Diagnosis of hidden faults in the RCLL

Marco De Bortoli
Stalin Muñoz Gutiérrez
Gerald Steinbauer
Autonomous Intelligent Systems Group
Institute of Software Technology
Graz University of Technology
Austria

Abstract

The importance of Artificial Intelligence and Flexible Production is increasing in industry. Factories are evolving from static automation to complex autonomous systems to better cope with the challenges introduced by globalization. *The RoboCup Logistics League* (RCLL) was created as a testbed for flexible production of on-demand orders. In such a flexible domain, reliable scheduling requires execution monitoring of actions. In this paper we propose, and experimentally evaluate, a diagnosis to deal with execution faults in the RCLL. The proposed solution is based on the monitoring of the execution state of actions of a temporal plan. Our approach consists of a simple fault model, cascade-faults and a knowledge base, followed by replanning. The experimental results support the use of this principle to face the inconsistencies occurring by a nonobservable fault during the execution of plans.

1 Introduction

In recent years, we have seen a growing interest in AI and Industry 4.0, also known as Flexible Production. It is driven by new demands from industry as a result of several factors, such as globalization, digital transformation, and the ongoing boom in e-commerce and the Internet of things. Within this context, existing logistics and production lines need to evolve in order to face the increasing challenges introduced by the industry. Because of the radical shift in perspective, Flexible Production constitutes a new production paradigm. In contrast to the current static production lines, production sites within this paradigm swiftly adapt production settings and processes to pursue dynamic goals in non-static contexts. At the same time this must be done efficiently while requiring minimal human intervention.

The *RoboCup Logistics League* (RCLL) Competition [1, 2] within the RoboCup initiative [3] exists to provide an appealing showcase for research and teaching in the area of Flexible Production. As motived in [4], for the RCLL benchmarking robot planning and execution in a dynamic and uncertain environment is of particular interest. During the competition, games are played by robots that mimic production steps by means of communication and physical interaction with production stations of different types to fulfil a set of on-demand product orders. Physical interactions between robots and machines consist of delivering or retrieving partially or fully assembled products. Failures frequently occur during the interactions despite the robust hardware and low-level control of robots and stations. In particular, the control software developed by the *Graz Robust and Intelligent Production System* (GRIPS) team from the Graz University of Technology is affected by a problem involving delivery tasks. Often a robot is not able to detect a delivery fault by means of its sensors, i.e. the robot commits to release the product being assembled at the expected delivery place on the station, but instead it drops it to the floor. A fault event like this will manifest later as an inconsistency. In this paper, we propose a proof of concept diagnosis system able to model the problem and that performs a rational handling of delayed faults.

The paper is organized as follows: in Section 2 we discuss the context of our work and related research. In Section 3 the RCLL is introduced. Section 4 presents the system developed by the GRIPS team, used to test the diagnosis system. Then, in Section 6 the rationale of the diagnosis system is explained as well as how diagnosis are used during plan execution. Finally, we present results of a proof of concept implementation in Section 7 and draw some conclusions in Section 8.

2 Related Work

Within the field of Robotics, execution monitoring refers to the capability of a system to identify and classify anomalies. For robot planning, it is fundamentally concerned with the assessment process of the realization of expected relevant effects for every executed step of a plan. Execution monitoring has been studied in industrial control. Pettersson [5] identified four main sources of uncertainties affecting robot behaviour: (1) missing

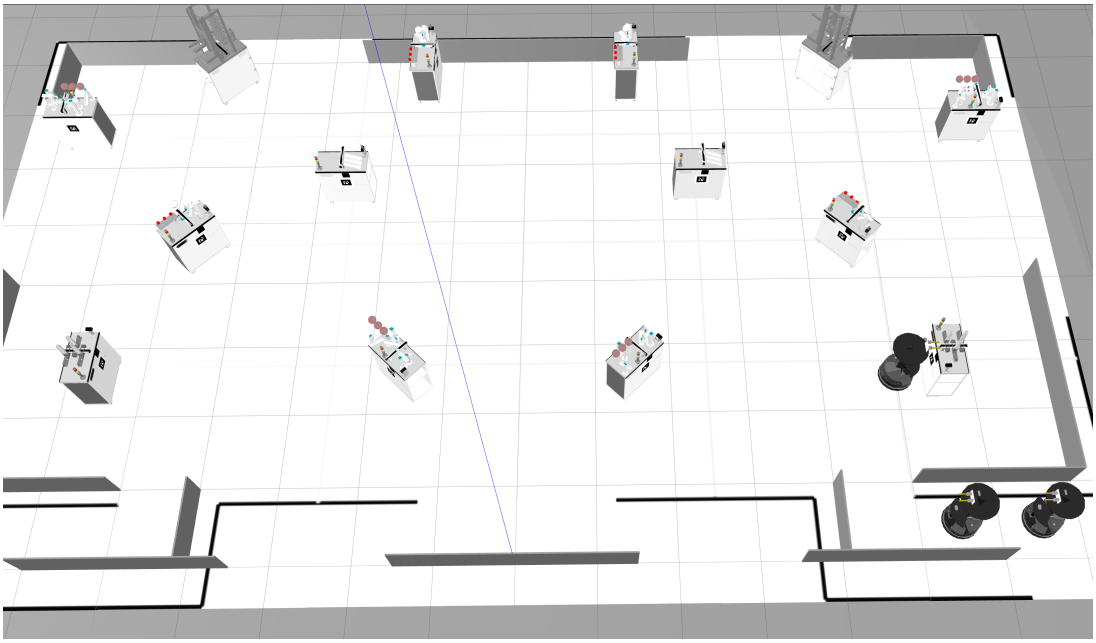

Figure 1: Simulation of the RCLL showing three robots of a single team. Various types of production stations are randomly placed across the playing field. One of the robots is interacting with a production station.

information, (2) unreliable resources, (3) stochastic phenomena, and (4) inherently vague concepts. In this work we designed a diagnosis system that exhibits primarily uncertainty of the types 1 and 2, but to a lesser degree also the last two. Studies within the field of fault detection and isolation (FDI) center on the state of the environment and on the correct operation of the robot inner functional components, however we focus exclusively on high-level deliberation functions.

Chiang et al. [6] distinguished three approaches to execution monitoring: (1) analytical, based on mathematical dynamic equations; (2) data driven, mostly rely on statistical methods; and (3) knowledge-based, comprising expert systems and logic inference based systems. Our approach can be classified as knowledge-based. A recent survey on fault detection and isolation for robotics systems by Khalastchi and Kalech [7] identify a set of five robotic systems characteristics: (1) dependency on exteroceptive sensor; (2) autonomous control; (3) deliberation; (4) dynamic context of operation; and (5) interaction with the environment. Robotic applications may have different degrees of concerns regarding these characteristics. Although for our application several concerns are relevant, we will only address deliberation. For plan execution applications it is common to use model-based reasoning techniques where the model is given by the planning domain definition and the plan itself. This is exactly the context of our work, as we address the dependable execution of the plan to achieve the desired goal.

Reasoning about faults can be handled with the use of history-based diagnosis (HBD), a technique that builds on Situation Calculus [8]. In HBD, action variations can relate to what henceforth we refer as *fault modes*, a description of the possible manners an action can be unaccomplished, i.e. where the nominal effects of an action are not fully present after execution. In [9], Gspandl et al. show the effectiveness of history-based diagnosis for the development of a belief management system implemented in IndiGolog. However, as reported by the authors the computational complexity of the approach is high. Additionally the knowledge base is implicit in the FOL representation, making it difficult the separation of concerns at the design phase of a monitoring system. Our exploration is preliminary and aims to find a performant solution that exhibits the soundness of their approach and from which further improvements can derive.

3 RCLL

The RCLL as part of the RoboCup initiative aims to stimulate the development of Robotics and Artificial Intelligence by means of robotics competitions. Researchers have access to highly standardized complex challenges, where they can evaluate their systems. This helps to bridge the gap between academy and industry, compelling academics to acknowledge and take into consideration the complexity of real world applications. In this

league the goal of each team of robots is to cooperate with a set of production machines to assemble and deliver goods on demand. Two competing teams share a common factory floor of $14m \times 8m$ in size. Figure 1 shows a visualization of a game using the robot simulator Gazebo [10]. Each team is comprised of up to 3 autonomous robots and owns 7 machines, emulated by Modular Production Systems (MPS®) provided by Festo [11]. There are different types of machines that resemble different production steps like fetching raw material, assembling parts, or delivering finished products. The main task of a team is to develop methods that coordinate their mobile robots and static machines. Robots and machines communicate using WiFi. Robots within the same team need to interact physically with their machines, e.g. fetching parts from a dispenser machine or providing intermediate products to the machines that refine the products. Usually teams recourse to a central team server that collects information from the production management system, the machines, and the robots, and at the same time, coordinates the tasks executed by the robots. The products are mimicked by stacks of bases, rings, and caps. More specifically, each product features one base, one cap, and from 0 to 3 intermediate rings. The amount of rings determines the complexity of the product. The complexity is labeled with the letter C followed by a number, from C0 to C3, where the number specifies the quantity of requested rings. In general, several refining steps of intermediate products by different machines are needed to obtain a product. A central agent randomly generates product orders with varying configurations and delivery time windows. These orders are communicated to the teams' server that needs to derive a production schedule and to distribute the tasks among the robots and machines. Teams are primarily awarded points for successfully delivering finished products. Products of higher complexity are awarded more points than products with lower complexity. As a reference, for assembling a C3 product the execution of up to 10 different intermediate steps is necessary. Some of the intermediate steps allow concurrent execution or can be reschedule in real time in order to optimize the awarded points. Teams attempt strategies that maximize the awarded points in a production phase that lasts 17 minutes. Delivery time windows are considered as soft constraints and teams delivering a product outside its required time window are awarded partial points for the achievement. Points are also awarded for each successfully accomplished intermediate assembling step, independently of whether or not the robot delivers the finished product.

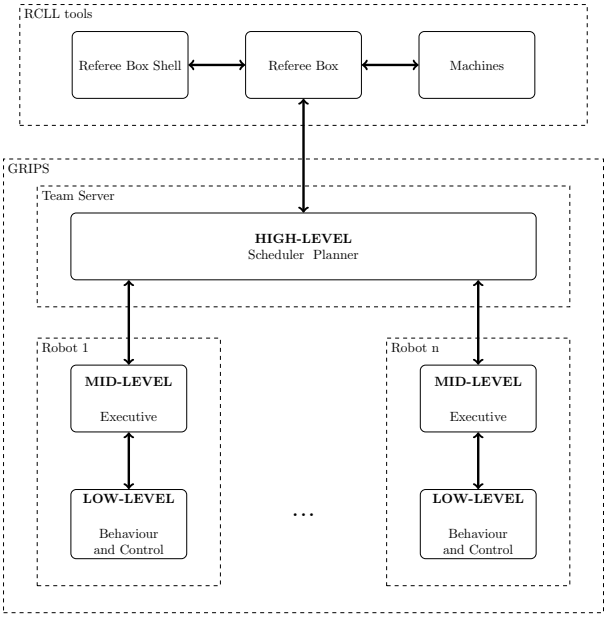

Figure 2: GRIPS distributed software architecture.

4 System Architecture

In this section we describe the software architecture developed by the GRIPS team to contend at the RCLL. Figure 2 summarizes the two types of components within the architecture: (1) the RCLL tools provided by the organizers, and (2) the main architectural subsystems comprising the GRIPS software stack. As can be seen in the diagram, the architecture is distributed: each of the three robot modules, as well as the *Team Server* (TS), are deployed across different computers. The Referee Box (RefBox) is responsible for the automated product orders generation and for awarding points to the teams.

The GRIPS software stack resembles an abstraction pyramid and is divided into three main layers. The high-level layer includes the *Scheduler* and the *Planner*. It also serves as a coordination hub between the three mobile robots and other agents like the RefBox. The implementation of this layer is based on Java technologies [12]. The middle-layer is responsible for reactive behaviour of each robot and is deployed onboard. It receives task goals from the Scheduler, and it refines them into subtasks with smaller granularity using the procedural reasoning system OpenPRS [13]. The low-level layer consists of the robot platform, i.e. sensing, low-level monitoring and low-level control of actuators as well as basic skills such as localization and navigation. It is implemented in C++.

The architecture of the high-level is depicted in Figure 3. The *Central Management and Monitoring Module* (CMMM) decides what other modules

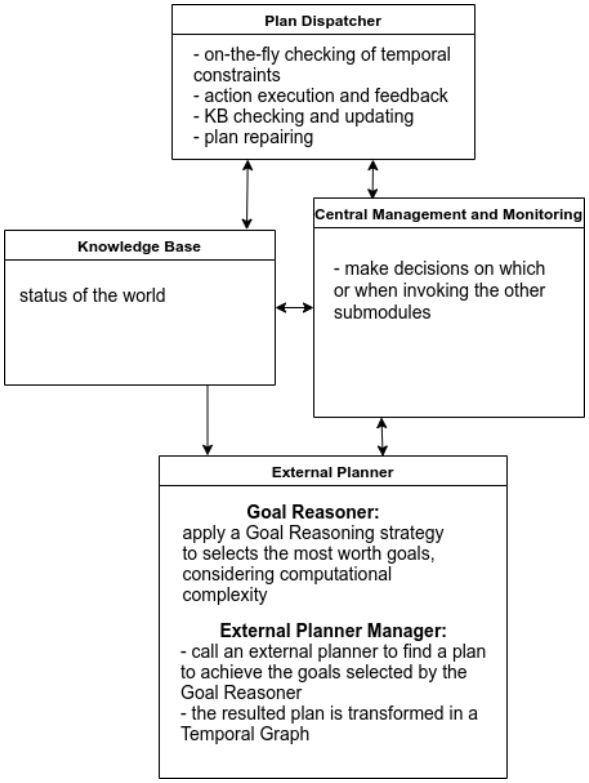

Figure 3: GRIPS high-level layer

to invoke depending on the current situation, and the feedback it receives from the *Plan Dispatcher* (PD) or the Refbox. The situation is kept in the *Knowledge Base* (KB) using a set of Atoms.
The architecture is designed to be flexible and modular so that strategies for decision making can be easily changed without modifying the entire structure. A key decision to make by the CMMM, is choosing when to call the external planner to recompute a new plan based on a planning model. A planning model is specified in two parts: 1) a planning domain, and 2) a planning problem, both expressed using Planning Domain Definition Language (PDDL) [14, 15]. For our experiments, we use a external planner, see [16] , although any other PDDL-compatible stand-alone planner can be used. Another important decision consists of selecting which goals to pursue. The *Goal Reasoner* (GR) selects the most promising goals. Once the goals are generated, a PDDL problem file is created dynamically according to the goals and the current KB. At this point, the external planner is invoked using as inputs a static PDDL domain file (writen by a human at design time), and the dynamically generated PDDL problem file. If the problem is solvable, a temporal plan is returned by the planner. The temporal plan is then executed by the PD. The PD also represents the connection point with the instances of the Middle-Level *Executive* (EX) of the general architecture. EX instances, one for each physical robot, are implemented in OpenPRS. Each EX accept two types of abstract tasks: (1) *GET* tasks and (2) *DELIVER* tasks. Tasks have parameters, for example the production station responsible for the assembly or the type and color of a requested piece. The PD assigns tasks to the specific robot using its unique identifier and gets feedback on the success or failure of the executed task from the point of view of the robot. This observation is important, as there are situations in which the robot is not able to detect a failed execution due to its partial knowledge of the situation or due to limited sensing capabilities. In particular, it may happen that the robot does not succeed in delivering a piece to a station and the piece drops on the floor. As there are no sensors able to detect it, the robot is not aware of the issue. If such failed actions are not detected they may lead to an inconsistency between the state of the environment and its internal representation. Our strategy is a preliminary approach towards giving the robots the capability of diagnosis of chains of faults for knowledge base repair and better decision making. Every time an inconsistency or a problem is detected, this module tries to repair the KB. If not possible, a replan is called by the central module. The repairing strategy is designed as an independent module, increasing the flexibility of the software architecture.
The PD, which is the most complex module, updates the KB accordingly to action executions. Thus, if the planner is called again, it relies on a up-to-date description of the current state of the environment. Logic inconsistencies in the KB are also detected. This has proven helpful when a problem happens during dispatching.

5 Simulated assembly of a simple product

We will describe and test our diagnosis and KB repair module using the simulated assembly of a C1 product. Products of complexity type C1 consist only of a base, a ring, and a cap. Figure 4 illustrate the dependency graph for high-level tasks required to assemble a C1 product. We consider here only the two high-level tasks: *GET* and *DELIVER*. Robots need to handle resources, as well as partial or finished products, generically called items. All machines operate in various modes, meaning that for many operations each machine should be appropriately prepared before physically interacting with it. The *GET* task acquires an item from a specific machine. It requires the robot to move to the machine where the item is located, align to the correct side of the machine and grasp the item. The DELIVER task will transport the item that has been grasped to a destination

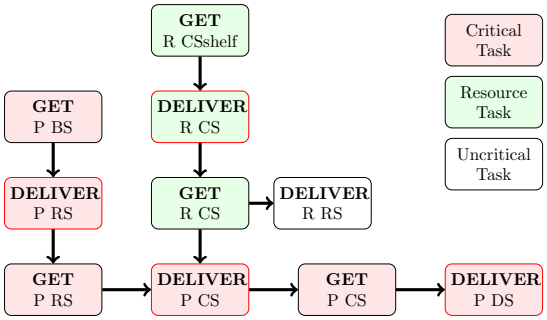

Figure 4: Dependency graph for tasks required to build and deliver a *C1* product order. Arrows represent ordering constraints for the tasks as well as enabling preconditions. Each tasks describe the involved piece (P - product, R - resource), its type and the used station (Base Station, Ring Station, Cap Station or Delivery Station). The tasks with the red border are the ones affected by the dropping piece problem.

machine.

6 Solution Architecture

Deliberation functions for cognitive robot architectures provide a well established and tested paradigm for designing robotic applications [17]. The diagnosis principle described in this section was implemented in the high-level layer of the GRIPS architecture. For completeness and context of the presented solution, we briefly mention some relevant aspects of the other layers.

Diagnosis at the robot platform level can benefit from a broad range of diagnosis techniques and algorithms. The diagnosis modules at this level of abstraction are mostly concerned with the correct detection and isolation of faulty hardware and software components. We won't address this layer in the present work and will focus exclusively on the EX and the deliberative levels. The EX is implemented in *OpenPRS* which is an open source version of the Procedural Reasoning System (PRS) [18, 19]. It receives the tasks dispatched by the TD and executes them in an episodic manner. The EX does not have context of the plan ahead, nor explicit memory about past actions. The design of the EX includes the refinement of the high-level tasks into a library of procedures that orchestrates the request of commands at the platform level. This orchestration implements reactive behavior that add to dependability by means of corrective strategies, such as reattempts of failed subtasks and timeout monitoring. The procedural library of the EX considers reactive behaviour that compensates for execution errors of platform commands. We call this procedures *fail-safe* actions. They can be considered as conditional reattempts that check the state of the execution and the outcome of commands.

We briefly explain here the internal representation of the plan, and refer the reader to [20] for a formal presentation of the topic. Our external planner is the temporal planner *optic* [16] which is based in POPF [21]. Following the least commitment principle, the solutions or plans provided by these planners do not enforce a strict total-order of the planned actions nor total concretization of the parameters of such actions. These features are highly valuable and ideal for the GRIPS architecture where the EX reactions in response to the environment give rise to high variability in execution times. Our temporal plan has to satisfy consistency of conditions and effects occurring at the start, during, and at the end of the execution of an action. Families of solutions can be represented as classical partial-order plans for example by splitting each step of the plan into 3 different steps. Additional constraints are added to satisfy group selection by the planner, see for example [22]. In particular, for our planning domain we can simplify further the transformation and work only with start and end conditions and effects. Therefore we can think of the satisfying temporal plan as a partial-order plan where each temporal action is split into two actions. This constitutes a simple temporal constraint network [23] where vertices are the split temporal steps and edges that connect them are labeled with temporal constraints. The GRIPS architecture is loosely coupled between the TS and the three EX instances running in each robot. The planner selects a binding to a specific robot, as well as all other parameter of the request, except for the specific time of dispatching. This keeps the plan flexible and feasible even though the real execution times will vary significantly. The time variance is due to the uncertainty in the application domain, the reactive behaviour of the executive trying to achieve the task, as well as the specific state of the world. The scheduler will decide on the next task to execute. Then the task is dispatched. After dispatching the tasks, a report message will be send back by the EX after finalizing the execution of the task. The report, for this experiments is limited to a binary answer $\{success, \neg success\}$, a failed task won't report any further diagnostic information.

Our fault-tolerant plan execution strategy is presented in Algorithm 1. The diagnosis is performed based on the plan execution history X which is updated in every iteration, the planning model M, the task that has failed, as well as the partial observations of the environment. The result of the diagnosis is a *new history* that explains what has

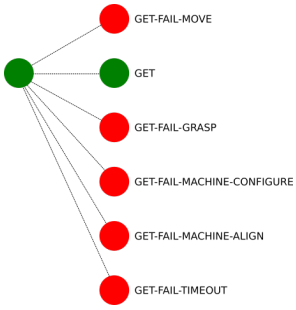

Figure 5: Fault Modes for task *GET*.

happened. The new history of tasks is *replayed* against the initial KB to progress it to the present time. This constitutes a repair in the KB that is consistent with the diagnosis.

Algorithm 1: Fault-tolerant plan execution

input : M: Planning Model,
P: Plan,
KB_0: Starting Knowledge Base
output: success or failure

```
KB ← KB_0 ;                                  ▷ working knowledge base
X ← [ ] ;                                    ▷ empty list of executed tasks
while KB ⊭ P.goal do
    if ¬ validate(P,KB) then
        return failure ;                     ▷ replanning is required
    end
    P,t ← schedule(P,KB) ;                   ▷ task:t, P:plan
    r,o ← dispatch(t) ;                      ▷ r:result,o:observation
    X' ← X + [(t,o)] ;                       ▷ temporary execution history
    if r = success ∧ consistent(P,X,KB)
     then
        KB ← update-KB(KB,t,o) ;             ▷ update KB
    else
        X' ← diagnose(M,X,t,o) ;
        KB ← repair-KB(X',KB_0) ;
    end
    X ← X' ;                                 ▷ update execution history
end
return success ;                             ▷ goal achieved
Procedure repair-KB(X, KB_0)
    KB ← KB_0 ;
    foreach (t,o) ∈ X do
        KB ← update-KB(KB,t,o) ;
    end
    return KB ;                              ▷ KB consistent with explanation
```

Our solution to the diagnosis of hidden faults relies on extending the planning model to include fault modes. Informally, for every high-level action (we referred here as tasks) in the planning domain we can define a set of task variants representing all the known possible faults of a task.
In Figure 5 we show different causes of failure for the action *GET*, a fault mode can be defined for each case. Figure 6 illustrates an example by depicting an hypothetical plan execution sequence. Tasks have the time step as subindex. In this example task t_n has failed. Fault modes of each executed task are indicated by circunflexing the task variable: $\hat{t}_j^i$ means the fault mode i for task t_j. Let us suppose that modes $\hat{t}_n^1$ and $\hat{t}_n^2$ are both possible because observations do not allow to single out a specific fault. For this example several diagnosis are possible: (1) the first two, $[t_{n-2}, t_{n-1}, \hat{t}_n^1]$, $[t_{n-2}, t_{n-1}, \hat{t}_n^2]$ correspond with the most simple explanations possible, i.e. that the only fault that has occurred is at time n; (2) a diagnosis $[t_{n-2}, \hat{t}_{n-1}^1, \hat{t}_n^1]$, that is the case we are interested in, it contains a causal dependency between two fault modes; and (3) two more cases, $[\hat{t}_{n-2}, t_{n-1}, \hat{t}_n^1]$, $[\hat{t}_{n-2}, t_{n-1}, \hat{t}_n^2]$ involving other combinations of fault modes and successful tasks. At this point a strict logical inference leading to uniquely identify the cause of the problem is impossible with the knowledge base of the agent due to partial observability of the effect of actions that lead to several possible diagnosis. An enhanced representation should capture the preference towards more likely explanations, independently of whether they are simple explanations or complex combinations of successful actions and fault modes. Likelihood of fault modes can be captured by probabilities. It follows from the nature of logistics tasks, that actions are dependent on each other and therefore faults that are analyzed based only on independent probabilities seem insufficient. Conditional probabilities for task (or fault mode) transitions can be a good start. In our example, we can expect the nominal execution to have a higher likelihood than any of the fault modes. If the fault dependency between $\hat{t}_{n-1}^1$ and $\hat{t}_n^1$ is modelled with certainty, then the fault chain explanation will be preferred if the likelihood of $\hat{t}_{n-1}^1$ given t_{n-2} is higher than the likelihood of $\hat{t}_n^1$ given t_{n-1}. We have an analogous reasoning for the hidden fault in our problem of the dropping piece.

7 Architecture Evaluation

To evaluate our diagnosis system, we performed a comparison with a previous version of the GRIPS architecture that does not include the fault-tolerant plan execution explained in the last section. In the previous version, replanning is performed every time a task fails. The planning algorithm relies on the state of the environment represented in the KB at the moment the fault occurs. If the KB is inconsistent with the real world, and this inconsistency is not detectable by direct sensing, the new plan will be most likely un-

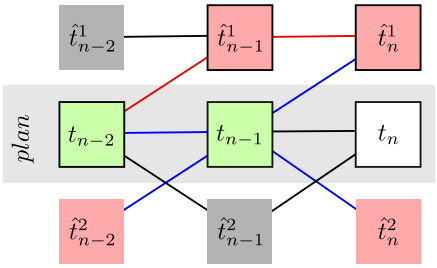

Figure 6: A fault model execution diagram example. The nominal tasks (task in the plan) are in the light gray box. Vertically aligned to each task are their fault modes. Task t_n has failed. Several fault modes are consistent with the KB, all are colored light red. This leads to several explanations: $\{\ [t_{n-2}, t_{n-1}, \hat{t}^1_n],\ [t_{n-2}, t_{n-1}, \hat{t}^2_n],\ [t_{n-2}, \hat{t}^1_{n-1}, \hat{t}^1_n],\ [\hat{t}_{n-2}, t_{n-1}, \hat{t}^1_n],\ [\hat{t}_{n-2}, t_{n-1}, \hat{t}^2_n]\}$.

feasible leading to plan execution failure. When this mismatch happens, recovery is not possible without the help of a diagnosis system. The goal of this evaluation is to measure how effective is our diagnosis system dealing with the dropping piece problem. In our experimental setting, we randomly generated 10 games and simulated each game against the two versions of the GRIPS architecture. To have a more controlled experiment, games were modified to place only three C1 product orders. In this way we increase the potential number of orders delivered in a game and facilitate the quantification of the results. An additional reasons for not including C3 products in the evaluation is the observation that building a complex product consumes almost all the time available in one game, making it almost impossible to recover from a failure of a C3 product if it happens near the end of the game. The evaluation has been performed using the simulation software Gazebo, able to reproduce the physics and interactions of real games. We summarize our results in Table 1.

We can conclude from the evaluation that diagnosis lead to better decisions. In our case the diagnosis allows to repair the KB. In the first run of the games, it is able to deliver 24 products. However, the dropping piece problem affects the overall performance even with the fault-tolerant plan execution, as can be seen from the second run. It seems apparent that although we have corrective measures in place at the deliberation layer, improvements must be also done in the two other layers.

8 Conclusions and Future Work

In this paper we proposed a proof of concept diagnosis able to deal with a systematic problem

Game	W\O DX			W DX		
	r1	r2	tot	r1	r2	tot
g1	3	1	4	2	3	**5**
g2	3	1	**4**	3	1	**4**
g3	1	1	2	1	2	**3**
g4	3	2	5	3	3	**6**
g5	0	1	1	3	1	**4**
g6	3	3	**6**	3	0	3
g7	3	3	**6**	3	1	4
g8	1	0	1	3	3	**6**
g9	1	2	3	1	3	**4**
g10	0	0	0	2	1	**3**
total	18	14	**32**	24	18	**42**

Table 1: Number of delivered C1 products by the two versions of the GRIPS architecture for 10 different randomly generated games. Each game was ran twice. Runs are labeled as r1 or r2. The total amount of delivered products for each game is shown in the column labeled tot.

affecting the GRIPS software in the RCLL. In our approach, diagnosis is possible, even when there is limited feedback from lower abstraction layers in the system architecture. The dropping piece problem affecting our software is not immediately detectable by the robot. There are occurrences of hidden faults, that manifest themselves with a delay. If no history of the execution of actions is available the manifested fault may lead the system to conclude a false belief, hindering decision making. In this scenario diagnosis is ineffective and only based on a local scope. For this reason, we implemented a proof of concept experimental setting that extends a plan model with fault modes and fault mode dependencies. If a *DELIVER* action fails, involving a specific production station, the history of plan execution steps involving the same station is analyzed. If the inference can be made to the root cause, then a KB repair mechanism takes place. If the KB repair is successful then replanning is more likely to tolerate the fault and eventually achieve the goal. As shown in our experimental result in Section 7, our model is able to detect and correct the problem several times, by calling a replanning phase which attempts to rebuild the lost product. Without such diagnosis, typically a robot remains blocked on a production station, trying to retrieve an object which is no longer there.

As future work, we plan to develop further our diagnosis approach and apply it to correct other planning execution problems, for example, localization and path finding problems of the architecture of the GRIPS software in the RCLL. Diagnosis at the EX and platform level are also necessary. Integrating a monitoring system that spans

the different levels of abstraction and is used for rational decision making would also be highly desirable. From the research perspective, one open question to explore would be the online learning of fault modes based on experience gained by the execution of plans.

9 Acknowledgments

This work is possible thanks to the LEAD project "Dependable Internet of Things in Adverse Environments", Funded by Graz University of Technology.

References

[1] RoboCup. RoboCup Logistics League. https://ll.robocup.org/, 2021. Accessed: 2021-08-19.

[2] Tim Niemueller, Sebastian Zug, Sven Schneider, and Ulrich Karras. Knowledge-based instrumentation and control for competitive industry-inspired robotic domains. *KI - Künstliche Intelligenz*, 30, 08 2016.

[3] Gerald Steinbauer and Alexander Ferrein. 20 Years of RoboCup. *Künstliche Intelligenz*, 30(3-4):221–224, 2016.

[4] Marco De Bortoli and Gerald Steinbauer. The robocup logistics league from a planning perspective. In *ICAPS 2020*, 2020.

[5] Ola Pettersson. Execution monitoring in robotics: A survey. *Robotics and Autonomous Systems*, 53(2):73–88, 2005.

[6] Leo H Chiang, Evan L Russell, and Richard D Braatz. *Fault detection and diagnosis in industrial systems*. Springer Science & Business Media, 2000.

[7] Eliahu Khalastchi and Meir Kalech. On fault detection and diagnosis in robotic systems. *ACM Computing Surveys (CSUR)*, 51(1):1–24, 2018.

[8] Gero Iwan. History-based diagnosis templates in the framework of the situation calculus. In *Annual Conference on Artificial Intelligence*, pages 244–259. Springer, 2001.

[9] Stephan Gspandl, Ingo Pill, Michael Reip, Gerald Steinbauer, and Alexander Ferrein. Belief management for high-level robot programs. In *Twenty-Second International Joint Conference on Artificial Intelligence*, 2011.

[10] Open Source Robotics Foundation. Gazebo. http://gazebosim.org/, 2014. Accessed: 2021-08-15.

[11] Festo. MPS® – the modular production system: From module to learning factory. https://www.festo-didactic.com/int-en/learning-systems/mps-the-modular-production-system/102775.htm, 2021. Accessed: 2021-08-12.

[12] Oracle. Oracle java technologies. https://www.oracle.com/java/technologies/, 2021. Accessed: 2021-08-15.

[13] Felix Ingrand. OpenPRS. https://git.openrobots.org/projects/openprs, 2021. Accessed: 2021-08-15.

[14] Patrik Haslum, Nir Lipovetzky, Daniele Magazzeni, and Christian Muise. An introduction to the planning domain definition language. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 13(2):1–187, 2019.

[15] Damien Pellier and Humbert Fiorino. Pddl4j: a planning domain description library for java. *Journal of Experimental & Theoretical Artificial Intelligence*, 30(1):143–176, 2018.

[16] J Benton, Amanda Coles, and Andrew Coles. Temporal planning with preferences and time-dependent continuous costs. In *Twenty-Second International Conference on Automated Planning and Scheduling*, 2012.

[17] Félix Ingrand and Malik Ghallab. Deliberation for autonomous robots: A survey. *Artificial Intelligence*, 247:10–44, 2017.

[18] François Félix Ingrand, Raja Chatila, Rachid Alami, and Frédéric Robert. Prs: A high level supervision and control language for autonomous mobile robots. In *Proceedings of IEEE International Conference on Robotics and Automation*, volume 1, pages 43–49. IEEE, 1996.

[19] Rachid Alami, Raja Chatila, Sara Fleury, Malik Ghallab, and Félix Ingrand. An architecture for autonomy. *The International Journal of Robotics Research*, 17(4):315–337, 1998.

[20] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: theory and practice*. Elsevier, 2004.

[21] Amanda Coles, Andrew Coles, Maria Fox, and Derek Long. Forward-chaining partial-order planning. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 20, 2010.

[22] Keith Halsey. Temporal planning with a non-temporal planner. In *Newsletter of the European Network of Excellence in AI Planning*, number 7, 2003.

[23] Rina Dechter, Itay Meiri, and Judea Pearl. Temporal constraint networks. *Artificial intelligence*, 49(1-3):61–95, 1991.