. Discrepancies are introduced dynamically during the execution of a plan whenever execution monitoring algorithm successfully identifies the reasons of the observed discrepancies. Our formal method for discrepancy generation is designed to work under partial observability since it only generates discrepancies in terms of the monitored fluents.

6.1 The Discrepancy Generation Problem

We formally define the problem of generating a relevant discrepancy that might have occurred due to a failure of a robotic component, after execution of some part of a plan.

Definition 10. *A discrepancy generation problem, DGP, is characterized by a tuple* $\langle \mathscr{D}^t_{diag}, s_0, P_{<t}, \mathscr{R}, X^{<t}_{real}, \sum X^{<t}_{real}, s_e, O^M_t, N \rangle$ *where*

- $\mathscr{D}^t_{diag}$ *is the diagnosis domain description in ASP,*
- s_0 *is an initial state,*
- $P_{<t}{=}\langle A_0, \dots, A_{t-1} \rangle$ *is the sequence of actions executed at the initial state* s_0 *until time step t,*
- $\mathscr{R}$ *is the set of pairs of all robots and their components that may get broken,*
- $X^{<t}_{real}$ *is a set of triples* $\langle r, p, i \rangle$ *that describes the parts p of robots r* $((r, p) \in \mathscr{R})$ *broken at an earlier time step i* $(i{<}t)$*,*
- $\sum X^{<t}_{real}$ *the set of all sets* $X^{<i}_{real}$ *of broken parts from previous executions of the plan* $(i{<}t)$*,*
- s_e *is the expected state at time step t,*

- *O^M_t is a set of all previous and current observed states o^M_i of monitored fluents in M $(i \leq t)$,*

- *N is a positive integer.*

A solution of DGP is a set X_{new} of (r,p,j) that describe robotic components (r,p) broken at time step j $(j \leq t)$, such that $|X^{<t}_{real}| + |X_{new}| \leq N$ and that $\mathscr{D}^t_{diag}$ when combined with the following rules has an answer set:

$$
\begin{array}{ll}
\leftarrow not\ F_{s_0}(0). & \\
\leftarrow not\ E_{A_i}(i) \quad (0 \leq i \leq t-1). & \\
\leftarrow F_{s_e}(t). & \\
\leftarrow not\ F_{o^M_i}(i) \quad (o^M_i \in O^M_t, i \leq t). & \qquad (6.1) \\
\leftarrow not\ broken(r,p,i) \quad \big((r,p,i) \in (X^{<t}_{real} \cup X_{new})\big). & \\
\leftarrow not\ \neg broken(r,p,l) \quad \big((r,p) \in \mathscr{R}, l \leq t, (r,p,l) \notin (X^{<t}_{real} \cup X_{new})\big). & \\
\leftarrow \textstyle{,}_{(r,p,i) \in X^{<i}_{real}} broken(r,p,i) \quad \big(X^{<i}_{real} \in \sum X^{<t}_{real}\big). &
\end{array}
$$

The first constraint in (6.1) ensures that the simulation starts at the given initial state. The second constraint ensures that only the actions $\langle A_0, A_1 \ldots, A_{t-1} \rangle$ of the plan are executed. The third constraint ensures that the expected state is not observed. The fourth constraint takes into account the earlier and current observations. The fifth and sixth constraints ensures that only the parts of the robots in $X^{<t}_{real}$ and X_{new} were broken at specified times. The seventh constraint ensures that introduced broken parts (X_{new}) are different from the previous executions ($\sum X^{<t}_{real}$).

6.2 The Simulation Algorithm

To evaluate the performance of an execution monitoring algorithm, possibly equipped with a diagnostic reasoning module that identifies discrepancies in terms of broken robotic components, a reasonable number of relevant scenarios should be tested. We propose a novel generic algorithm for systematic testing of such execution monitoring algorithms in simulation by dynamically generating discrepancies.

Our algorithm (Algorithm 4) takes as input the following:

- $\mathscr{D}^k$ is a planning domain description in ASP,
- s_0 is an initial state,
- s_g is a goal state,
- $\mathscr{R}$ is a set of pairs of robots and their components that may get broken,
- *disables* is a relation that describes which effects of actions are affected if some robots and their components in $\mathscr{R}$ are broken,
- M is a set of monitored fluents,
- P is a plan,
- T is a diagnosis type: Revised, Reset, Augmented, or None,
- k is an upper bound on the total makespan,
- N is an upper bound on the number of broken parts.

Algorithm 4 returns all possible scenarios (i.e., execution histories of plans) of length less than or equal to k, with at most N broken parts. It also returns the following:

- *#goalReached*: the number of scenarios that reach the goal.
- *#totalReplannings*: the total number of replannings for scenarios that reach the goal.
- *totalPlanLength*: the sum of final plan lengths for all scenarios that reach the goal.

According to Algorithm 4, initially the set X_{hyp} of "believed-to-be-broken" robotic parts, the set $X_{real}^{<t}$ of "really-broken" robotic parts, and the set O_t^M of observations are empty (line 5). Algorithm 4 generates a full expected state s_e (line 10) by solving a *prediction problem*. Then Algorithm 4 introduces broken robotic parts into $X_{real}^{<t}$, if possible, by solving a *discrepancy generation problem* (lines 11-13). Then $X_{real}^{<t}$ is used to generate an observed state at time step t (line 14). The expected state s_e and the observed state s_o are compared to each other in terms of the monitored fluents in M, to detect any discrepancies (line 17). If the detected discrepancy is relevant (line 20) then a diagnosis X_{hyp} is generated with respect to a diagnosis type T (line 22). Then, Algoirthm 4 finds a new plan, considering the diagnosis X_{hyp}, from the predicted state s_t, if possible, and continues the execution of this plan.

When plan execution finishes, if $X_{real}^{<t}$ is not empty, then $X_{real}^{<t}$ is added into $\sum X_{real}^{<t}$ (line 32) to generate a different scenario with broken parts different from every $X_{real}^{<i}$ in $\sum X_{real}^{<t}$.

The goal state is compared with the last predicted expected state to check whether the goal is reached or not (line 33). If the goal is reached, the relevant metrics are updated (lines 34 and 35). If $X_{real}^{<t}$ is empty at the end of the execution, then it means that all possible scenarios have been simulated (line 37).

Algorithm 4 Simulation Algorithm

```
Input: $\mathscr{D}^k$, $s_0$, $s_g$, $\mathscr{R}$, $disables$, $M$, $k$, $P$, $T$, $N$.
Output: All possible $ExecutionHistories$ , $\#executions$, $\#goalReached$,
    $\#totalReplannings$.
 1: $Execute = True$;
 2: $\#goalReached = 0$, $\#totalReplannings = 0$, $\#executions = 0$.
 3: while $Execute$ do
 4:     $t = 0$;
    //Initially, the sets $X_{hyp}$, $X_{real}^{<t}$ and $O_t^M$ are empty.
 5:     $X_{hyp} = \emptyset$; $X_{real}^{<t} = \emptyset$; $O_t^M = \emptyset$;
 6:     $\#replannings = 0$;
 7:     while $t < k$ and $P$ is not null do
 8:         Execute $A_t$;
 9:         $t \mathrel{+}= 1$;
    //Predict a full expected state $s_e$ at step $t$ considering $X_{hyp}$ updated before $t-1$.
10:         $s_e \leftarrow Predict(\mathscr{D}_{diag}^t, s_0, P_{<t}, O_t^M, X_{hyp})$;
11:         $X_{new} \leftarrow GenerateDiscrepancy(\mathscr{D}_{diag}^t, s_0, P_{<t}, \mathscr{R}, X_{real}^{<t}, \sum X_{real}^{<t}, s_e, O_t^M, N)$;
12:         if $X_{new}$ is not $null$ then
13:             $X_{real}^{<t} \leftarrow X_{new} \cup X_{real}^{<t}$;
    //Predict a full observed state $s_o$ at step $t$ considering $X_{real}^{<t}$.
14:         $s_o \leftarrow Predict(\mathscr{D}_{diag}^t, s_0, P_{<t}, O_t^M, X_{real}^{<t})$;
    // Extract the monitored fluents from full observed state.
15:         $o_t^M \leftarrow Extract(s_o, M)$
16:         $O_t^M \leftarrow$ Add $o_t^M$ into $O_t^M$;
    // Check whether there is a discrepancy between the expected state $s_e$ and the current
    observations $o_t^M$
17:         $discrepancy \leftarrow Discrepancy(\mathscr{D}_{state}, s_e, o_t^M)$;
18:         if $discrepancy$ then
19:             $s_v = ValidState(\mathscr{D}_{state}, s_e, o_t^M)$;
    //If there is a discrepancy, check whether it is relevant to the rest of the plan.
20:             $relevant = RelevancyCheck(\mathscr{D}_{diag}^t, s_v, s_g, P_{>t}, X_{hyp})$;
21:             if $relevant$ then
    //Find the cause of the detected discrepancy and update $X_{hyp}$ with possible broken
    robotic components.
22:                 $X_{hyp} \leftarrow Diagnose(\mathscr{D}_{diag}^t, \mathscr{R}, s_0, P_{<t}, O_t^M, X_{hyp}, T)$;
    //Replan from the expected state $s_t$ at step $t$, considering the current diagnosis $X_{hyp}$.
23:                 $s_t \leftarrow Predict(\mathscr{D}_{diag}^t, s_0, P_{<t}, O_t^M, X_{hyp})$;
24:                 $P_{new} \leftarrow RePlan(\mathscr{D}_R^{k-t}, s_t, s_g, X_{hyp}, k-t)$;
```

```
25:             $\#replannings \mathrel{+}= 1$;
26:             if $P_{new}$ is not $null$ then
27:                 $P \leftarrow$ Append $P_{new}$ to the end of $\langle A_0, ..., A_{t-1} \rangle$;
28:             else
29:                 $P \leftarrow$ null
30:     if $X_{real} \neq \emptyset$ then
31:         $\#executions \mathrel{+}= 1$;
    //Add $X^{<t}_{real}$ into $\sum X^{<t}_{real}$ such that algorithm generates different scenarios in the later executions
32:         $\sum X^{<t}_{real}.Add(X^{<t}_{real})$;
    //Compare the final expected state with the goal state to check whether the goal is reached or not.
33:         if $s_e == s_g$ then
34:             $\#goalReached \mathrel{+}= 1$;
35:             $\#totalReplannings \mathrel{+}= \#replannings$;
36:     else
    //If a discrepancy cannot be generated, terminate simulation.
37:         $Execute = False$;
38: return $ExecutionHistories$, $\#executions$, $\#goalReached$, $\#totalReplannings$
```

Figure 6.2.1 illustrates how Algorithm 4 generates scenarios. Consider a successful execution of a plan $P1 = \langle A1_0, A1_1, \ldots, A1_n \rangle$ highlighted in blue: $\langle S_0, A1_0, S1, A1_1, \ldots, A1_9, S_{10} \rangle$. At time step 1, suppose that Algorithm 4 introduces the robotic component $(R1, P1)$ as broken: $X_{new} = X^{<t}_{real} = \{(R1, P1)\}$. As a result, the plan execution ends up at state S'_2 (highlighted red). After a diagnosis of this failure, suppose that Algorithm 4 can not find a feasible plan, and thus the execution terminates. Note that $X^{<t}_{real} = \{(R1, P1)\}$ is added into $\sum X^{<t}_{real}$: $\sum X^{<t}_{real} = \{\{(R1, P1)\}\}$.

In the next iteration of Algorithm 4, another robotic component $(R2, P2)$ is introduced as broken at time step 3: $X_{new} = X^{<t}_{real} = \{(R2, P2)\}$. Suppose that a discrepancy is observed at time step 5 due to this broken part. The execution monitoring algorithm successfully identifies the reason of this discrepancy and performs replanning to generate a plan $P2 = \langle A2_0, A2_1, \ldots, A2_m \rangle$ that reaches the

goal state (highlighted orange). Then $X_{real}^{<t} = \{(R2,P2)\}$ is added into $\sum X_{real}^{<t}$: $\sum X_{real}^{<t} = \{\{(R1,P1)\},\{(R2,P2)\}\}$.

In the next iteration while Algorithm 4 is executing $P2$, Algorithm 4 introduces a new component $(R3,P3)$ as broken at time step 6: $X_{new} = \{(R3,P3)\}$. Then $X_{real}^{<t} = \{(R2,P2),(R3,P3)\}$. As a result, the plan execution ends up at the state S_7'', where a discrepancy is observed. Suppose that the execution monitoring algorithm can not generate a new plan and thus the execution terminates. Then $\sum X_{real}^{<t} = \{\{(R1,P1)\},\{(R2,P2)\},\{(R2,P2),(R3,P3)\}\}$.

In the next iteration while Algorithm 4 is executing $P2$, Algorithm 4 introduces a new component $(R4,P4)$ as broken at time step 8: $X_{new}^{<t} = \{(R4,P4)\})$. Then $X_{real}^{<t} = \{(R2,P2),(R4,P4)\}$. Suppose that the execution monitoring algorithm detects a discrepancy at time step 10 and generates a new plan which reaches the goal state S_{10}'' (highlighted purple). Then $\sum X_{real}^{<t} = \{\{(R1,P1)\},\{(R2,P2)\},\{(R2,P2),(R3,P3)\},\{(R2,P2),(R4,P4)\}\}$.

In the rest of the simulation, Algorithm 4 introduces different robotic parts as broken to generate different scenarios

We have implemented Algorithm 4 in Python to report also the following metrics:

- *totalReplanningTime*: total CPU time spent during replanning for scenarios that reach the goal.
- *totalDiagnosisTime*: total CPU time spend during diagnosis for scenarios that reach the goal.

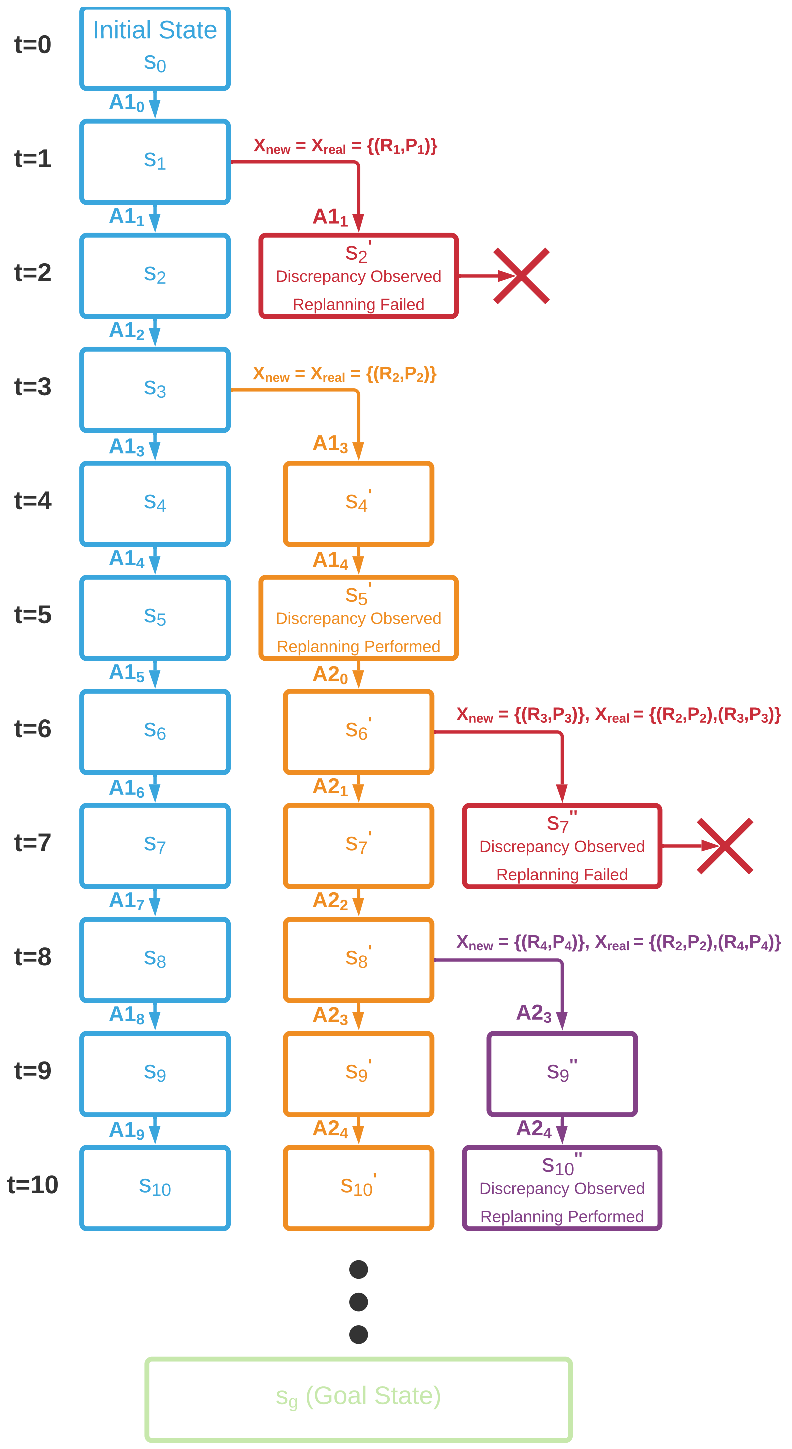


Figure 6.2.1: Possible scenarios generated by Algorithm 4.

6.3 Evaluation Criteria

We say that an execution of a plan P is successful if the goal state s_g is reached from the initial state s_0 by Algorithm 4, which simulates an execution monitoring algorithm that utilizes a diagnosis of type T, without exceeding the maximum makespan k. Accordingly, the success rate is defined as follows:

$$\mathit{success(P,T)} = \frac{\mathit{\#goalReached(P,T)}}{\mathit{\#generatedScenarios(P,T)}} \times 100$$

where *#goalReached* is defined as the number of scenarios that reaches the goal and *#generatedScenarios* represents the number of different scenarios generated by Algorithm 4.

We quantify the efficiency of an execution monitoring algorithm (simulated by Algorithm 4) by means of the average number of replannings, the average plan length, the CPU time spent for replannings and diagnoses.

$$\mathit{\#averageReplannings} = \frac{\mathit{\#totalReplannings(P,T)}}{\mathit{\#goalReached(P,T)}}$$

$$\mathit{averagePlanLength} = \frac{\mathit{totalPlanLength(P,T)}}{\mathit{\#goalReached(P,T)}}$$

$$\mathit{averageReplanningTime} = \frac{\mathit{totalReplanningTime(P,T)}}{\mathit{\#goalReached(P,T)}}$$

$$\mathit{averageDiagnosisTime} = \frac{\mathit{totalDiagnosisTime(P,T)}}{\mathit{\#goalReached(P,T)}}$$

6.4 Case Studies: Collaborative Service Robotics Domain

Let us demonstrate applications of Algorithm 4 with a service robotics setting, the kitchen domain, described in Section 5.3.1.

6.4.1 Case Study 1: Discrepancy generation without replanning

Suppose that no replanning algorithm is implemented within the execution monitoring. That is, the execution of the initial plan continues even after a discrepancy is detected.

Consider the kitchen domain with two mobile manipulators, two distinct objects and a table as shown in Figure 6.4.1. Assume that there is partial observability of the domain, such that only the objects located on the table and the robots around the table can be observed. Suppose that initially, the table top is empty, while the blue robot $R1$ and the yellow robot $R2$ hold a knife and a fork at their right hands, respectively. The goal is to have the knife at the yellow robot's left hand and the fork on the table.

Assume that for this planning problem instance, the following plan is calculated:

$$P1 = \langle \textit{move(R1,TableLeft,0),placeOn(R1,RightArm,Table,1),move(R2,TableLeft,2),} \\ \textit{move(R1,Corner,2),pickUp(R2,Knife,LeftArm,3),placeOn(R2,RightArm,Table,4)} \rangle.$$

The left column of Figure 6.4.1 presents the world states during the execution of this plan through a dynamic simulation, for which all feasibility checks are implemented.

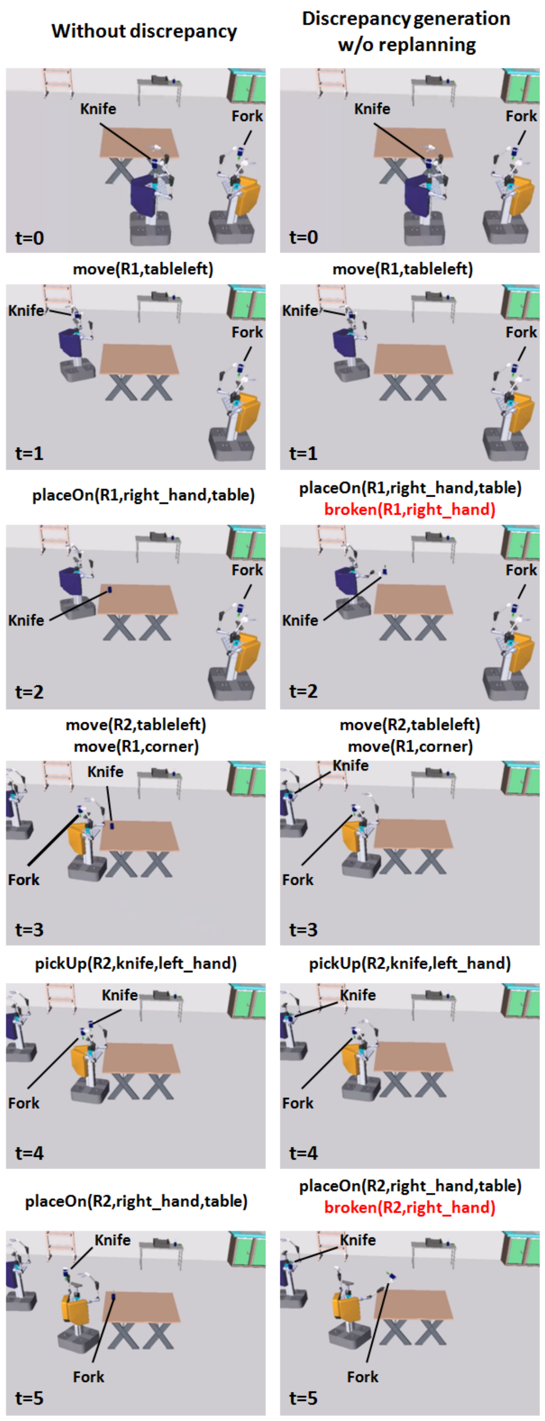

Without discrepancy
Discrepancy generation w/o replanning
Knife
Fork
t=0
move(R1,tableleft)
t=1
placeOn(R1,right_hand,table)
placeOn(R1,right_hand,table)
broken(R1,right_hand)
t=2
move(R2,tableleft)
move(R1,corner)
t=3
pickUp(R2,knife,left_hand)
t=4
placeOn(R2,right_hand,table)
placeOn(R2,right_hand,table)
broken(R2,right_hand)
t=5

Figure 6.4.1: Left: World states at each time step. Right: Observed states generated at each time step for Case Study 1.

The right column of Figure 6.4.1 presents observed states generated by Algorithm 4. At time step 0, *move* action is performed by the blue robot $R1$. If $R1$ places the knife in its right hand on the table at time step 1, no discrepancy will be observed at time step 2. To ensure that a discrepancy is observed at time step 2, Algorithm 4 introduces *(R1,RightArm)* to the set $X^{<t}_{real}$, so that *placeOn* action cannot be executed at time step 1. As a result, a discrepancy can be detected at time step 2, since the knife is expected to be on the table according to the world state (left column), while it has not been placed there (right column).

Since the utilized execution monitoring algorithm does not employ replanning, the execution of the initial plan continues as follows. At time step 2, the blue robot $R1$ moves away from the table, while the yellow robot $R2$ moves to the table. At time step 3, $R2$ executes *pickUp* action with its left arm to grab the knife from the table. The execution of this action cannot succeed since the knife is not at the table. Note that the execution takes place at the world state and the table becomes empty (left column), causing the previously introduced discrepancy to become undetectable. At this point, unless a new relevant broken part is introduced, no discrepancy will be detected during the execution of the rest of the plan. Therefore, at time step 4, Algorithm 4 introduces *(R2,RightArm)* to the set $X^{<t}_{real}$. With the addition of this new broken part, $R2$ cannot place the fork on the table at time step 4 and a discrepancy can be detected at time step 5. Robots perform the rest of the plan and cannot reach the goal state. For this scenario, $X^{<t}_{real} = \{(R1,RightArm),(R2,RightArm)\}$. Then $X^{<t}_{real}$ is added into $\sum X^{<t}_{real}$ so that Algorithm 4 can generate different scenarios with different broken part pairs.

6.4.2 Case Study 2: Discrepancy generation with replanning

We present another case study in Figure 6.4.2 to demonstrate the use of the discrepancy generation within an execution monitoring algorithm, where diagnostic reasoning and guided replanning algorithms are implemented.

We consider the same scenario as in Case Study 1. These two scenarios are identical until the first discrepancy is detected at time step 2. However, when the discrepancy is detected, the execution monitoring algorithm identifies the cause of the failure as the right arm of blue robot $R1$ being broken and generates a new plan to reach the goal without using the broken robot component as:

$P2 = \langle$*move(R1,Corner,2), move(R2,Corner,2), pickUp(R2,Knie,LeftArm,3), move(R2,TableLeft,4), placeOn(R2,RightArm,Table,5)*$\rangle$.

The right column of Figure 6.4.2 presents the observed states generated by Algorithm 4. Once again in this case, Algorithm 4 requires the right hand of the blue robot $R1$ to be broken at time step 1. Also the base of the yellow robot $R2$ is considered as broken at time step 4.

According to $P2$, the robots navigate to the corner at time step 2, and $R2$ picks up the knife with its left hand from the right hand of $R1$ at time step 3. At this point, Algorithm 4 introduces *(R2,base)* to the set $X^{<t}_{real}$. Consequently, $R2$ cannot perform *move* action at time step 4, so the yellow robot will not be around the table at time step 5 (right column of Figure 6.4.2); this results in a discrepancy detected by the algorithm. Afterwards, the execution monitoring algorithm successfully identifies the broken components as the cause of this discrepancy, and generates a new plan which reaches the goal state. Then $\{$*(R1,RightArm)*, *(R2,base)*$\}$ is added into $\sum X^{<t}_{real}$ such that Algorithm 4 can generate different scenarios with different broken part pairs in later executions.

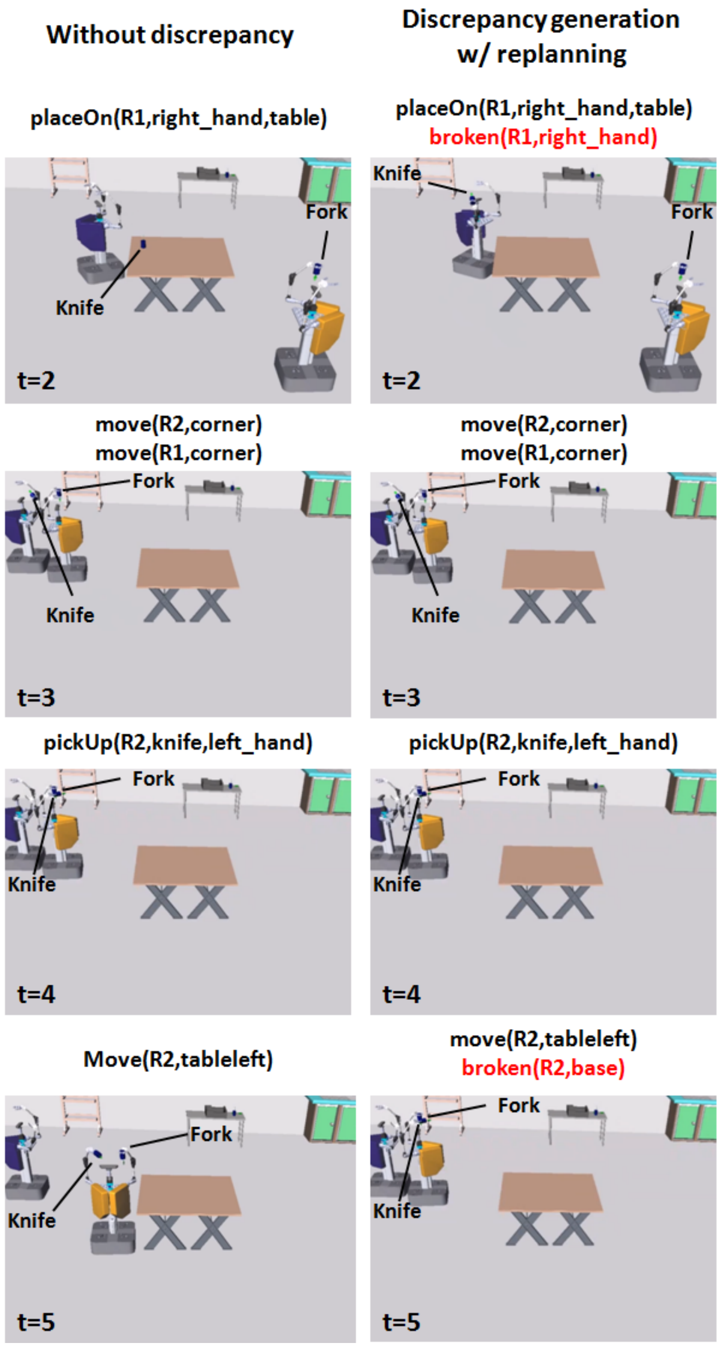


Figure 6.4.2: Left: World states at each time step. Right: Observed states generated at each time step for Case Study 2.

6.5 Case Studies: Cognitive Factory Domain

Let us also demonstrate applications of Algorithm 4 with a cognitive factory setting. We consider an execution monitoring algorithm that utilizes revised diagnosis.

6.5.1 A Representation of the Cognitive Factory Domain in ASP

We consider a cognitive factory where multiple robots collaboratively paint boxes by performing painting, waxing and stamping processes, in this order.

There are two types of robots in the domain workers and chargers. The worker robots have reconfigurable end-effectors which can perform different actions by changing its end-effector at pit stop location. The charger robots charge worker robots whenever their batteries are depleted. The battery levels of worker robots decreases as they move: 1 level decrease for 1 unit of movement. They also decreases by *workCons* amount as they work on boxes.

The cognitive factory domain is partially observable meaning that only boxes at the delivery position (i.e., at the end of production line) can be observed. Therefore, any defects (i.e., discrepancies) related to these boxes can only be detected at this position.

We represent this cognitive factory in ASP. Our factory domain representation mainly builds on and extends the one introduced in [65] [66]. This factory is depicted as a grid where holonomic robots can move from one grid to another in straight lines.

We consider the following set of atoms to describe the domains of variables used in our ASP formulation:

- *worker(w)*: *w* is a worker robot,

- *batteryLvl(bl)*: *bl* is possible battery level of workers,
- *effector(e)*: *e* is a possible end effector that a worker robot can use,
- *charger(c)*: *c* is a charger robot,
- *robot(r)*: r is a robot which can be both worker or charger,
- *xCoord(x)*: *x* is a possible coordinate in x direction,
- *yCoord(t)*: *y* is a possible coordinate in y direction,
- *dir(d)*: *d* is a possible direction where robots can move (e.g., up, left),
- *unit(u)*: *u* is a possible amount of movement in any direction,
- *box(b)*: *b* is a box that robots can work on,
- *workstages(ws)*: *ws* is a possible operation (e.g., 0:paint, 1:wax, 2:stamp),
- *time*(i): $i \in \{0,\dots,k\}$ where *k* is a given upper bound on the makespan,
- *atime*(i): $i \in \{0,\dots,k-1\}$ where *k* is a given upper bound on the makespan.

We consider the following set of fluents:

- *posX(r,x), posY(r,y)*: robot *r* is located at coordinate *x* and *y*,
- *battery(w,bl)*: battery level of worker robot *w* is *bl*,
- *endEffector(w,e)*: end effector type of worker *w* is *e*,
- *docked(c,w,n)*: charger *c* is docked to worker *w* (*n*=1 if they are docked 0 otherwise),
- *linePos(b,x)*: position of box *b* is *x* on the production line,

- *wetPaint(b)*: box *b* is painted recently and it is still wet,
- *workDone(b,ws)*: workstage *ws* is performed on box *b*.

We consider the following set of actions:

- *move(r,d,u)*: robot *r* moves in direction *d* by *u* units,
- *workOn(w,b)*: worker *w* works on the box *b*,
- *swapEndEffector(w,e)*: worker *w* changes its current end effector with *e*,
- *dock(c,w)*: charger *c* docks to worker *w*,
- *undock(c)*: charger *c* undocks from the worker it is docked,
- *charge(c)*: charger *c* chargers the worker it is docked,
- *lineShift*: shifts box positions on the conveyor belt.

Since the cognitive factory domain is partially observable and defects of the boxes can only be observed at the end of the production line, we have introduced some additional actions in addition to actions given in [65] [66]:

- *feed(b)*: feeder places the box *b* on the production line.
- *feedBack(b)*: if the box *b* has any defects, another feeder moves box *b* to beginning of the production line.

Direct Effects and Preconditions of Actions: Following rules describes direct effects and preconditions of *move*, *workOn* and *charge* actions.
<u>Action *move*:</u> When a robot *r* at position *x* moves *u* amount of units in the *right*

direction at time step *i*, then at the next time step $i+1$ its position in the x coordinates becomes $x+u$.

$$posX(r,x+u,i+1) \leftarrow move(r,right,u,i), posX(x,i), rob(r), unit(u), xCoord(x), atime(i).$$

After a worker robot w with battery level bl moves u amount of units in any direction at time step i, then at the next time step $i+1$ its battery level becomes $bl-u$.

$$battery(w,bl-u,i+1) \leftarrow move(w,d,u,i), battery(w,bl,i), dir(d), worker(w), batteryLvl(bl), unit(u), atime(i).$$

A worker robot w cannot move u amount of units if its battery level is not enough.

$$\leftarrow move(w,d,u,i), battery(w,bl,i), bl < u, worker(w), dir(d), unit(u), batteryLvl(bl), atime(i).$$

A worker robot w cannot move if it is docked to a charger robot c.

$$\leftarrow move(w,d,u,i), docked(c,w,1,i), worker(w), dir(d), unit(u), atime(i).$$

Robots cannot move outside of the predefined grid borders. For instance, following precondition prevents robots from moving outside of the right border.

$$\leftarrow move(r,right,u,i), posX(r,x,i), x > maxX\text{-}u, robot(r), unit(u), xCoord(x), atime(i).$$

<u>Action *workOn*:</u> After a worker robot w works on a box b whose current work stage is $ws-1$ at time step i, then at the next time step $i+1$ the work stage of the

box becomes *ws*.

$$workDone(b, ws, i+1) \leftarrow workOn(w, b, i), workDone(b, ws-1, i), box(b),$$
$$workstages(ws), worker(w), atime(i).$$

A worker robot's battery level *bl* decreases by a predefined amount *workCons* after the robot works on a box.

$$battery(w, bl-workCons, i+1) \leftarrow workOn(w, b, i), battery(w, bl, i), batteryLvl(bl),$$
$$worker(w), box(b), atime(i).$$

If the performed work stage is painting then condition of box *b* becomes wet paint.

$$wetpaint(b, i+1) \leftarrow workOn(w, b, i), workDone(b, 0, i), box(b), worker(w),$$
$$atime(i).$$

A worker robot *w* cannot work on a box if its end effector is not appropriate for the work stage.

$$\leftarrow workOn(w, b, i), workDone(b, ws-1, i), endEffector(w, e, i), e \neq ws, box(b),$$
$$worker(w), workstages(ws), atime(i).$$

A worker robot *w* cannot work on a box if its battery level is not sufficient.

$$\leftarrow workOn(w, b, i), battery(w, bl, i), bl < workCons, worker(w), box(b),$$
$$batteryLvl(bl), atime(i).$$

Action *charge*: The battery level of a worker robot *w* is set to predefined value *maxLvl* after a charger robot *c* charges the worker.

$$battery(w, maxLvl, i+1) \leftarrow charge(c,i), docked(c,w,1,i), worker(w), charger(c),\\ atime(i).$$

However, charging is not possible if the worker *w* and the charger *c* are not docked together.

$$\leftarrow charge(c,i), docked(c,w,0,i), charger(c), worker(w), atime(i).$$

Ramification Rules: We express that a box with wet paint dries at the next time step by the following ramification rule.

$$\neg wetpaint(b, i+1) \leftarrow wetpaint(b,i), box(b), atime(i).$$

State Constraints: We ensure that at any state, two worker robots cannot occupy the same grid cell by the following constraint:

$$\leftarrow posX(w1,x1,i), posY(w1,y1,i), posX(w2,x2,i), posY(w2,y2,i), x1 = x2,\\ y1 = y2, w1 \neq w2, worker(w1), worker(w2), xCoord(x1), xCoord(x2), yCoord(y1),\\ yCoord(y2), time(i).$$

Two boxes cannot occupy the same position on the production line unless they are at the starting position (i.e., *minLine*) or the delivery position (i.e., *maxLine*).

$$\leftarrow linePos(b1,x,i), linePos(b2,x,i), x < maxLine, x > minLine, box(b1), box(b2),\\ b1 \neq b2, xcoord(x), time(i).$$

Noconcurrency Constraints: There exist several concurrency constraints to ensure some actions do not occur at the same time. For instance, a worker robot *w* cannot work on a box while the line is shifting or the robot is moving.

$$\leftarrow \mathit{workOn}(w,b,i), \mathit{lineShift}(i), \mathit{worker}(w), \mathit{box}(b), \mathit{atime}(i).$$

$$\leftarrow \mathit{workOn}(w,b,i), \mathit{move}(w,d,u,i), \mathit{worker}(w), \mathit{box}(b), \mathit{dir}(d), \mathit{unit}(u), \mathit{atime}(i).$$

The worker and charger robots cannot move while they are docking.

$$\leftarrow \mathit{dock}(c,w,1,i), \mathit{move}(c,d,u,i), \mathit{charger}(c), \mathit{worker}(w), \mathit{dir}(d), \mathit{unit}(u), \mathit{atime}(i).$$

External Atoms: Since robots are operating in a continuous space, for the feasibility of actions, we embed the relevant low-level feasibility checks into the domain description. For instance, a robot *r* cannot move in directions *d1* and *d2* with *u1* and *u2* units if there is no collision-free trajectory that it can follow.

$$\begin{aligned}&\leftarrow \mathit{move}(r,d1,u1,i), \mathit{move}(r,d2,u2,i), \mathit{posX}(r,x,i), \mathit{posY}(r,y,i), \mathit{robot}(r),\\ &\mathit{not}\ \&\mathit{checkCollision}[x,y,d1,d2,u1,u2](), \mathit{dir}(d1), \mathit{dir}(d2), \mathit{unit}(u1), \mathit{unit}(u2),\\ &\mathit{xCoord}(x), \mathit{yCoord}(y), \mathit{atime}(i).\end{aligned}$$

6.5.2 Systematic Evaluation of an Execution Monitoring Algorithm Guided with Revised Diagnosis

We have systematically evaluated the performance of an execution monitoring algorithm, equipped with revised diagnosis, for a case study in the cognitive factory domain by using Algorithm 4. In this case study, we consider 3 worker robots and 2 charger robots, and 3 boxes to be painted.

We have performed two sets of experiments where the upper bound on the number of broken parts are set to 1 and 2, respectively. The results of these experiments are shown in Table 6.5.2. In the first part of the evaluation, the upper bound on the number of broken parts is set to 1; Algorithm 4 has generated all possible 29 scenarios. In this experiment, the execution monitoring algorithm displayed a 76% success rate.

In the second part of the evaluation, the upper bound on the number of broken parts is set to 2. This time, Algorithm 4 has generated 369 scenarios; the execution monitoring algorithm displayed a 65% success rate.

Table 6.5.1: Systematic evaluation of execution monitoring algorithm with revised diagnosis

Number of Broken Parts	AverageNumber Replannings	Number of Generated Scenarios	Success (%)
1	1.23	29	76
2	2.4	369	65

Please note that Algorithm 4 could not generate all possible scenarios for the case where the upper bound on the number of broken parts is set to 2. Let us explain this by some examples shown in Figure 6.5.1.

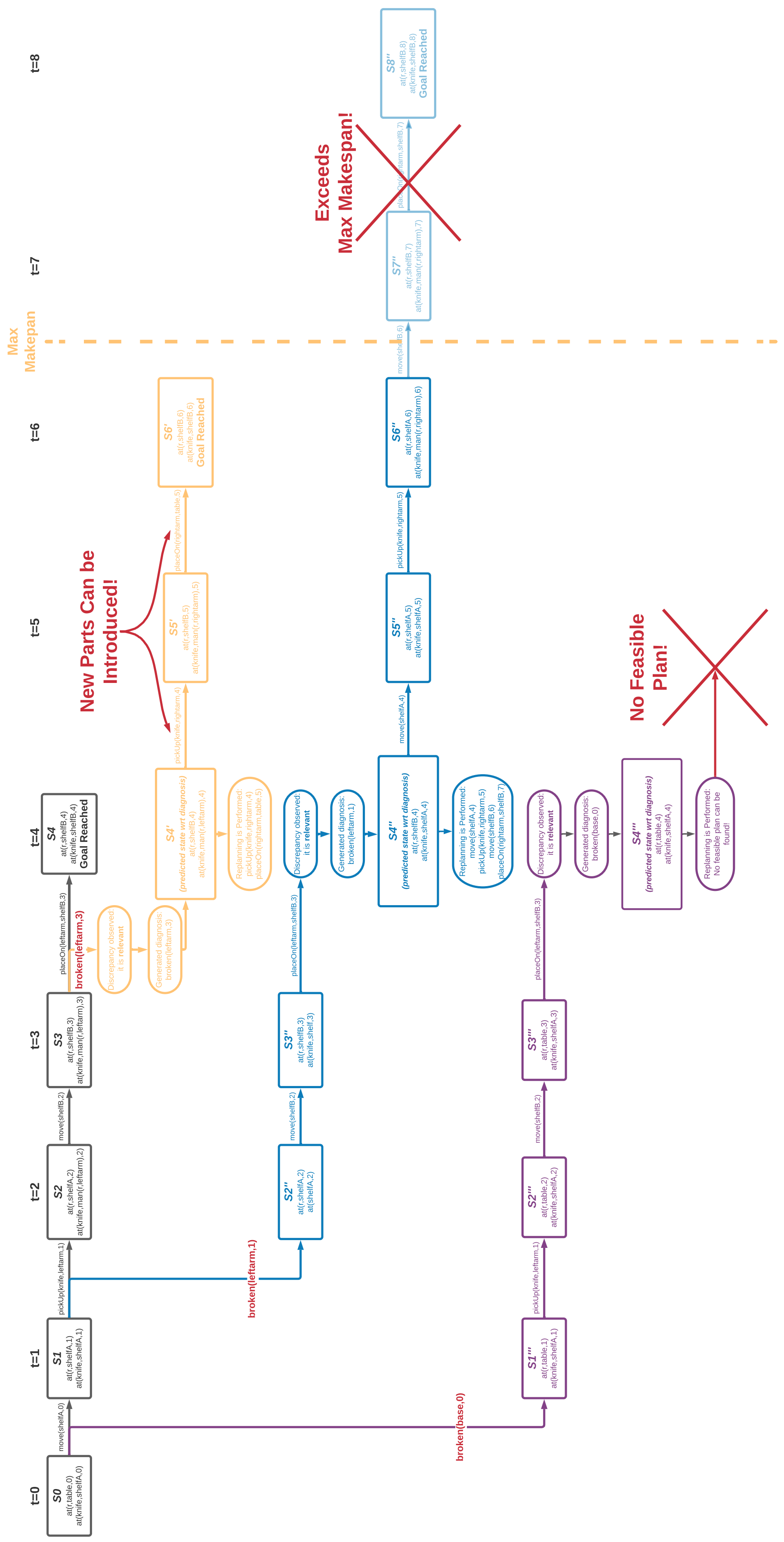
The black trace indicates a successful execution of the initial plan where robot *r* moves to *shelfA*, picks up the *knife*, moves to *shelfB* and places *knife* on the *shelfB*.

The blue trace indicates the scenario where Algorithm 4 introduces *(leftarm,1)* at time step *1* as broken. The execution monitoring algorithm generates a new plan considering the correct diagnosis: robot moves back to *shelfA*, picks up the *knife* with its *rightarm*, moves to *shelfB* and places the *knife* on the *shelfB*. Let us assume that the maximum makespan for this example is specified as 6. Then this newly generated plan exceeds this upper bound, and no shorter plan can be generated. Therefore, Algorithm 4 cannot introduce any further scenarios.

The purple trace indicates the scenario where Algorithm 4 introduces *(base,0)* at time step *0* as broken. Since the *base* of the robot is diagnosed as broken, Algorithm 4 cannot generate a new plan and introduce any further scenarios.

The orange trace shows an example where Algorithm 4 introduces *(leftarm,3)* at time step *3* as broken. Algorithm 4 performs replanning considering the correct diagnosis so that robot picks up the *knife* with its *rightarm* and places it on the *shelfB*. For this trace, Algorithm 4 can introduce further robotic parts as broken to generate additional scenarios. Note that if the utilized execution monitoring algorithm generates an incorrect diagnosis (e.g., *broken(rightarm,2)*), the observed relevant discrepancy will not be resolved and Algorithm 4 will not introduce any additional broken components to generate additional scenarios.

As discussed with the cases above, Algorithm 4 cannot generate all scenarios but 369 different and relevant scenarios. Generating 369 scenarios is sufficient to comprehensively evaluate the performance of an execution monitoring algorithm.



t=0
t=1
t=2
t=3
t=4
t=5
t=6
t=7
t=8
Max Makepan
S0 at(r,table,0) at(knife,shelfA,0)
move(shelfA,0)
S1 at(r,shelfA,1) at(knife,shelfA,1)
pickUp(knife,leftarm,1)
S2 at(r,shelfA,2) at(knife,man(r,leftarm),2)
move(shelfB,2)
S3 at(r,shelfB,3) at(knife,man(r,leftarm),3)
placeOn(leftarm,shelfB,3)
S4 at(r,shelfB,4) at(knife,shelfB,4) Goal Reached
broken(leftarm,3)
Discrepancy observed: it is relevant
Generated diagnosis: broken(leftarm,3)
S4' (predicted state wrt diagnosis) at(r,shelfB,4) at(knife,man(r,leftarm),4)
Replanning is Performed: pickUp(knife,rightarm,4) placeOn(rightarm,table,5)
pickUp(knife,rightarm,4)
S5' at(r,shelfB,5) at(knife,man(r,rightarm),5)
placeOn(rightarm,table,5)
S6' at(r,shelfB,6) at(knife,shelfB,6) Goal Reached
New Parts Can be Introduced!
broken(leftarm,1)
S2'' at(r,shelfA,2) at(knife,shelfA,2)
move(shelfB,2)
S3'' at(r,shelfB,3) at(knife,shelf,3)
placeOn(leftarm,shelfB,3)
Discrepancy observed: it is relevant
Generated diagnosis: broken(leftarm,1)
S4'' (predicted state wrt diagnosis) at(r,shelfB,4) at(knife,shelfA,4)
Replanning is Performed: move(shelfA,4) pickUp(knife,rightarm,5) move(shelfB,6) placeOn(rightarm,shelfB,7)
move(shelfA,4)
S5'' at(r,shelfA,5) at(knife,shelfA,5)
pickUp(knife,rightarm,5)
S6'' at(r,shelfA,6) at(knife,man(r,rightarm),6)
move(shelfB,6)
S7'' at(r,shelfB,7) at(knife,man(r,rightarm),7)
placeOn(rightarm,shelfB,7)
S8'' at(r,shelfB,8) at(knife,shelfB,8) Goal Reached
Exceeds Max Makespan!
broken(base,0)
S1''' at(r,table,1) at(knife,shelfA,1)
pickUp(knife,leftarm,1)
S2''' at(r,table,2) at(knife,shelfA,2)
move(shelfB,2)
S3''' at(r,table,3) at(knife,shelfA,3)
placeOn(leftarm,shelfB,3)
Discrepancy observed: it is relevant
Generated diagnosis: broken(base,0)
S4''' (predicted state wrt diagnosis) at(r,table,4) at(knife,shelfA,4)
Replanning is Performed: No feasible plan can be found!
No Feasible Plan!

Figure 6.5.1: Examples where additional scenarios cannot be generated

Chapter 7

Robustness of Plans

For a given planning problem, multiple initial plans can be generated to reach the goal, possibly subject to different optimizations. We introduce a method (based on Algorithm 4) to analyze the “robustness” of these plans with respect to possible failures due to broken robotic components.

7.1 Robustness of Plans with respect to Possible Failures

We define the robustness of a plan as the ratio between the number of successful scenarios where the goal state is reached and the total number of scenarios generated by Algorithm 4 with respect to a diagnostic reasoning of type T.

Definition 11. *A plan P_1 is* ***more robust*** *than a plan P_2* ***with respect to*** *a diagnosis* *type T if*

$$\big(\textit{success}(P_1,T),\ \textit{-\#averageReplannings}(P_1,T)\big) >_T \big(\textit{success}(P_2,T),\ \textit{-\#averageReplannings}(P_2,T)\big)$$

where $>_T$ denotes lexicographic ordering.

If the success rate of the execution monitoring algorithm for plan P_1 is greater than plan P_2, then we can conclude that P_1 is more robust than P_2. If the success rates of the execution monitoring algorithm for plans P_1 and P_2 are equal, than we compare these plans in terms of their average number of replannings. The one with the lower average number of replannings is considered to be more robust.

Definition 12. *A plan P_1 is* ***equally robust*** *with a plan P_2* ***with respect to*** *a diagnosis type T if*

$$\big(\textit{success}(P_1,T),\ \textit{-\#averageReplannings}(P_1,T)\big) =_T \big(\textit{success}(P_2,T),\ \textit{-\#averageReplannings}(P_2,T)\big).$$

Instead of comparing robustness of plans with respect to a single type of diagnosis type T, we can compare them with respect to a set $\mathscr{T} = \{T_1, T_2, \ldots, T_n\}$ of diagnosis types.

Definition 13. *A plan P_1 is* ***more robust*** *than a plan P_2* ***with respect to*** *$\mathscr{T} = \{T_1, T_2, \ldots, T_n\}$ if*

$$\left(\frac{\sum_{i=1}^{n} success(P_1,T_i)}{n}, -\frac{\sum_{i=1}^{n} \#averageReplannings(P_1,T_i)}{n}\right) >_{\mathscr{T}} \left(\frac{\sum_{i=1}^{n} success(P_2,T_i)}{n}, -\frac{\sum_{i=1}^{n} \#averageReplannings(P_2,T_i)}{n}\right).$$

where $>_{\mathscr{T}}$ denotes denotes lexicographic ordering.

Definition 14. *A plan P_1 is* ***equally robust*** *with a plan P_2* ***with respect to*** *$\mathscr{T} = \{T_1, T_2, \ldots, T_n\}$ if*

$$\left(\frac{\sum_{i=1}^{n} success(P_1,T_i)}{n}, -\frac{\sum_{i=1}^{n} \#averageReplannings(P_1,T_i)}{n}\right) =_{\mathscr{T}}$$
$$\left(\frac{\sum_{i=1}^{n} success(P_2,T_i)}{n}, -\frac{\sum_{i=1}^{n} \#averageReplannings(P_2,T_i)}{n}\right).$$

Let us illustrate plan robustness evaluation on the same experimental setup as in Section 5.4.1.

We consider a cognitive factory domain setting with 3 worker and 2 charger robots to paint 3 boxes. We consider three different initial plans, P_1, P_2 and P_3, which reach the goal state with different optimizations: P_1 minimizes the number of *charge* actions, P_2 minimizes the number of *move* actions, and P_3 minimizes the amount of time between the first and the last *lineshift* actions.

First, suppose that at most a single component is broken to generate discrepancies in each scenario. Table 7.1.1 illustrates the robustness results for these plans with respect to an execution monitoring algorithm that utilizes revised diagnosis. Table 7.1.2 illustrates the robustness results for an execution monitoring algorithm that utilizes reset diagnosis. According to Table 7.1.1, P_2 is more robust than P_1, and P_1 is more robust than P_3 with respect to the revised diagnosis:

$$P_2 >_{revised} P_1 >_{revised} P_3.$$

Table 7.1.2 illustrates that P_1 is more robust than P_2, and P_2 is more robust than P_3 with respect to the reset diagnosis:

$$P_1 >_{reset} P_2 >_{reset} P_3.$$

Let us now suppose that at most two robotic components can be broken in each scenario.

Table 7.1.1: Robustness results with respect to revised diagnosis (single broken component)

Plan	P_1	P_2	P_3
success (%)	74	76	42
#averageReplannings	1.18	1.23	1.14
averageReplanningTime	116.8	217.1	85.8
averageDiagnosisTime	0.88	1.55	0.64
averagePlanLength	39.4	51.6	34.8
#generatedScenarios	23	29	33

Table 7.1.2: Robustness results with respect to reset diagnosis (single broken component)

Plan	P_1	P_2	P_3
success (%)	74	69	42
#averageReplannings	1.12	1.15	1.14
averageReplanningTime	110	178	99
averageDiagnosisTime	0.73	0.80	0.80
averagePlanLength	38.9	39.5	34.8
#generatedScenarios	23	29	33

Table 7.1.3 illustrates the robustness results for the plans with respect to an execution monitoring algorithm that utilizes revised diagnosis. According to these results, P_2 is more robust than P_3, and P_3 is more robust than P_1:

$$P_2 >_{revised} P_3 >_{revised} P_1.$$

Table 7.1.4 shows that P_2 is more robust than P_1, and P_1 is more robust than P_3 with respect to an execution monitoring algorithm that utilizes reset diagnosis:

$$P_2 >_{reset} P_1 >_{reset} P_3.$$

Table 7.1.3: Robustness results with respect to revised diagnosis (up to two broken components)

Plan	P_1	P_2	P_3
success (%)	59	65	60
#averageReplannings	2.2	2.4	2.3
averageReplanningTime	101.6	168.8	136.1
averageDiagnosisTime	2.03	2.48	2.05
averagePlanLength	52.2	55.7	49.4
#generatedScenarios	257	369	224

Table 7.1.4: Robustness results with respect to reset diagnosis (up to two broken components)

Plan	P_1	P_2	P_3
success (%)	46	57	40
#averageReplannings	2.9	2.2	2.8
averageReplanningTime	224	144	128
averageDiagnosisTime	3.73	2.56	2.83
averagePlanLength	58	45	53
#generatedScenarios	263	416	216

Table 7.1.5 illustrates the robustness results for the plans with respect to $\mathscr{T} = \{reset, revised\}$: P_2 is more robust than P_1, and P_1 is more robust than P_3.

$$P_2 >_{\mathscr{T}} P_1 >_{\mathscr{T}} P_3.$$

Table 7.1.5: Robustness results (up to two broken components)

Plan	P_1	P_2	P_3
success (%)	52	61	50
#averageReplannings	2.5	2.3	2.5
averageReplanningTime	163	156	131
averageDiagnosisTime	2.89	2.52	2.44
averagePlanLength	55	50	51
#generatedScenarios	520	785	440

In light of these experiments, we have made the following observations:

- Regardless of the diagnostic reasoning type, plan P_2 is more robust for the execution.
- Minimizing the amount of time between the first and the last *lineshift* actions may result in observing several discrepancies at the same time step, since before detecting any broken components of worker robots, they might work one multiple boxes. Therefore, robustness of this plan (P_3) is low.
- If the base of a robot is broken, it cannot navigate and perform its duties. Since plan P_2 minimizes number of *move* actions, it reduces the possibility of such failures. Therefore, P_2 has the highest robustness.
- A plan's execution is more robust with respect to an execution monitoring algorithm that utilizes revised diagnosis compared to reset diagnosis.

7.2 Reliability of Robotic Components

After identifying the most robust plan, we can further improve the robustness of the plan by examining the reliability of robotic components. Let us consider the scenarios generated for plan robustness evaluation. For each robotic part $(r,p) \in \mathscr{R}$, suppose that $\#scenarios_{(r,p)}$ denotes the number of scenarios where the robotic component *(r,p)* is introduced as broken by Algorithm 4. Similarly, $\#failedScenarios_{(r,p)}$ denotes the number of scenarios that an execution monitoring algorithm could not reach the goal state when the robotic component *(r,p)* is broken. Then we define $occuranceRate_{(r,p)}$, $failureRate_{(r,p)}$ and $severityRate_{(r,p)}$ as follows:

$$occuranceRate_{(r,p)} = \frac{\#scenarios_{(r,p)}}{\#generatedScenarios} \times 100$$

$$failureRate_{(r,p)} = \frac{\#failedScenarios_{(r,p)}}{\#scenarios_{(r,p)}} \times 100$$

$$severityRate_{(r,p)} = occuranceRate_{(r,p)} \times failureRate_{(r,p)} \div 100$$

Table 7.2.1 illustrates the reliability of robotic components during an execution of plan P_1. According to these results, *(Worker3, Arm)* is the cause of a discrepancy for 15% of the generated scenarios and 39% of these scenarios have failed to reach the goal. As a result this component has the highest severity rate of 5.8%.

Table 7.2.1: Reliability of robotic components during execution of P_1

(r, p)	occuranceRate (%)	failureRate (%)	severityRate (%)
(Charger1, Charge Port)	7.0	51	3.6
(Charger1, Base)	9.9	47	4.7
(Charger2, Charge Port)	7.0	70	4.9
(Charger2, Base)	4.2	83	3.5
(Worker1, Arm)	11.9	38	4.5
(Worker1, Base)	11.5	47	5.4
(Worker2, Arm)	12.9	39	5.1
(Worker2, Base)	10.5	54	5.7
(Worker3, Arm)	**15.1**	**39**	**5.8**
(Worker3, Base)	9.6	55	5.2

Table 7.2.2 illustrates the reliability of robotic components during an execution of plan P_3. According to these results, *(Worker1, Base)* is the cause of a discrepancy for 17% of the generated scenarios and 54% of these scenarios have failed to reach the goal. As a result this component has the highest severity rate of 9.3%.

Table 7.2.2: Reliability of robotic components during execution of P_3

(r, p)	occuranceRate (%)	failureRate (%)	severityRate (%)
(Charger1, Charge Port)	7.8	58	4.5
(Charger1, Base)	8.8	55	4.9
(Charger2, Charge Port)	5.7	48	2.8
(Charger2, Base)	7.8	42	3.3
(Worker1, Arm)	11.0	40	4.4
(Worker1, Base)	**17.1**	**54**	**9.3**
(Worker2, Arm)	8.9	38	3.4
(Worker2, Base)	9.9	25	2.6
(Worker3, Arm)	6.9	47	3.2
(Worker3, Base)	15.6	46	7.1

Table 7.2.3: Reliability of robotic components during execution of P_2

(r, p)	occuranceRate (%)	failureRate (%)	severityRate (%)
(Charger1, Charge Port)	7.5	57	4.3
(Charger1, Base)	5.4	56	3.1
(Charger2, Charge Port)	9.5	65	6.3
(Charger2, Base)	4.3	74	3.2
(Worker1, Arm)	10.8	56	6.0
(Worker1, Base)	12.3	43	5.3
(Worker2, Arm)	**17.0**	**48**	**8.2**
(Worker2, Base)	13.8	35	4.8
(Worker3, Arm)	15.6	42	6.5
(Worker3, Base)	3.5	35	1.2

Table 7.2.3 illustrates the reliability of robotic components during an execution of plan P_2. According to these results, *(Worker2,Arm)* is the cause of a discrepancy for 17% of the generated scenarios and 48% of these scenarios have failed to reach the goal. As a result this component has the highest severity rate of 8.2%. Results of robustness evaluation suggest that user should pick plan P_2 since it is the most robust plan for given execution monitoring algorithm. Furthermore, reliability evaluation indicates that user may consider replacing *(Worker2, Arm)* with a more reliable component to further increase performance of an execution.

Chapter 8

Conclusion and Future Work

We have introduced a formal method to predict states under partial observability which takes into account the previous observations and diagnoses. We have used this method for three purposes: to detect discrepancies, to check their relevancies, and to infer a full current state for replanning.

We have introduced a formal method for *diagnostic reasoning* under partial observability to generate meaningful explanations for the relevant discrepancies. This method provides not only the most likely causes for relevant discrepancies, but also informative explanations regarding action failures. Our method is general in that it can be utilized in three different modes taking into account the previous observations, previous diagnosis, current observation and/or earlier observations.

We have introduced a formal method for *replanning* under partial observability that generates new plans, guided by the diagnosis with explanations. This method also considers repairing minimum number of broken robotic components, if needed.

Based on formal methods for various reasoning tasks; hybrid planning, diagnostic reasoning and explanation generation, discrepancy detection and checking for their relevancy, and guided replanning with repairs, we have introduced a plan execution monitoring algorithm that works under partial observability and can be used autonomously or interactively. The introduced execution monitoring is flexible and modular such that it can be used with different diagnostic reasoning and (guided) replannig approaches.

The experimental evaluation of the proposed execution monitoring algorithm over large sets of problem instances has shown the usefulness of diagnostic reasoning with explanations and guided replanning with repairs. We have also observed that utilizing the current and the previous observations and being able to revise the previous diagnose improve the effectiveness of the execution monitoring algorithm.

We have introduced a formal method to dynamically and systematically generate relevant discrepancies that occur due to broken robotic components. Based on this method, we have introduced a simulation algorithm to evaluate performance of various execution monitoring algorithms and the robustness of plans. This simulation algorithm is applicable to variety of execution monitoring algorithms with/without diagnostic reasoning and/or replanning.

Future Work

In this dissertation, we consider a special type of faults, "broken components of robots", as the cause of a discrepancy that lead to a plan failure. We plan to extend causes of discrepancies with different type of faults, such as discrepancies caused by exogenous actions or unreliable action executions.

Our diagnostic reasoning method assumes broken components can only affect execution of actuation actions. As part of our future work, we plan to extend our method to conditional planning, where in addition to actuation actions, sensing action failures need to be considered.

Given that conditional planning necessitates representation of non-deterministic effect for sensing actions, this extension will also help us to generalize our approach to system with non-deterministic affects.

Bibliography

[1] C. Fritz, “Monitoring the generation and execution of optimal plans,” Ph.D. dissertation, University of Toronto, 2009.

[2] S. J. Levine, “Monitoring the execution of temporal plans for robotic systems,” Ph.D. dissertation, MIT, 2012.

[3] M. Gelfond and V. Lifschitz, “Classical negation in logic programs and disjunctive databases,” *New Generation Computing*, vol. 9, pp. 365–385, 1991.

[4] ——, “The stable model semantics for logic programming,” in *Proc. of ICLP*. MIT Press, 1988, pp. 1070–1080.

[5] V. Lifschitz, *Answer Set Programming*. Springer, 2019. [Online]. Available: https://doi.org/10.1007/978-3-030-24658-7

[6] E. Erdem, M. Gelfond, and N. Leone, “Applications of answer set programming,” *AI Magazine*, vol. 37, no. 3, pp. 53–68, 2016.

[7] E. Erdem and V. Patoglu, “Applications of ASP in robotics,” *Kunstliche Intelligenz*, vol. 32, no. 2-3, pp. 143–149, 2018.

[8] G. De Giacomo, R. Reiter, and M. Soutchanski, “Execution monitoring of high-level robot programs,” in *Proc. of KR*, 1998, pp. 453–465.

[9] E. Khalastchi and M. Kalech, “On fault detection and diagnosis in robotic systems,” *ACM Comput. Surv.*, vol. 51, no. 1, 2018.

[10] G. Havur, K. Haspalamutgil, C. Palaz, E. Erdem, and V. Patoglu, “A case study on the tower of hanoi challenge: Representation, reasoning and execution,” in *Proc. of ICRA*, 2013, pp. 4552–4559.

[11] E. Erdem, E. Aker, and V. Patoglu, “Answer set programming for collaborative housekeeping robotics: representation, reasoning, and execution,” *Intelligent Service Robotics*, vol. 5, pp. 275–291, 2012.

[12] M. Colledanchise, D. Almeida, and P. Ögren, “Towards blended reactive planning and acting using behavior trees,” in *Proc. of ICRA*, 2019, pp. 8839–8845.

[13] H. Kress-Gazit, G. E. Fainekos, and G. J. Pappas, “Where’s waldo? sensor-based temporal logic motion planning,” in *Proc. of ICRA*, 2007, pp. 3116–3121.

[14] K. He, M. Lahijanian, L. E. Kavraki, and M. Y. Vardi, “Towards manipulation planning with temporal logic specifications,” in *Proc. of ICRA*, 2015, pp. 346–352.

[15] R. E. Fikes, P. E. Hart, and N. J. Nilsson, “Learning and executing generalized robot plans,” *Artificial Intelligence*, vol. 3, pp. 251–288, 1972.

[16] C. Fritz, “Execution monitoring,” 2005. [Online]. Available: http://www.cs.toronto.edu/~fritz/publications/depth_oral.pdf

[17] O. Pettersson, “Execution monitoring in robotics: A survey,” *Robotics and Autonomous Systems*, vol. 53, no. 2, pp. 73–88, 2005.

[18] J. A. Ambros-Ingerson and S. Steel, “Integrating planning, execution and monitoring,” in *Proc. of AAAI*, 1988, pp. 83–88.

[19] M. Fichtner, A. Großmann, and M. Thielscher, “Intelligent execution monitoring in dynamic environments,” *Fundamenta Informaticae*, vol. 57, no. 2-4, pp. 371–392, 2003.

[20] J. Winkler, G. Bartels, L. Mösenlechner, and M. Beetz, “Knowledge Enabled High-Level Task Abstraction and Execution,” *Conference on Advances in Cognitive Systems*, vol. 2, no. 1, pp. 131–148, 2012.

[21] S. Lemai and F. Ingrand, “Interleaving temporal planning and execution in robotics domains,” in *Proc. of AAAI*, 2004, pp. 617–622.

[22] S. Zhang, M. Sridharan, M. Gelfond, and J. Wyatt, “Towards an architecture for knowledge representation and reasoning in robotics,” in *International Conference on Social Robotics*. Springer, 2014, pp. 400–410.

[23] L. P. Kaelbling and T. Lozano-Pérez, “Integrated task and motion planning in belief space,” *Int. J. Robotics Res.*, vol. 32, no. 9-10, pp. 1194–1227, 2013.

[24] J. P. Mendoza, M. Veloso, and R. Simmons, “Plan execution monitoring through detection of unmet expectations about action outcomes,” in *Proc. of ICRA*, 2015, pp. 3247–3252.

[25] M. Hanheide, M. Göbelbecker, G. S. Horn, A. Pronobis, K. Sjöö, A. Aydemir, P. Jensfelt, C. Gretton, R. Dearden, M. Janicek *et al.*, “Robot task planning and explanation in open and uncertain worlds,” *Artificial Intelligence*, vol. 247, pp. 119–150, 2017.

[26] T. Eiter, E. Erdem, W. Faber, and J. Senko, “A logic-based approach to finding explanations for discrepancies in optimistic plan execution,” *Fundamenta Informaticae*, vol. 79, no. 1–2, pp. 25–69, 2007.

[27] R. Reiter, “A theory of diagnosis from first principles,” *Artificial intelligence*, vol. 32, no. 1, pp. 57–95, 1987.

[28] J. De Kleer, A. K. Mackworth, and R. Reiter, “Characterizing diagnoses and systems,” *Artificial intelligence*, vol. 56, no. 2-3, pp. 197–222, 1992.

[29] N. Roos and C. Witteveen, “Models and methods for plan diagnosis,” *Autonomous Agents and Multi-Agent Systems*, vol. 19, no. 1, pp. 30–52, 2009.

[30] S. McIlraith, G. Biswas, D. Clancy, and V. Gupta, “Hybrid systems diagnosis,” in *International Workshop on Hybrid Systems: Computation and Control*. Springer, 2000, pp. 282–295.

[31] M. Balduccini and M. Gelfond, “Diagnostic reasoning with a-prolog,” *Theory and Practice of Logic Programming*, vol. 3, no. 4-5, p. 425âĂŞ461, 2003.

[32] C. Baral, S. McIlraith, and T. C. Son, “Formulating diagnostic problem solving using an action language with narratives and sensing,” in *KR*, 2000, pp. 311–322.

[33] C. Baral, M. Gelfond, and A. Provetti, “Representing actions: Laws, observations and hypotheses,” *J. Log. Program.*, vol. 31, no. 1-3, pp. 201–243, 1997.

[34] V. Marek and M. Truszczyński, "Stable models and an alternative logic programming paradigm," in *The Logic Programming Paradigm: a 25-Year Perspective*. Springer Verlag, 1999, pp. 375–398.

[35] I. Niemelä, "Logic programs with stable model semantics as a constraint programming paradigm," *Annals of Mathematics and Artificial Intelligence*, vol. 25, pp. 241–273, 1999.

[36] V. Lifschitz, "Answer set programming and plan generation," *Artificial Intelligence*, vol. 138, pp. 39–54, 2002.

[37] I. F. Yalciner, A. Nouman, V. Patoglu, and E. Erdem, "Hybrid conditional planning using answer set programming," *Theory and Practice of Logic Programming*, vol. 17, no. 5-6, pp. 1027–1047, 2017.

[38] M. Rizwan, V. Patoglu, and E. Erdem, "Human robot collaborative assembly planning: An answer set programming approach," *Theory Practice of Logic Programming*, vol. 20, no. 6, pp. 1006–1020, 2020.

[39] A. Nouman, V. Patoglu, and E. Erdem, "Hybrid conditional planning for robotic applications," *The International Journal of Robotics Research*, 2020, online first.

[40] D. Das, S. Banerjee, and S. Chernova, "Explainable AI for robot failures: Generating explanations that improve user assistance in fault recovery," in *Proc. of HRI*, 2021, pp. 351–360.

[41] D. E. Smith, "Planning as an iterative process," in *Proc. of AAAI*, J. Hoffmann and B. Selman, Eds., 2012.

[42] T. Chakraborti, S. Sreedharan, and S. Kambhampati, “The emerging landscape of explainable automated planning & decision making,” in *Proc. of IJCAI*, 2020, pp. 4803–4811.

[43] E. Erdem, V. Patoglu, and Z. G. Saribatur, “Integrating hybrid diagnostic reasoning in plan execution monitoring for cognitive factories with multiple robots,” in *Proc. of ICRA*, 2015, pp. 2007–2013.

[44] B. C. Williams, P. P. Nayak *et al.*, “A model-based approach to reactive self-configuring systems,” in *Proceedings of the national conference on artificial intelligence*, 1996, pp. 971–978.

[45] A. Lindsey and C. Pecheur, “Simulation-based verification of autonomous controllers via livingstone pathfinder,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2004, pp. 357–371.

[46] T. Kurtoglu, O. J. Mengshoel, and S. Poll, “A framework for systematic benchmarking of monitoring and diagnostic systems,” in *2008 international conference on prognostics and health management*. IEEE, 2008, pp. 1–13.

[47] K. L. Myers, “Towards a framework for continuous planning and execution,” in *Proc. of the AAAI Fall Symposium on Distributed Continual Planning*, vol. 12, 1998, p. 13.

[48] M. M. Veloso, M. E. Pollack, and M. T. Cox, “Rationale-based monitoring for planning in dynamic environments.” in *AIPS*, 1998, pp. 171–180.

[49] O. Sapena and E. Onaindia, “Execution, monitoring and replanning in dynamic environments,” *Working Notes of the AIPS*, vol. 2, 2002.

[50] J. McCarthy, “Elaboration tolerance,” in *Proc. of Commonsense*, 1998.

[51] T. Eiter, G. Ianni, R. Schindlauer, and H. Tompits, “A Uniform Integration of Higher-Order Reasoning and External Evaluations in Answer-Set Programming,” in *Proc. of IJCAI*, 2005, pp. 90–96.

[52] C. Dornhege, P. Eyerich, T. Keller, S. Trüg, M. Brenner, and B. Nebel, “Semantic attachments for domain-independent planning systems,” in *Proc. of ICAPS*, 2009.

[53] S. Thiébaux, J. Hoffmann, and B. Nebel, “In defense of PDDL axioms,” *Artif. Intell.*, vol. 168, no. 1-2, pp. 38–69, 2005.

[54] G. Brewka, T. Eiter, and M. Truszczynski, “Answer set programming at a glance,” *Commun. ACM*, vol. 54, no. 12, pp. 92–103, 2011.

[55] M. Gebser, B. Kaufmann, R. Kaminski, M. Ostrowski, T. Schaub, and M. T. Schneider, “Potassco: The potsdam answer set solving collection,” *AI Commun.*, vol. 24, no. 2, pp. 107–124, 2011.

[56] F. Buccafurri, N. Leone, and P. Rullo, “Enhancing disjunctive datalog by constraints,” *IEEE Trans. Knowl. Data Eng.*, vol. 12, no. 5, pp. 845–860, 2000.

[57] R. Diankov, “Automated construction of robotic manipulation programs,” Ph.D. dissertation, Carnegie Mellon University, Robotics Institute, August 2010.

[58] “Convexdecomposition module,” Website, 2006, last checked: 06.08.2021. [Online]. Available: http://openrave.org/docs/latest_stable/openravepy/convexdecompositionpy/

[59] “Grasping module,” Website, 2006, last checked: 06.08.2021. [Online]. Available: http://openrave.org/docs/latest_stable/openravepy/databases.grasping/

[60] “Ikfast: The Robot Kinematics Compiler,” Website, 2006, last checked: 06.08.2021. [Online]. Available: http://openrave.org/docs/latest_stable/openravepy/ikfast/

[61] “Kinematicreachability module,” Website, 2006, last checked: 06.08.2021. [Online]. Available: http://openrave.org/docs/latest_stable/openravepy/databases.kinematicreachability/

[62] “Inversereachability module,” Website, 2006, last checked: 06.08.2021. [Online]. Available: http://openrave.org/docs/latest_stable/openravepy/databases.inversereachability/

[63] G. Coruhlu, E. Erdem, and V. Patoglu, “A formal approach to discrepancy generation for systematic testing of execution monitoring algorithms in simulation,” in *2016 IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*. IEEE, 2016, pp. 67–74.

[64] G. Çoruhlu, “Design, control and implementation of cocoa: a human-friendly autonomous service robot,” Master’s thesis, Sabancı University, 2014.

[65] E. Erdem, K. Haspalamutgil, V. Patoglu, and T. Uras, “Causality-based planning and diagnostic reasoning for cognitive factories,” in *Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies Factory Automation (ETFA 2012)*, 2012, pp. 1–8.

[66] Z. G. Saribatur, V. Patoglu, and E. Erdem, “Finding optimal feasible global plans for multiple teams of heterogeneous robots using hybrid reasoning: An application to cognitive factories,” *Autonomous Robots*, vol. 43, pp. 213–238, 2019.