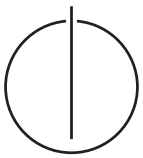
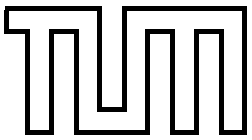


Institut für Informatik
Technische Universität München



Self-diagnosing Agent: Tight Integration of Operational Planning and Active Diagnosis

Dissertation

Lukas Daniel Kuhn

TECHNISCHE UNIVERSITÄT MÜNCHEN

Lehrstuhl für Bildverstehen und wissensbasierte Systeme
Institut für Informatik

Self-diagnosing Agent: Tight Integration of Operational Planning and Active Diagnosis

Lukas Daniel Kuhn

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Alfons Kemper, Ph.D.
Prüfer der Dissertation:	1. Univ.-Prof. Michael Beetz, Ph.D.
	2. Johan de Kleer, Ph.D.

Die Dissertation wurde am 13.08.2010 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 12.11.2010 angenommen.

Acknowledgements

First of all, I owe my deepest gratitude to my advisor Johan de Kleer. Johan has been a great source of inspiration and I cannot thank him enough for all the endless conversations that enabled me to develop a deep understanding of the subject. I especially thank Johan for his patient and openness during our conversations.

I would like to express my special gratitude to my advisor Prof. Michael Beetz, Ph.D. Michael's support and guidance have been essential to this journey. I would like to say a special thank you for your willingness to communicate with me at all times, especially in the evenings and at the weekends.

This thesis would not have been possible without my great colleagues at the Palo Alto Research Center with whom I had the pleasure to interact with: Johan de Kleer, Bob Price, Julia Liu, Tim Schmidt, Serdar Uckun, John Hanley, Minh Do, Rong Zhou, Lara Crawford, Haitham Hindi, Ying Zhang, Gabe Hoffman, and Sungwook Yoon. Only their encouragement, guidance, inspiration, enthusiasm, and support from the initial to the final level enabled my accomplishments. In particular, I would like to express my gratitude to Bob Price, whose valuable feedback encouraged me to greater work. Furthermore, I would like to show my gratitude to Julia Liu. Julia has support me in shaping initial ideas into solid contributions.

My special thanks belong to Manuela Fischer for her great administrative support during the submission process.

Last, but certainly not least, I like to thank my wife Vanessa Cirannek. Your commitment to support this thesis was of great importance to me. Your understanding gave me the freedom to go on this journey and I am very thankful for that. Bausi, I love you Vanessa!

Lukas Kuhn, Palo Alto, CA, August 2010

Abstract

Artificial intelligence has long been inspired by the vision of autonomous systems that not only act on the larger world, but also maintain and optimize themselves. Autonomous systems perform desired tasks to achieve desired goals continuously over a long period of time without external guidance or intervention. This requires that autonomous systems know about their true capabilities (model), reason about their course of action with respect to their current conditions (planning), and reflect on their actual behavior to determine their current conditions (diagnosis).

In general, long-life autonomy can be seen as a combination of two methods: (1) diagnosis to determine the current condition of the system and (2) planning to optimize system operation for the current condition. A step towards long-life autonomy is to automate diagnosis.

The integration of diagnosis into regular system operation is typically realized by one of the following approaches: Alternating between explicit diagnosis and regular operation or simultaneous execution of passive diagnosis and regular operation. Alternating phases often result in long periods during which regular operation must be suspended. This is particularly true, when diagnosing complex fault scenarios, such as faults that occur intermittently. The combination of passive diagnosis with regular operation is often unsuccessful, as regular operation may not sufficiently exercise the underlying system to isolate the underlying fault.

This work introduces a new architecture, coined a *Self-diagnosing Agent*, which realizes the integration by a novel diagnosis paradigm called *pervasive diagnosis*. Pervasive diagnosis actively manipulates the course of action during operation in order to gain diagnostic information without suspending operation. Consider a system where operational goals can be achieved in multiple ways. This flexibility can be exploited to generate operational plans that simultaneously gather information by trading off information gain objectives with performance objectives. Therefore active diagnosis and regular operation occur at the same time leading to higher long-run performance than an integration of regular operation with passive diagnosis or alternating between explicit diagnosis and regular operation.

The main contribution is an overall framework, which tightly integrates regular operation and active diagnosis. In particular, this contribution introduces an information criterion that defines how informative a plan is, a strategy to derive informative operational plans, and a diagnosis framework to efficiently update the belief states. The overall framework is optimized for systems with faults multiple in number, intermittent in appearance and potentially caused by hidden interactions. As a result, systems that embody the Self-diagnosing Agent architecture benefit from active diagnosis during regular operation leading to a higher, more reliable, and more robust long-run performance.

Kurzfassung

Autonome Systeme sind Systeme die eigenständig Aktionsfolgen bestimmen die ihre Ziele erreichen. Während autonome Systeme bereits in verschiedensten Bereichen Anwendung finden, stellt die Langzeitautonomie immer noch eine ungelöste Herausforderung dar. Diese Dissertation greift diese Herausforderung auf und untersucht eine Kombination von zwei Methoden: Diagnose zur Bestimmung des aktuellen Zustandes und Planung zur Generierung von bestmöglichen Aktionsfolgen basierend auf dem diagnostizierten Zustand. Durch eine neuartige Integration von Planung und Diagnose kann der Informationsgewinn als Optimierungsfaktor in die Generierung von Aktionsplänen aufgenommen werden. Das resultierende Gesamtsystem profitiert von Synergieeffekten zwischen Planung und Diagnose die zu einer gesteigerten Langzeitsystemleistung führen.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

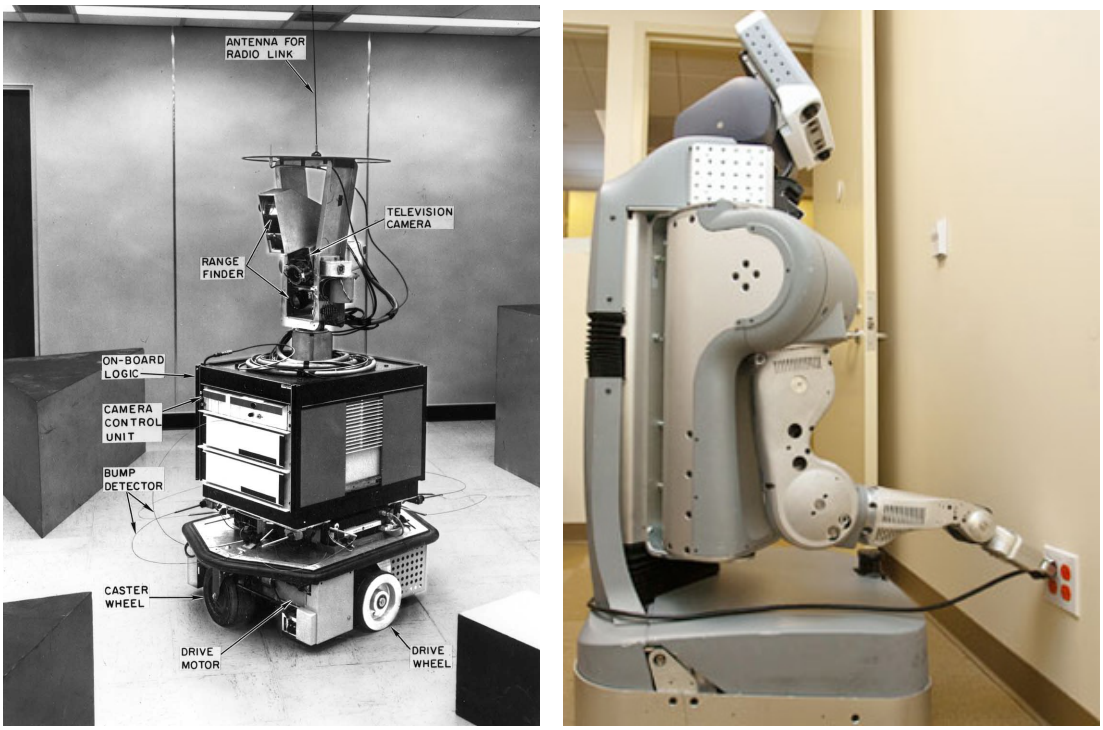
> Follow effective action with quiet reflection. From the quiet reflection will come even more effective action.
>
> *(James Levin:* Reflection*)*

1.1 Introduction

Artificial intelligence has long been inspired by the vision of autonomous systems that not only act on the larger world, but also maintain and optimize themselves. In its ambition, an autonomous system achieves desired goals continuously over a long period of time without external guidance or intervention. Such systems are especially beneficial when jobs are too dangerous, dirty, or dull for humans or require repetitive operation with great accuracy and reliability. Autonomous and semi-autonomous systems are already used in a wide range of domains such as manufacturing, assembly, packing, logistic, transport, earth and space exploration, surgery, weaponry, and laboratory research (Christensen, Batzinger, Bekris, Bohringer, Bordogna, Bradski, Brock, Burnstein, Fuhlbrigge, Eastman, Edsinger, Fuchs, Goldberg, Henderson, Joyner, Kavaraki, Kelly, Kelly, Kumar, Manocha, McCallum, Mosterman, Messina, Murphey, Peters, Shephard, Singh, Sweet, Trinkle, Tsai, Wells, Wurman, Yorio & Zhang 2009). However, most of today's deployed systems are centered around short-life autonomy. A system embodies short-life autonomy if it has the ability to achieve desired goals without external guidance or intervention, but not necessarily continuously over an extended period of time.

An early example for short-life autonomy is the mobile robot named Shakey (Nilsson 1984), illustrated in Figure 1.1(a), which was introduced in August 1967 by the Artificial Intelligence Center at SRI International (then Stanford Research Institute). Shakey performed tasks that required planning, navigation, and the rearrangement of simple objects by leveraging its limited ability to perceive and model its environment. Shakey was not designed to perform those tasks continuously over an extended interval of time. For instance it lacked the ability to autonomously calibrate itself, to recover from failures, or to adapt to changes. Nevertheless, demonstrating short-life autonomy at this early time, even if the system had to be restarted before each experiment, was already a great achievement. Today's standards are higher. Various research programs shifted their focus to emphasize long-life aspects of autonomy. In 2007, the robotic research lab Willow Garage introduced its second mobile robot platform, named PR2 (illustrated in Figure 1.1(b)), and stated long-life autonomy for personal robotics as one of their core research objectives. In 2009, McGann, Berger, Bohren, Chitta, Gerkey, Glaser,

Marthi, Meeussen, Pratkanis, Marder-Eppstein & Wise (2009) and Marder-Eppstein, Berger, Foote, Gerkey & Konolige (2010) demonstrated PR2's ability to reliably navigate inside an office environment, open doors, and plug into regular outlets over a total distance of a full 26.2 miles marathon. Even though, the individual tasks were demonstrated before, performing the same tasks continuously over a long period of time without any external intervention was very challenging. Whereas short-life autonomy often neglects the occurrence of rear events, assumes correct calibration, and relies on the availability of a fully functional system, long-life autonomy requires the ability to adapt to changes, and robustly and reliably plan the course of action even in unforeseen situations.



(a) Shakey: One of the first (short-life) autonomous mobile robots.

(b) PR2: (Long-life) autonomous mobile robot plugging itself into a regular outlet.

FIGURE 1.1 Autonomous mobile robot platforms: Shakey and PR2.

Another program that focuses on long-life autonomy, particular in the field of autonomous driving, is the DARPA Grand Challenge created in 2004 by the Defense Advanced Research Projects Agency (DARPA). The DARPA Grand Challenge is a series of competitions targeted to develop ground vehicles that navigate and drive entirely autonomous. It pursues the goal to keep war fighters off the battlefield and out of harms' way (DARPA 2004). Figure 1.2(a) shows Stanford's winning autonomous ground vehicel, named Stanley, passing the finishing line (Thrun, Montemerlo, Dahlkamp, Stavens, Aron, Diebel, Fong, Gale, Halpenny, Hoffmann, Lau, Oakley, Palatucci, Pratt, Stang, Strohband, Dupont, Jendrossek, Koelen, Markey, Rummel, van Niekerk, Jensen, Alessandrini, Bradski, Davies, Ettinger, Kaehler, Nefian & Mahoney 2006). During the race in 2005, Stanley drove completely autonomously a total distance of approximately 175 miles in 6 hours and 54 minutes by navigating through off-road

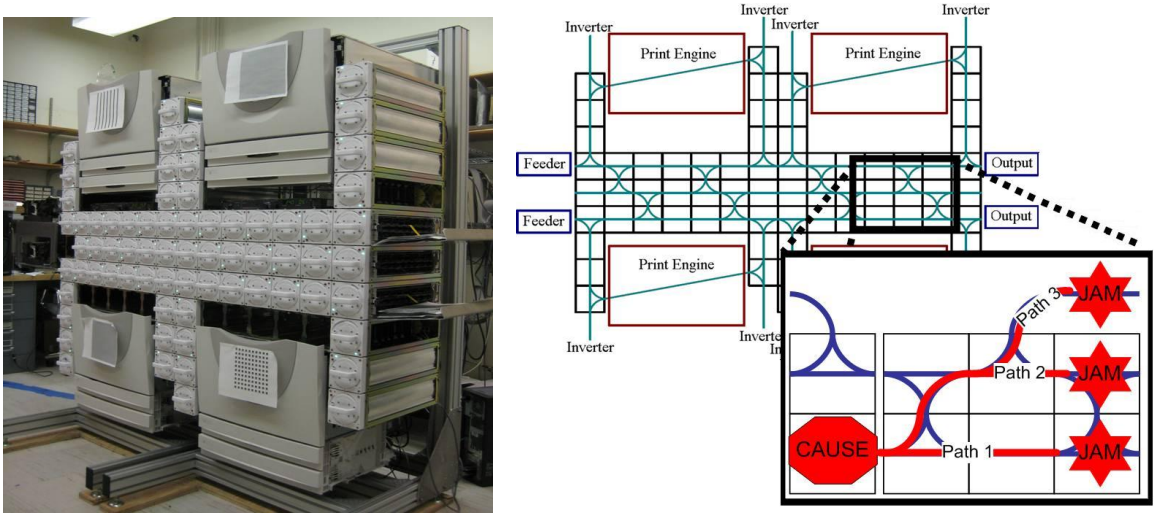
terrain and taking sharp turns as shown in Figure 1.2(b). A key element of Stanley's success was its extremely robust and reliable perception and modeling capability, which enabled very accurate models of the environment. These models were then leveraged by a planner to consider possible futures before committing to a course of action. A prerequisite of planning is the availability of models that accurately describe the capabilities of the system and the environment it is operating in. However, in a real-world setting accurate models might not be available. A system model can be incomplete or obsolete due to the system breaking, evolving over time, or experiencing unforeseen situations. As a consequence, a plan which is valid with respect to the original model might cause exceptions during execution. A great example is the autonomous vehicle from the Carnegie Mellon University, named H1ghlander, which participated in the 2005 DARPA Grand Challenge. Despite its high reliability, the system experienced an exception during the race causing the vehicle to slow down and ultimately leading to Stanley's victory (Thrun et al. 2006).

(a) Stanley at the finish line, winning the race.

(b) Stanley navigating though the terrain including sharp turns.

FIGURE 1.2 Autonomous ground vehicle named Stanley at the 2005 DARPA Grand Challenge.

Both previously introduced programs, personal robotic and autonomous driving, led to a tremendous advancement in the field of long-life autonomy. However, the advancements were mainly achieved through increased system reliability to avoid unforeseen situations such as exceptions or failures. Despite the increased reliability, systems eventually break, exceptions occur, and unforeseen events happen. This is especially true when systems have to be controlled and operated over a very long time. For instance, the Voyager 1 and 2 spacecrafts have each been controlled and operated successfully in space for more than 27 years and the Galileo spacecraft for 14 years. Even though all three unmanned spacecrafts were designed to be highly reliable, a total of 3,300 exceptions were tracked during the three flight operations (Green, Garrett & Alan Hoffman 2006). This supports the argument that long-life autonomy can not simply be reached through increased robustness and reliability, but rather through the ability to handle unforeseen situations. For instance, during the landing maneuver of NASA's Apollo 11 an exception caused a near failure of the mission. The exception occurred in a navigation and guidance computer, despite the fact that the system was extensively tested. In the end, the exception was handled manually after diagnosing the problem remotely. In a world where exceptions occur, autonomous systems require the ability to diagnose them-



(a) Hyper-modular, Multi-engine Printer.

(b) Reactive Exception Handling, exception handling without root-cause analysis, may fail given non-local root-cause scenarios.

FIGURE 1.3 Hyper-modular, Multi-engine Printer build at the Palo Alto Research Center (PARC, a Xerox Company).

selves and to adapt their behavior to the diagnosed conditions. Diagnosis is a reasoning process which explains discrepancies between predicted behavior and observed behavior, determines the system state, and improves the understanding of the system. In general, long-life autonomy can be seen as a combination of two methods: (1) diagnosis to determine the current condition of the system and (2) planning to optimize system operation for the diagnosed condition. A step towards long-life autonomy is to integrate automated diagnosis.

Long-life autonomy is not only required for space exploration, but also essential in many other domains. Automated manufacturing, for instance, requires automated diagnosis to handle production failures in order to avoid cost extensive down times. Consider the Hyper-modular, Multi-engine printer illustrated in Figure 1.3(a). Such printers resemble a product manufacturing plant. Raw materials (blank sheets) enter at the plant's inputs, they are routed through different machines, and final products (printed sheets) are collected at the outputs. Autonomous printer receive a stream of print jobs continuously over a long period of time, constructs a plan for each sheet without external guidance, and optimizes themselves for long-run performance. A long-life autonomous printer can use its own model to diagnose faults during execution (e.g. diagnosing which module causes a paper jam) to then stimulate the sheet planner to choose plans that avoid certain modules.

Envision a system without diagnosis, but instead with the ability to detect discrepancies and to react to the detected discrepancies without performing root-cause analysis. As a result, the system reacts to surface symptoms and not to the root-cause. Consider the example provided in Figure 1.3(b). Three single sheets of paper are planned to follow path 1. Assume that the module at location *CAUSE* creates a scuff mark. Once a scuff mark exists, it builds up over time and causes a sheet to jam downstream. As a result, the first sheet jams along path 1 indicated by the star labeled *JAM*. Following a purely reactive strategy, a system avoids the

jammed module in future plans as it has detected an exception in this module. Consequently, the second sheet is rerouted around the jammed module following path 2. However, path 2 still traverses the module causing the scuff mark and causes the paper to jam further downstream along path 2. By now, two modules are jammed and therefore taken offline. The reactive strategy continues and consecutive sheets are planed around the two jammed modules. However, those plans may still traverse the module causing a scuff mark as in path 3. Consider the third sheet then follows path 3 and jams. As a consequence the entire system is jammed.

The previous example illustrates that long-life autonomy has to be a combination of diagnosis to determine the current condition of the system and planning to optimize system operation for the diagnosed condition. The main challenge is to integrate planning and diagnosis such that the overall system benefits from its diagnosis capabilities while still maintaining high long-run performance. One of the most sophisticated and well developed integrations of diagnosis and planning was realized in the remote agent project (Muscettola, Nayak, Pell & Williams 1998). The overall system was successfully demonstrated onboard the NASA Deep Space One mission in May, 1999. The remote agent project performs passive diagnosis during regular operation by monitoring execution without actively optimizing operation for information gain. As a consequence, the combination of passive diagnosis with regular operation is often unsuccessful, as regular operation may not sufficiently exercise the system to isolate the underlying fault. An alternative approach is to alternate between explicit diagnosis and regular operation. Explicit diagnosis executes test cases which are optimized purely for information gain while regular operation is suspended. Such alternating phases typically result in long periods of time in which regular operation is suspended. This is particularly inefficient when diagnosing complex fault scenarios, such as faults that occur intermittently. Motivated to overcome the shortcomings of the two previous integrations, this work introduces a novel integration of regular operation and diagnosis leading to a higher long-run performance.

1.2 Motivation and Aims:

This work introduces a new architecture, coined a *Self-diagnosing Agent*, which realizes the integration by a novel diagnosis paradigm called *pervasive diagnosis*. Pervasive diagnosis actively manipulates the course of action during operation in order to gain diagnostic information without suspending operation. Consider a system where operational goals can be achieved in multiple ways. This flexibility can be exploited to generate operational plans that simultaneously gather information by trading off information gain with performance objectives. Therefore the resulting approach combines system operation and diagnosis in the following way:

Simultaneous execution of **active diagnosis** and **regular operation**. Consequently, active diagnosis and regular operation occur at the same time leading to higher long-run performance than an integration of regular operation with passive diagnosis or alternating between explicit diagnosis and regular operation.

The conceptual framework of pervasive diagnosis is illustrated in Figure 1.4. The framework is implemented by a loop: The core idea of pervasive diagnosis is that plans achieve operational goals and informative observations at the same time, coined informative operational plans. Those plans are executed and sent together with their corresponding observations

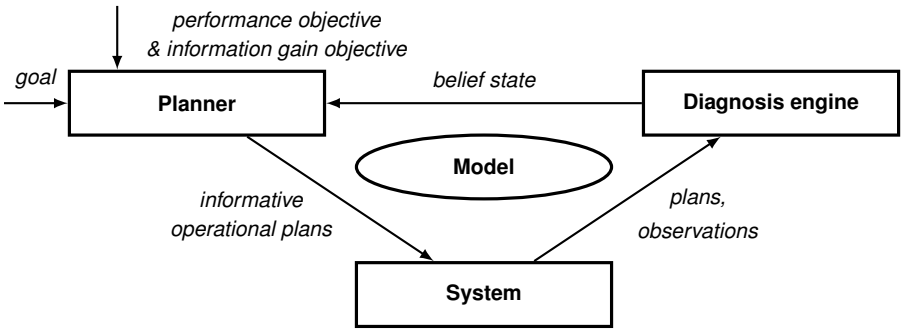


FIGURE 1.4 Pervasive Diagnosis: Integration of Operational Planning and Active Diagnosis.

to the diagnosis engine. The diagnosis engine updates its beliefs online to be consistent with the observations, updates the model, and forwards the beliefs to the planner. The planner then determines future plans based on the current beliefs, the performance objectives, and the information gain objectives. Systems that embody pervasive diagnosis benefit from high long-run performance by exploiting the overlap between operational plans and diagnostic plans.

However, not all systems can leverage pervasive diagnosis. In order to benefit form pervasive diagnosis the following three assumptions have to be true:

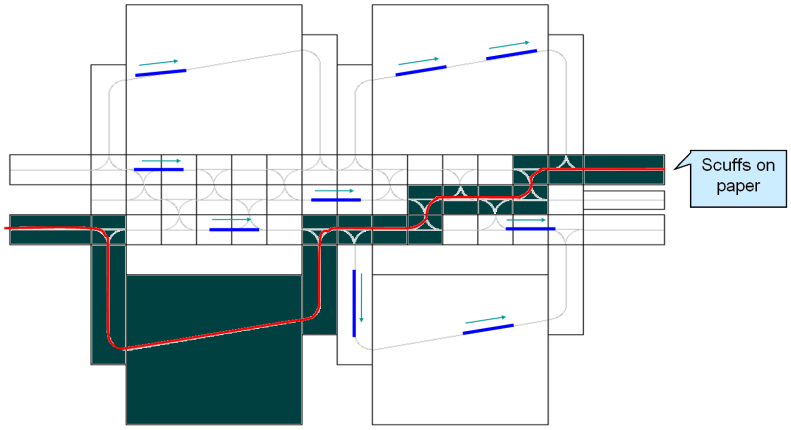
- **Assumption 1:** A **sequence of plans** is executed and **observed**. Pervasive diagnosis is a sequential diagnosis technique where individual operational plans are optimized for information gain. For example, a continuous stream of print jobs can be used for root-cause analysis by choosing alternative routes while printing.

- **Assumption 2:** There are **multiple different ways** in which **operational goals can be achieved**. For example, consider that the same print job can be achieved by different routes through a print system. In the case where no redundancy can be leveraged, it might be possible to alter action parameters such as execution speed.

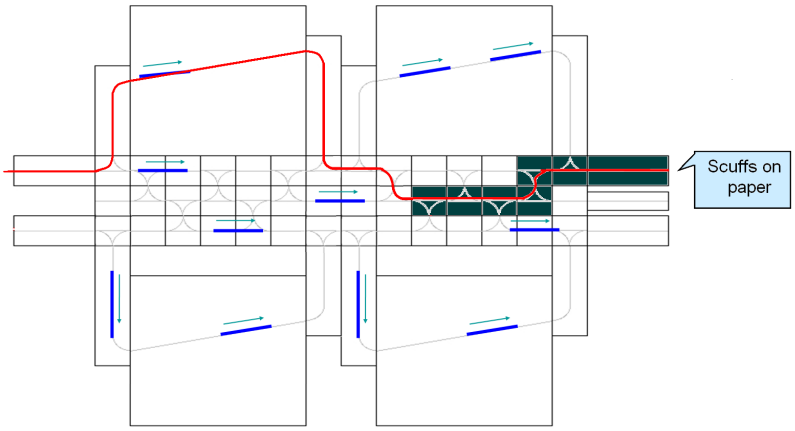
- **Assumption 3:** The space of **operational plans** and **diagnostic plans intersect**. The space of operational plans contains plans that are optimized to achieve operational goals and the space of diagnosis plans contains plans that are optimized for information gain. Those spaces may or may not overlap. In general, an observation process may not conflict with operational goals. A negative example would be a product, which needs to be cut open for observation and thus is not longer a valid product.

The core idea of the Self-diagnosing Agent Architecture is to enable systems to reflect on their actual behavior, to determine their own states, to infer their true capabilities, and to leverage their gained knowledge to operate robustly and reliably.

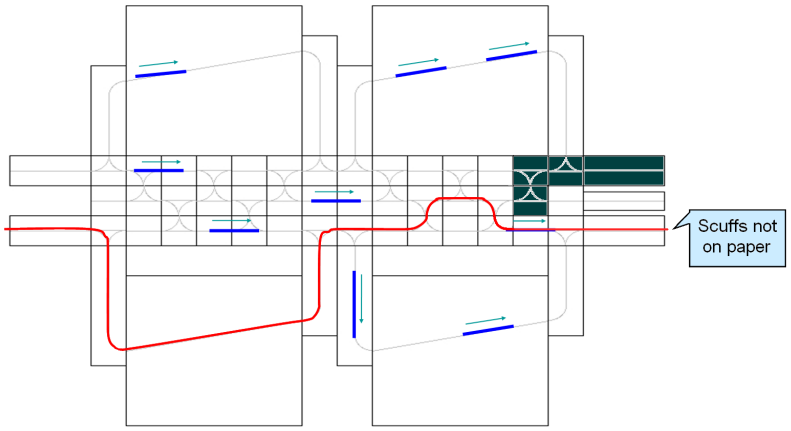
Consider a Hyper-modular, Multiple-engine Printer that contains a single, persistent fault, initially unknown to the system. The example is illustrated in Figure 1.5. The system prints a sequence of jobs with the overall objective to maximize long-run performance. Assume a sheet of paper is printed following the plan illustrated in Figure 1.5(a) and that the execution



(a) Plan execution with scuffed paper, thus all modules along the plan are suspected.



(b) Informative operational plan achieves either operational goals, suspected set reduction, or both.



(c) Here, plan execution leads to valuable diagnostic information without disrupting operation.

FIGURE 1.5 Self-diagnosing Agent deriving Informative Operational Plans.

results in an observable fault, a scuffed sheet. Given that there is no other information available, each module along the execution path is suspected to have potentially contributed to the fault. The set of suspected modules is highlighted in Figure 1.5(a). Since a fault was detected, a Self-diagnosing Agent optimizes future plans to perform regular operation and active information gain at the same time. Figure 1.5(b) illustrates such an informative operational plan. Note that the plan is optimized for system operation (traverses a print engine), but routes the sheet through some additional modules to increase information gain. In this particular case the plan fails to achieve an operational goal as the output sheet is scuffed. However, the plan still contributes by improving the diagnosis state. The core idea is that all plans either achieve operational goals, improve the diagnosis state by reducing the suspected set, or some combination. The next plan, illustrated in Figure 1.5(c), is again optimized to achieve operational goals and information gain at the same time. The execution results in a valid output (printed sheet), which leads to synergy effects between regular operation and active diagnosis.

In summary, this work introduces the *Self-diagnosing Agent* Architecture which uses *pervasive diagnosis*. Pervasive diagnosis is a novel diagnosis paradigm that enables regular operation and active diagnosis to occur at the same time leading to higher long-run performance than passive diagnosis or alternating between active diagnosis and regular operation.

1.3 Technical Challenges

The Self-diagnosing Agent framework is intended to run on real-world systems during long-life operation. This poses several technical challenges both on the conceptual and on the algorithmic side. The overall challenge decomposes into the following two challenges: Generating operational plans that simultaneously gather diagnostic information and performing online diagnosis for systems that plan. On the planning side, a planner has to tightly integrate operational objectives and diagnosis objectives such that regular operation and active information gathering can occur within a single plan. On the diagnosis side, a diagnosis framework has to perform diagnosis based on an abstract planning model while scaling with the size of the planning problem.

In order to determine a plan that gathers diagnostic information, an information criterion has to be defined that quantifies the potential information gain. This is particular challenging if the system under observation exhibits intermittent behavior, e.g. only one out of one thousand executions causes an observable abnormality. Consider the printer example from before where sheets are routed through modules and eventually get scuff marked. Suppose a plan is executed and a scuff mark is observed at the output as illustrated in Figure 1.5(a). Without further information all modules that have been involved in the print job are suspected to have potentially contributed to the fault. In order to diagnosis the system efficiently, a plan has to be generated that leads to informative observations. Intuitively, the most informative plan is the plan that traverses half of the suspected modules. However, consider the same problem where some of the modules fail intermittently, e.g. produce a scuff mark once every thousand times. The challenge is to examine if the half split intuition remains to be optimal even with a changing intermittency rate. After an information criterion is defined, an algorithm has to be designed that efficiently generates informative operational plans. This is difficult as the in-

formation gain of an action depends on the actions that have been executed previously to this action as well as on the actions that will be executed before the next observation is done. Suppose there is a set of suspected actions, we know one of the actions is faulty and we want to determine the potential information gain executing a specific action. The potential information gain is very high if we execute the action by itself. That is because if a fault is observed we can unambiguously identify this action as faulty. On the other hand, if we observe all of the actions together, observing a fault is informative. This dependencies between the individual actions/modules lead to a combinatorial search which is challenging to solve efficiently.

Another technical challenge is to perform online diagnosis for systems that plan. Online diagnosis considers diagnosis reasoning during system operation. This requires fast diagnostic inference to keep up with execution. This is especially challenging if diagnosing a system that plans based on its planning model. A planning model describes the nominal system behavior on an abstract level and omits many of the system details. This demands the consideration of advanced fault models to compensate for the lack if knowledge. For instance, a planning model may capture the transport actions of a paper transport module without going into detail which roller and motor is involved in any particular action execution. Due to the abstract nature of a planning model, faults may appear intermittently or only if a group of modules interact. The intermittency might be due to details, which are omitted by the model, e.g. the room temperature. An interaction fault, is a fault where a set of components behaves abnormal if they interact, but each individual component remains within specification if they are tested individually. For instance, imagine modules age over time and show some small delay in their behavior, but remain within specification. Suppose that executing two modules together leads to an accumulated delay that causes the system to fail, but this information is abstracted away by the model. As a result an execution that involves more modules fails, even though all individual models are within specification. The goal is to perform diagnosis despite the abstract model by introducing domain independent advanced fault models. The challenge is to enable efficient diagnosis inference despite the fact that advanced fault models generally increase the number of hypothesis exponentially. Therefore a diagnosis framework needs to be designed that handles complex fault scenarios while performing sufficiently fast to be applicable during operation.

In summary, the overall challenge is to perform online diagnosis to determine the current state of the system and to generate operational plans that gather informative observations based on the diagnosed state. This results in the following technical challenges: Define an information criterion that defines how informative a plan is, leverage this information criterion to generate informative operational plans, and perform efficient online diagnosis for systems that plan.

1.4 Contributions

This work presents the following five contributions:

- Self-diagnosing Agent framework: This contribution introduces an overall framework to tightly integrate regular operation, active information gathering, and online diagnosis.

The key idea is to exploit the flexibility of plan construction to simultaneously achieve operational goals while gathering diagnostic information. This contribution introduces an information criterion that determines how informative plans are, a strategy to derive informative operational plans, and an online diagnosis framework to efficiently update the belief states.

- Tiered-Partitioned Inference framework: This contribution outlines an efficient computational framework for statistical diagnosis featuring two main ideas: Tiered inference, which structures fault hypotheses into tiers based on their fault cardinality, and partitioned inference, which dynamically partitions the overall system into subsystems such that single fault inference can be used within each subsystem. As a result, this framework enables efficient online diagnosis for multiple faults by leveraging high-speed single fault inference (e.g. the one presented in the next contribution).

- Continuously Estimating Persistent and Intermittent Failure Probabilities: This contribution enables fast diagnostic inference for systems with persistent and intermittent faults. The memory and time efficient inference framework is realized by a combination of Bayesian Inference and count-based estimation to derive failure probabilities. In particular, this framework enables high-speed single fault inference, which leads to efficient online multiple fault diagnosis if combined with the previous contribution.

- Diagnosing Hidden Interaction Fault: This contribution presents a novel framework that extends model-based diagnosis to systems with hidden interaction faults. A hidden interaction fault is present if an unknown interaction among a set of components leads to an observable failure, even though each individual component meets specifications. The extension is realized by a general, domain independent model. This contribution enables diagnosis of systems, even if the underlying model is incomplete.

- Target-Value Search Problem: This contribution defines a new class of combinatorial search problems, called Target-Value Search Problem, in which the objective is to find the path or set of paths between two given vertices in a graph whose length/value is as close as possible to some *target-value*. Target-Value Search can be applied in the context of a Self-diagnosing Agent to derive informative operational plans.

1.5 Reader's Guide

The five contributions presented in the previous section are described in five chapters. An additional chapter on intelligent agents sets the context and outlines short comings of previous agent architectures.

- Chapter 2 describes the general concept of intelligent agents. An intelligent agent can be implemented by numerous agent architectures, which are outlined in this chapter, before short comings are presented, which motivate the rest of the discussion.

- Chapter 3 formulizes the Self-diagnosing Agent framework, which leverages a novel integration of regular operation and active diagnosis.

- Chapter 4 presents a computationally efficient diagnosis framework, which is based on tiered-partitioned inference. This framework leverages high-speed single fault inference to diagnose systems with multiple faults.
- Chapter 5 studies a framework to estimate persistent and intermittent failure probabilities from a continuous stream of observations. In particular, this framework realizes high-speed single fault inference.
- Chapter 6 describes a general, domain independent extension of model-based diagnosis to systems with hidden interaction faults.
- Chapter 7 defines a new class of combinatorial search problems, called Target-Value Search Problem. Target-Value Search can be applied in the context of a Self-diagnosing Agent to derive informative operational plans.
- Finally, Chapter 8 summarizes the work on Self-diagnosing Agents and points out the benefits of a Self-diagnosing Agent. Finally, it concludes by revisiting the core contributions of this work.

Some aspects of former versions of this work have been published in (Kuhn, Price, Do, Liu, Zhou, Schmidt & de Kleer 2010), (Kuhn, de Kleer & Liu 2009), (Liu, Kuhn & de Kleer 2009), (de Kleer, Kuhn, Liu, Price, Do & Zhou 2009), (Kuhn, Price, de Kleer, Do & Zhou 2008*b*), (Kuhn & de Kleer 2010), (Schmidt, Kuhn, Zhou, de Kleer & Price 2009), (Kuhn, Price, de Kleer, Do & Zhou 2008*a*), (Kuhn & de Kleer 2008).

CHAPTER 2

A World of Agents

This chapter studies agents by building on the concepts introduced in (Russell & Norvig 2009). It starts with an introduction in Section 2.1 followed by describing the general concept of an agent in Section 2.2. A formal framework of an intelligent agent is outlined in Section 2.3. Section 2.4 then introduces an example from the printer domain called Sheet Path Finder (Shepafi). Until then, agents have been described in terms of behavior without talking about their internal implementation. Section 2.5 presents architectures on how an agent might be realized. In some situations agents might have to reason about the different available action choices before committing to a specific action. This process is called planning and is outlined in Section 2.6. Section 2.7 gives a brief overview how agent technology can be used to control hyper-modular, multi-engine printer and outlines the challenges which might occur. Finally, Section 2.8 concludes the chapter by motivating a tight integration of regular operation and online diagnosis.

2.1 Introduction

For thousands of years, mankind has tried to understand how the human brain works and what it is that makes humans intelligent. In 1956 the name artificial intelligence, in short AI, was coined as part of the proposal titled "A Proposal for the Dartmouth Summer Research Project on Artificial Intelligence" (McCarthy, Minsky, Rochester & Shannon 2006). Artificial intelligence studies, not only how intelligence works, but also how intelligence can be built.

But what is intelligence? Numerous researchers have attempted to answer this question. The answers can be summarized in four categories of definitions (Russell & Norvig 2009). The common understanding is that a system is perceived to be intelligent if it can be characterized by one of the four definitions in Table 2.1.

The definitions of intelligence bring out two dimensions. The first differentiator is whether it is enough for an intelligent entity to be passive, as in the sense of thinking (but maybe not acting), or active, in the sense of not only thinking, but also acting. The second dimension highlights the reference for intelligence. The question is if either the human, as an intelligent being, sets the bar, or intelligence has to be compared to rational behavior.

All four definitions are interesting and have been pursuit in science. Without engaging in a deeper discussion about what the correct definition is, in the context of this work artificial intelligence is simply defined as the computational study of agents with rational behavior, agents that act rationally.

	Humanly
Thinking	Thinking Humanly (Haugeland 1985, Bellman 1978)
Acting	Acting Humanly (Kurzweil 1990, Rich & Knight 1990)
	Rationally
Thinking	Thinking Rationally (Charniak & McDermott 1985, Winston 1992)
Acting	Acting rationally (Poole, Mackworth & Goebel 1998, Nilsson 1998)

TABLE 2.1 Definitions of Artificial Intelligence

2.2 Concept of an Agent

The word agent origins from the Latin word “agere”, which translates into “to do”. As the origin of the word agent suggests, an agent is something that acts. In general, an agent is everything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators (Russell & Norvig 2009).

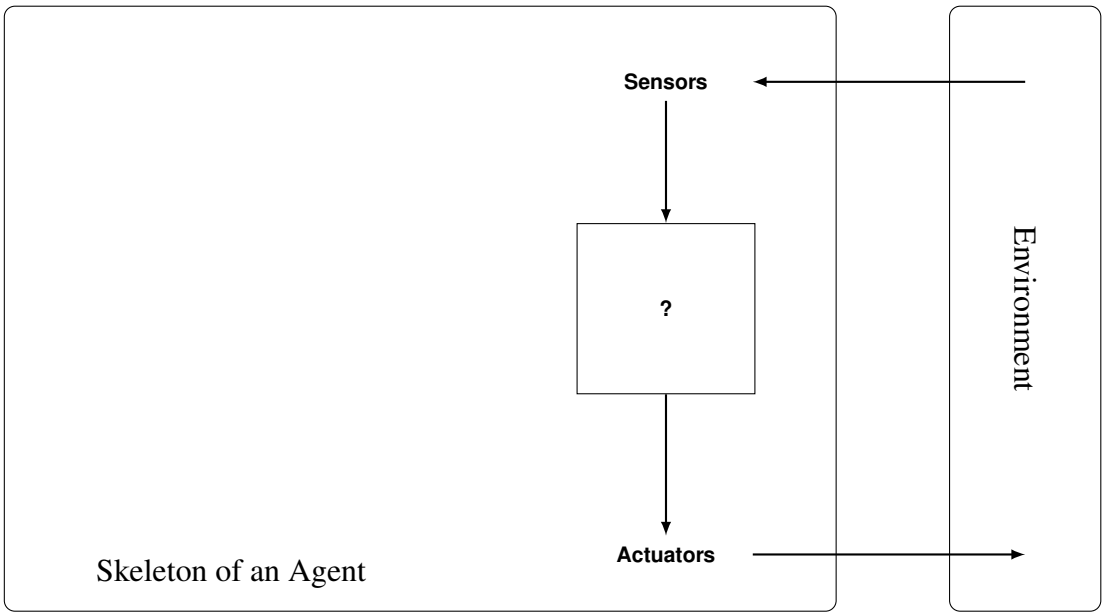


FIGURE 2.1 Skeleton of an Agent

The concept of an agent is illustrated in Figure 2.1. A human can be seen as an agent with eyes, ears, nose, skin, and other organs as sensors and hands, legs, mouth, and other body parts as actuators. Similarly, a robot can also be viewed as an agent with its input as sensor input and its output as actuator effects. Every agent is able to sense its own actions, but not necessarily

the effects caused by those actions. For example, a robot that grabbed for an object knows that it has grabbed, but might not know if it holds the grabbed object in hand. The success of the grabbing action might not be observable.

An agent receives a continuous stream of inputs through sensors. The term percept is used to refer to the perceptual inputs at any given instant and the term percept history for the entire stream of received percepts. As indicated in Figure 2.1, there might be a dependency between the perceptual inputs and the agent's choice of actions. In general, the behavior of an agent can be described by specifying a choice of action for every possible percept history. A complete specification of the behavior is called the agent function.

2.2.1 Agent Function

The behavior of an agent can be formalized by a function mapping every possible percept history onto an action choice. This function is referred to as agent function. In principle, an agent function can be stored in a lookup table, where every possible percept history is mapped to an action choice. Using the lookup table as part of an actual implementation is rather impractical because the size of the resulting lookup table would be infinite for most agents as the set of unique percept histories is infinite. Instead, the concept of an agent program is introduced to implement an agent function.

2.2.2 Agent Program

The behavior of an agent is characterized by the agent function. However, an actual implementation is realized by the so called agent program. An agent program implements an agent function with the limitation that it is only parameterized with the currently available percept. In comparison, an agent function maps entire percept histories to actions. The important distinction is that it receives the complete percept history as an input whereas the agent program receives only the current percept at any given instant. The agent function is to be understood as a theoretical concept whereas the agent program is an actual implementation of an agent function.

However, both, the agent function and the agent program, describe the behavior of an agent. But how should an agent behave? The next section continues the discussion by studying intelligent agents.

2.3 Intelligent Agent

An intelligent agent acts autonomously, perceives the environment, acts persistently over a period of time, adapts to changes, and adopts predefined goals. An intelligent agent that does the right thing based on the available knowledge is coined rational agent (Russell & Wefald 1991). But what is the right thing? This question is addressed in the next section.

2.3.1 Performance Measurement

A performance measure can be used to quantify the degree of an agent of doing the right thing. The happiness of an agent can be used as performance measure by asking the agent about its degree of happiness, assuming that being happy is the right thing, and stimulate the agent to reach the highest degree of happiness. The disadvantage of such an approach is that some agents might not know how happy they are, or they delude themselves about their own happiness. Another problem is that being happy is very subjective. Similar to people, some agents might be happy in a situation, whereas other agents might be very unhappy in the same situation. To come to a more objective performance measure the performance of an agent is evaluated against an external objective, typically defined by the designer of the agent. Such a performance measure is referred to as objective performance measurement, or simply objective. The objective is used to quantify the behavior of an agent. A rational agent is an agent that maximizes its performance measure based on the knowledge available to the agent. The next section defines a rational agent more formally.

2.3.2 Rational Agent

Generally, a rational agent does the right thing given its available knowledge. The concept of a performance measurement can be used to adapt this very general definition to the following definition of a rational agent:

Definition 1. *For each possible percept history, a rational agent should select an action that is expected to maximize its performance measure, given the evidence provided by the percept history and whatever built-in knowledge the agent has (Russell & Norvig 2009).*

Note, that a rational agent is not an omniscient agent. An omniscient agent can precisely predict how the world evolves and how its actions effect the environment. Based on this prediction, an omniscient agent then chooses actions in order to maximize its performance measure. In comparison, a rational agent can only hypothesize about expected outcomes of its actions and how the world evolves. Based on those hypotheses a rational agent chooses the action that maximizes its performance measure. The important difference is that an omniscience agent is certain about the future whereas a rational agent can only anticipate how the future might be.

An agent does not only rely on the currently available knowledge, but also actively engages in increasing its knowledge about the environment. It would not be rational to cross a road without checking the traffic and then determining that the percept history did not indicate traffic. Instead, a rational agent should be aware of its lack of knowledge and choose actions to change that. In general, a rational agent should consider gathering information in order to increase the expected performance measure. This leads to the concept of active information gathering. Information gathering is the process of choosing actions with the goal to alter future percepts in order to increase the agent's knowledge. It is also referred to as exploration.

Another important quality of a rational agent is the ability to learn. Information gathering by itself is not all to useful if an agent is not able to incorporate the gathered information into its knowledge. Learning is the ability to enrich its own knowledge with new information and to make it available for later use.

The rest of this document discusses how rational agents are built. Before going into more detail, an example domain is introduced, which will serve as a reference example to illustrate further concepts.

2.4 A Real-World Agent

The increasing complexity of systems makes it far more important that intelligent agents control themselves. A good example of this trend is the domain of high-end printers. Even though desktop printers are limited to printing and copying, a high-end printer is much more complex. They come with an entire collection of functions, like printing, scanning, faxing, binding, stapling, sorting, stacking, etc., and give the user a broad range of options to choose from, such as different paper types, color options, printing simplex or duplex, etc. An example for such a high-end printer is the Xerox iGen3 shown in Figure 2.2.

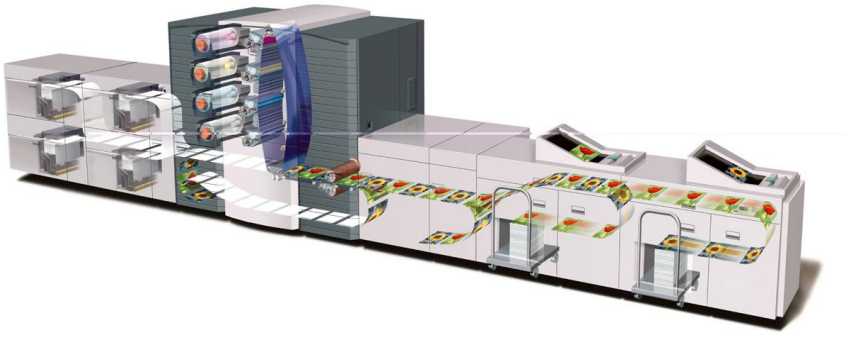


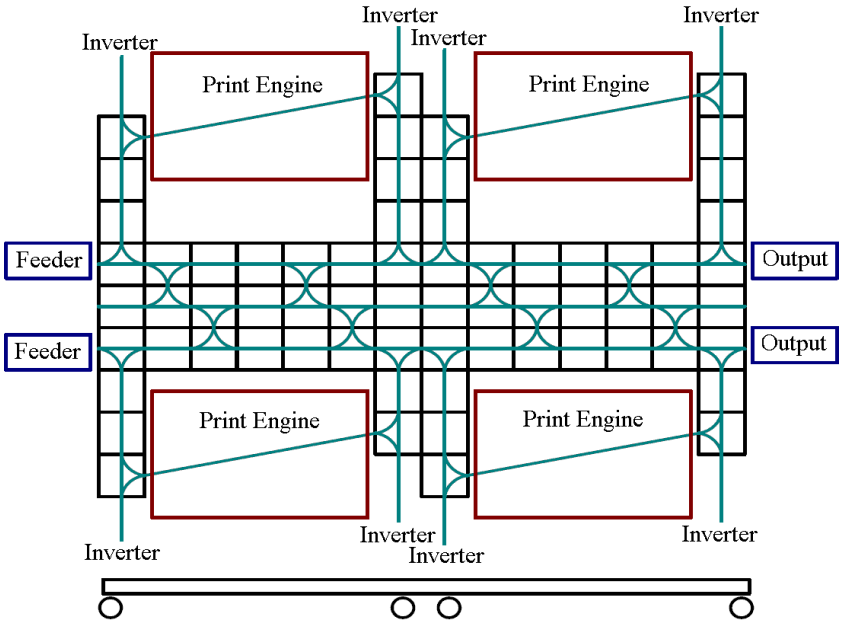
FIGURE 2.2 High-end production color printer: Xerox iGen3.

Many different possible combinations of functions and options increase the complexity not only in terms of hardware, but also in terms of software. Each possible combination might require a slightly different program to control the printer, which potentially leads to an enormous amount of software. For example, a printer with 10 functions and 10 options, where all combinations of functions and options are valid configurations, requires $1,048,576 = 2^{10} \times 2^{10}$ programs. This leads to an enormous amount of software. To avoid this complexity without restricting the functionalities of future products, XEROX pursues the goal that future machines can automatically plan, control, and reconfigure themselves from knowledge of their own physical constraints and their environment in order to achieve the users goals. As part of this vision a group of researchers at the Palo Alto Research Center, Inc. (PARC, formerly XEROX PARC) started to work on intelligent agent technologies that enables machines to automatically plan and control themselves based on a system model and available sensor data. In particular, the group developed a hyper-modular multi-engine printer, which fully embraces model-based planning and scheduling.

2.4.1 Hyper-modular, Multi-engine Printer

This section introduces the concept of the hyper-modular, multi-engine printer developed at the PARC. The hyper-modular, multi-engine printer can be seen as a network of paper trans-

(a) Picture of the prototype.



(b) Schematic of the prototype.

FIGURE 2.3 The hyper-modular, multi-engine printer combines over 170 independently controlled modules.

port modules, linking multiple print engines and other special purpose modules together. Figure 2.3 shows the hyper-modular, four-engine printer. It has over 170 independently controlled modules, including many paper transport modules, two paper feeders, four printers, and two output trays.

Thus printer resemble a product manufacturing plant, with raw materials (blank sheets) entering at the plant's inputs, being routed through different machines that can change the properties of the materials, and the final products (printed sheets) being collected at the outputs. This domain also has certain properties of package routing or logistics problems with cross-goal resource constraints, which represent a wide range of on-line decision-making settings, such as multi-robot coordination.

2.4.2 Running Example: Sheet Path Finder (Shepafi)

In the following, a running example system is introduced, which is referred to as *Sheet Path Finder*, in short *Shepafi*. The system is motivated by the earlier introduced hyper-modular, multi-engine printer.

Shepafi is a system with reduced complexity, simplifying the problem to a sheet transportation problem. It assumes that individual sheets of paper have to be navigated through the system starting from one of two feeder modules, traversing some paper transport modules, and finishing at one of the two finisher modules. Details about the process of printing or other capabilities, such as stapling, are abstracted away. Further moving from module to module is assumed to be an atomic action. The resulting problem resembles a sheet transportation problem. This problem is small enough to be quickly understood, but conveys all necessary properties to serve as a meaningful example. A schematic of the *Shepafi* system is illustrated in Figure 2.4.

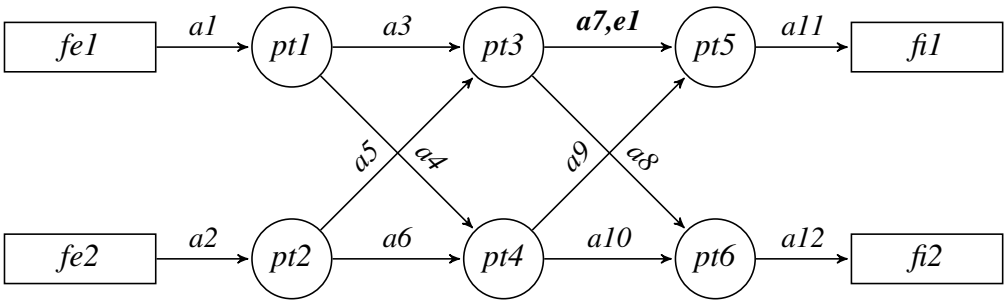


FIGURE 2.4 **She**et **Pa**th **Fi**nder (Shepafi): Topology

Shepafi consists of a set of ten modules, *M={fe1,fe2,pt1,pt2,pt3,pt4,pt5,pt6,fi1,fi2}* (two feeder modules, two finisher modules, and six intermediate paper transport modules) and a set of actions *A={a1, a2,..., a12}*. The modules are connected by actions as indicated in the schematic shown in Figure 2.4.

Suppose an uncontrolled failure event *e1*. Further assume, that the occurrence of this failure event can ony be observed at a finisher. The failure event *e1* occurs if and only if action *a7* is executed. The failure event is assumed to be persistent (not intermittent), hence the resulting

observation can be simulated by checking if an executed course of action contains action *a7*. Given a course of action, denoted plan *p*, a successful execution is indicated by <*p,succ*> and a failed one by <*p,fail*>.

Shepafi serves as a running example through out the entire discussion. Consider a system such as *Shepafi*, it remains to be shown how an intelligent agent can be implemented controlling it. Therefore next section introduces numerous agent architectures and highlights their advantages and disadvantages.

2.5 Architecture of an Agent

Until now, agents have been described in terms of behavior without talking about their internal implementation. This section introduces architectures on how an agent might be realized. Even though internal implementations might vary greatly, conceptually they can be fit into six architectures. The skeleton of an agent program is illustrated in Figure 2.1 and can be extended to the following existing agent architectures:

- Table-driven Agent,
- Simple Reflex Agent,
- Model-based Reflex Agent,
- Goal-based Agent (Model-based, goal-based Agent),
- Utility-based Agent (Model-based, utility-based Agent),
- Learning Agent (Model-based, utility-based, learning Agent),

2.5.1 Table-driven Agent

A Table-driven Agent is the most trivial implementation of an agent program. Internally, the current percept is directly mapped onto a choice of action. The agent bases its decision-making directly on the perceptual input, without interpreting the perceptual input by some kind of higher level situation or world state. It maintains a lookup table of actions indexed with all possible percepts and uses those rules to determine what action to perform next. The architecture of a Table-driven Agent is illustrated in Figure 2.5.

This limits the Table-driven Agent to fully-observable worlds of limited complexity. The main drawback of this architecture is the lack of an interpretation step. An example that illustrates the problem is an agent that plays a board game equipped with a camera to sense and a display to indicate the intended moves. For simplicity, assume that the move on the game board is realized by a human. In a table-driven architecture the perceptual input is directly mapped to an action. In our example the agent maps raw image data to move actions. Due to lighting conditions or the camera angle the same game state might result in many different raw image data, even though they all represent the same high level game state. Given this architecture, each unique image data has to be captured in a lookup table. This quickly gets out

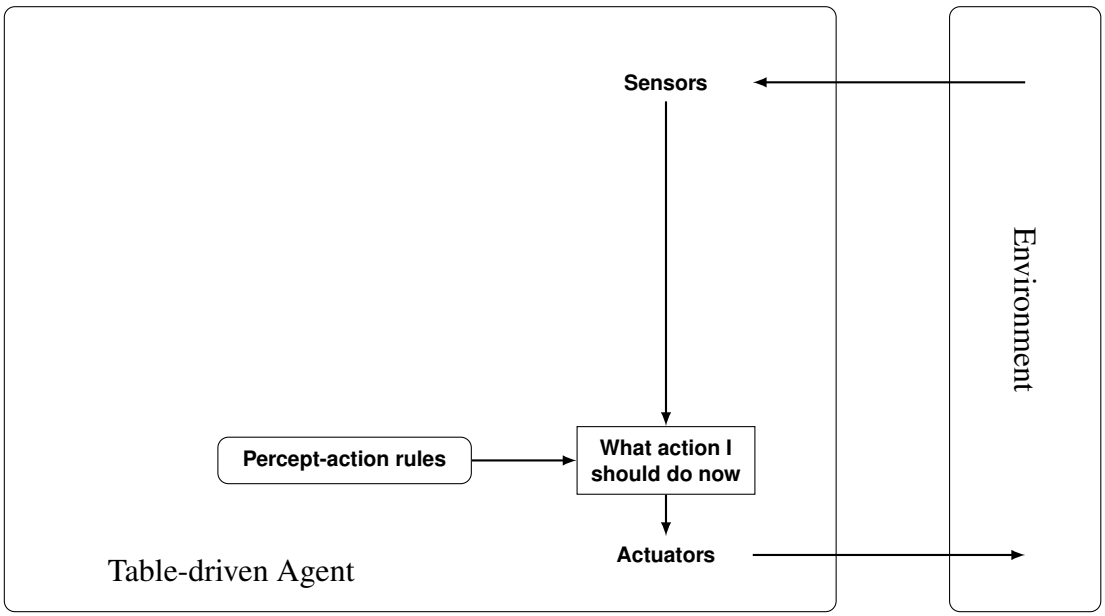


FIGURE 2.5 Table-driven Agent

of hands in terms of the table size and leads to an enormous overhead within the lookup table. The next section introduces the simple reflex agent who avoids this overhead by introducing an abstract interpretation step mapping perceptual inputs to a higher level world state, before choosing an action.

2.5.2 Simple Reflex Agent

The Simple Reflex Agent, is another simple implementation of an agent. It works similarly to a Table-driven Agent, with the only difference that it interprets the current perceptual input as a higher-level state before choosing an action. The interpretation step reduces the number of stored mappings, because multiple unique perceptual inputs are interpreted by the same higher-level situation or state. For example, in the context of the board game it is only necessary to maintain an action choice for each game state but not for all possible raw images. This leads to enormous savings in the number of stored mappings, because the same game state might be represented by many different images due to lighting differences or camera angles.

Internally, the agent stores a set of condition-action rules, which map from a condition to an action. As illustrated in Figure 2.6, the agent first determines the current state, before evaluating the condition-rules with respect to the current state. The first condition-rule with a positive evaluating condition determines the choice of action. A template of such a rule is shown in Table 2.2. Since an agent maintains its knowledge in rules it, is also called a Rule-based Agent. Simple Reflex Agent emerged as a subfield of artificial intelligence in order to capture expert knowledge. The goal of Simple Reflex Agent is to make expert knowledge computationally available (Hayes-Roth 1985).

The Simple Reflex Agent architecture only succeeds in an environment where the knowl-

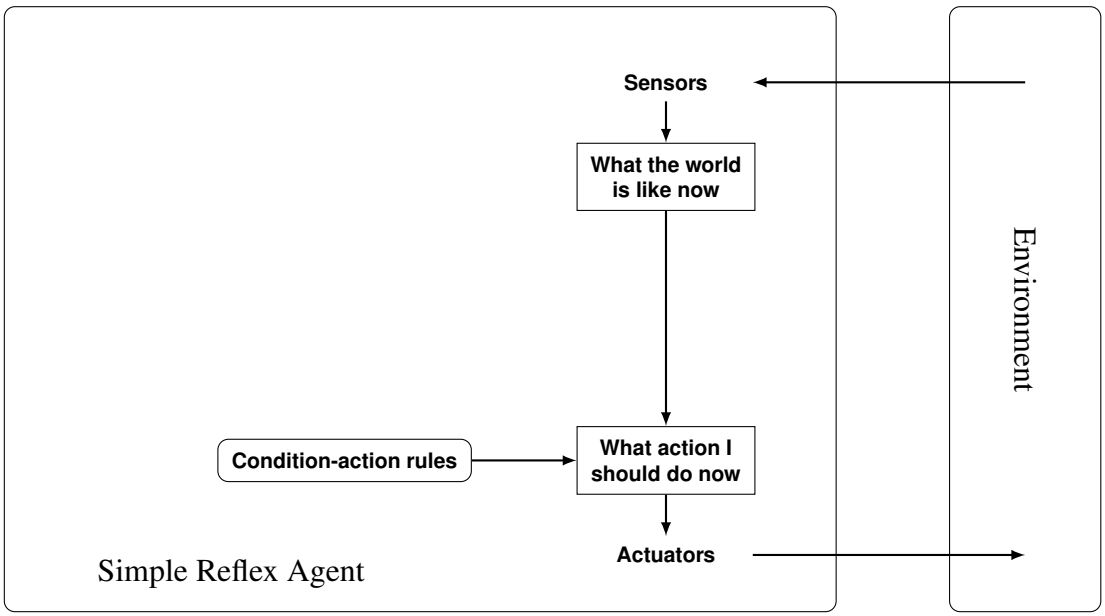


FIGURE 2.6 Simple Reflex Agent

Rule n:	**IF**	*condition*	**THEN**	*action*

TABLE 2.2 Template of a condition-action rule.

edge of non-perceptual parts of the current state is required. In the context of our running example system *Shepafi* the adoption of this agent means that, if the location of a sheet would not be fully observable at any given time the agent would be setup to fail. The reason behind is a Simple Reflex Agent determines the current state purely by its current perceptual input. As a result, a Simple Reflex Agent is limited to fully observable environments. The next section introduces an agent architecture that maintains an internal state to handle worlds with partial observability.

2.5.3 Reflex Agent with Internal State (Model-based Reflex Agent)

In some situations an agent might interact with a world, where not all parts of the world are always fully observable at any given time. In those situations it is useful to maintain an internal state in order to keep track of the parts that are not fully observable. In our *Shepafi* example, a sheet location might not be observable, but the agent can still succeed by tracking the current location given the selected actions. Such kind of tracking can be realized by an internal state.

A Reflex Agent with Internal State selects actions based on its current state, with the difference that the current state is derived not only from the current percept but also based on an internally stored state. Maintaining an internal state enables the agent to remember information contained in earlier percepts. In order to maintain an internal state, the agent needs an understanding of "How the world works". This knowledge can be decomposed into two categories: The first one captures aspects regarding the environment, basically knowledge of

"How the world evolves". The second part holds information regarding the actions of an agent, in particular the effects of those actions. This is often described as "What my actions do".

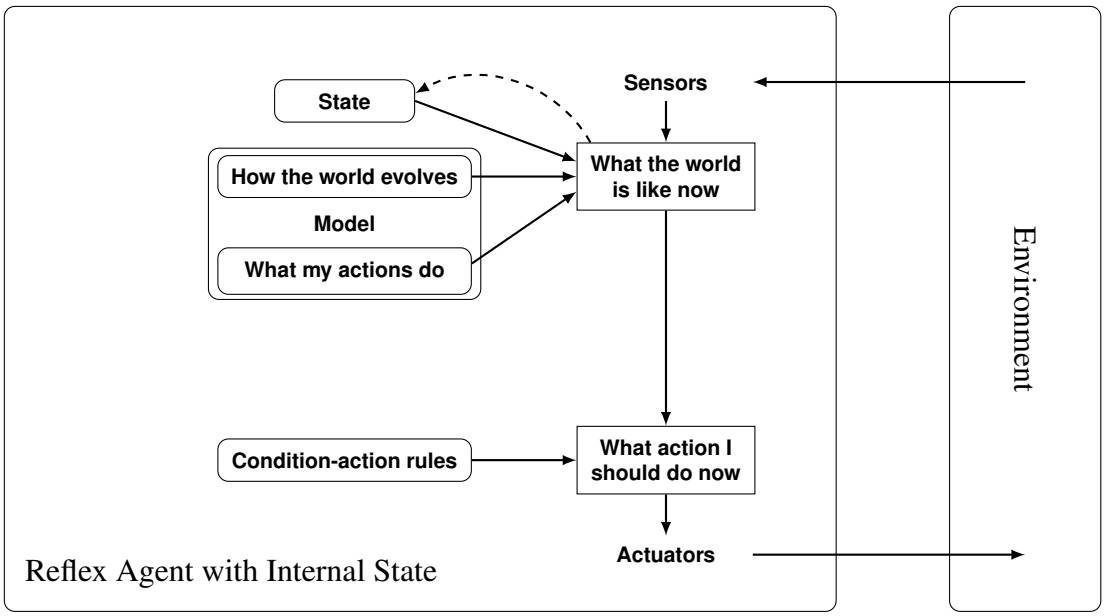


FIGURE 2.7 Reflex Agent with Internal State

This knowledge is called a world model, or simply a model. The model represents the knowledge of the agent on "How the world works". The model is an integral element in the process of maintaining an agent's perceived world state. As shown in Figure 2.7, a world state is derived from the previous state, action, the model, and the current percept. Once the world state is determined the Reflex Agent with Internal State continues its selection process in accordance to the Simple Reflex Agent. That is by matching a condition-action rule to the current state and choosing the corresponding action.

In comparison to a Simple Reflex Agent, a Reflex Agent with Internal State overcomes the limitation of being constrained to a fully-observable world. However, all reflex agents, which are based on condition-action rules share the lack of flexibility to adapt to dynamically changing goals. Condition-action rules implicitly capture the goal, which an agent intends to achieve by characterizing the behavior of an agent. In case the goal changes, the entire rule set has to be updated in order to adapt the behavior according to the new goal. This leads to an enormous overhead, especially if goal changes are numerous.

The next section discusses an agent architecture that is able to take on different goals and to simulate how different action choices compare with respect to the goal achievement.

2.5.4 Goal-based Agent

The previous agents, in particular reflex agents, suffer from a lack of flexibility to adapt to dynamically changing goals. A reflex agent requires a complete re-build of its condition-action rule set if a new goal emerges. In the situation of a new goal, an agent should be able to

leverage its understanding of its own actions and how the world evolves, to adapt its behavior. The knowledge required to do so is captured in the model currently used to maintain an internal state. The same knowledge can be leveraged to adapt the action selection process to a new goal.

An agent with explicit goal description and the ability to adapt its behavior accordingly is referred to as Goal-based Agent. A Goal-based Agent differs from a reflex agent as it does not rely on predefined condition-action rules to describe its behavior. Instead, a Goal-based Agent dynamically leverages its knowledge of its own actions of how the world evolves in order to determine the best action. Given a goal, it engages in an explicit deliberation process to evaluate potential action choices, based on the current state, an internal model of its own actions, and a model of how the world evolves. A Goal-based Agent considers possible future scenarios to decide what the best action is. These considerations can be viewed as simulations of possible futures to determine "What will happen if I do action A or action B?" and "What effect is better with respect to the goal?". The consideration process is referred to as planning or search and is discussed in more detail in Section 2.6.

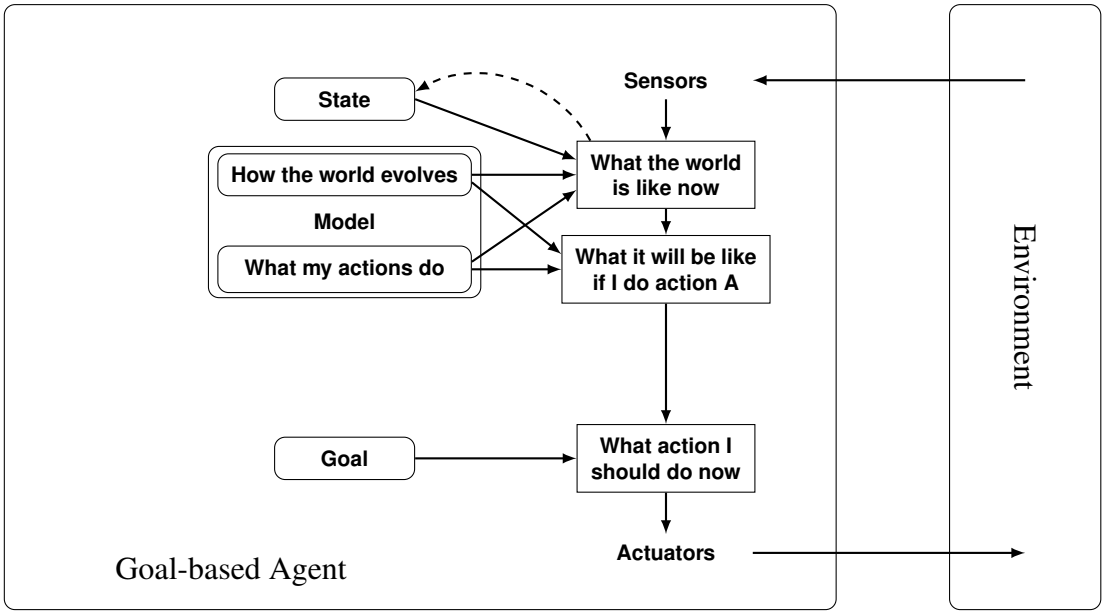


FIGURE 2.8 Goal-based Agent

The main advantage of a Goal-based Agent is its increased flexibility. That results from the capability to leverage its own understanding of the world not only to determine a world state, but also to adapt its own behavior in the presence of new goals.

A Goal-based Agent has the advantage of being able to adapt to changing goals, but has no notion of how different goal achieving strategies compare. As the idiom "All roads lead to Rome" indicates there might be many different ways to achieve a goal. Among those different goal achieving strategies, a Goal-based Agent expresses no preferences. This might result in goal achieving strategies which are less optimal than others.

The next section presents an Utility-based Agent approach that introduces the concept of a continuous success measurement instead of a binary measurement of either goal achieved or

not.

2.5.5 Utility-based Agent

Sometimes an agent might be able to retrieve multiple different ways to achieve a goal. In such situations it is of interest to evaluate the alternatives in order to determine which one is "best". A Goal-based Agent distinguishes only between goal achievement and goal failure, but lacks the concept of a quality of achievement.

A Utility-based Agent chooses actions based on the utility, resulting from a successor state. The utility is determined from an utility function. A utility function indicates a quality for each state by mapping a state to a measure of the utility of that state. In the case, multiple alternative goal achieving strategies can be identified, e.g. in the example of traveling to Rome, a Utility-based Agent can determine the preferable way.

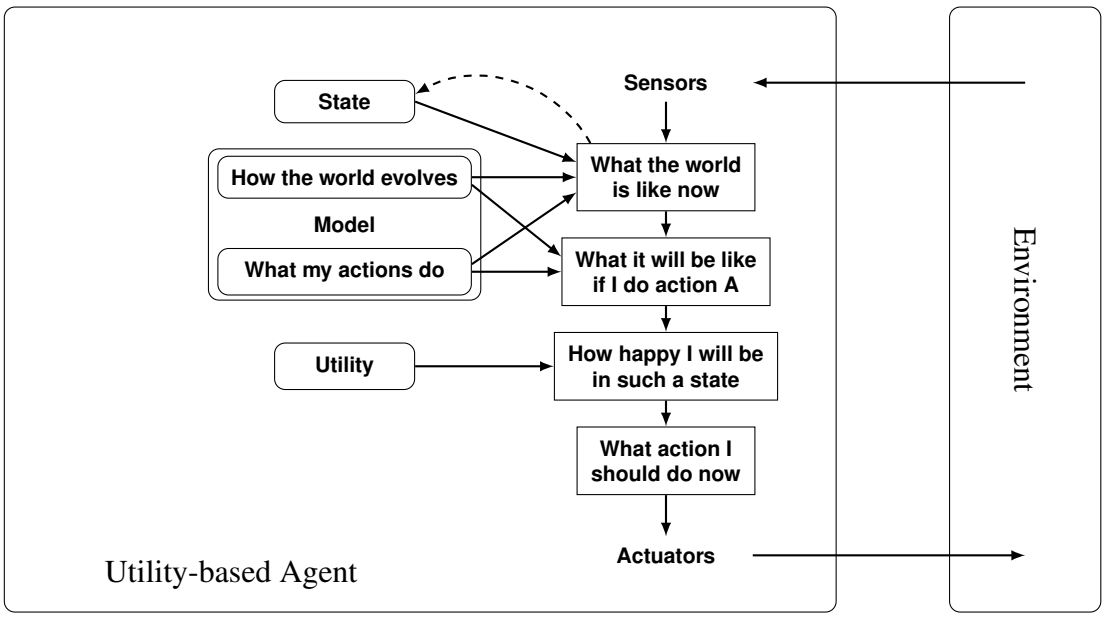


FIGURE 2.9 Utility-based Agent

As illustrated in Figure 2.9, the goal is replaced by an utility function and a reasoning step, which quantifies the utility of an action, hence of a possible successor state. For example, one could define in our example system *Shepafi* that it is preferable to route a sheet without bending it. In this case, the utility function would need to reflect that those goal achieving strategies are preferred, which do not bend the sheet.

A Utility-based Agent even allows the tradeoff between multiple competing quality measures by combining all sub-utility functions into one overall utility function. As an example imagine a system that optimizes the trade-off between the likelihood of success and the cost of reaching the goal.

Until now agent have been built on the assumption, that there is a correct model available which describes how the world works. But what happens if this assumption does not hold?

The next section presents a general concept of a learning agent that intends to continuously improve its model by learning how the world works.

2.5.6 Learning Agent

The previous sections covered various architectures of agents, which described how an agent leverages its information about the world to act in order to achieve a goal or lead to the highest utility state. This builds on the underlying assumption that there is an accurate model available to the agent. In an early paper, Turing considers the possibility of programming the knowledge manually into an agent, but finally concludes in the same paper that this would not only result in too much work, but also limits an agent to known environments. In order to act in unknown environments, an agent must be able to recover from an incomplete or misrepresented model of the world. The difficulty is, that if an agent derives its action choices from an inaccurate model, it is most likely to fail, given its misunderstanding of the world. Turing therefore suggests an alternative approach, which envisions agents that learn. Those agents can then be taught how they should behave. As a consequence, at first an agent needs to be aware of the possibility that its understanding of the world might be incorrect. Only agents with this awareness can increase their understanding of the world. A Learning Agent needs to identify incorrect or incomplete knowledge and be able to improve its knowledge by learning (Russell & Norvig 2009). The goal of a learning agent is to increase its understanding of the world and to leverage its improved understanding to perform better than without learning.

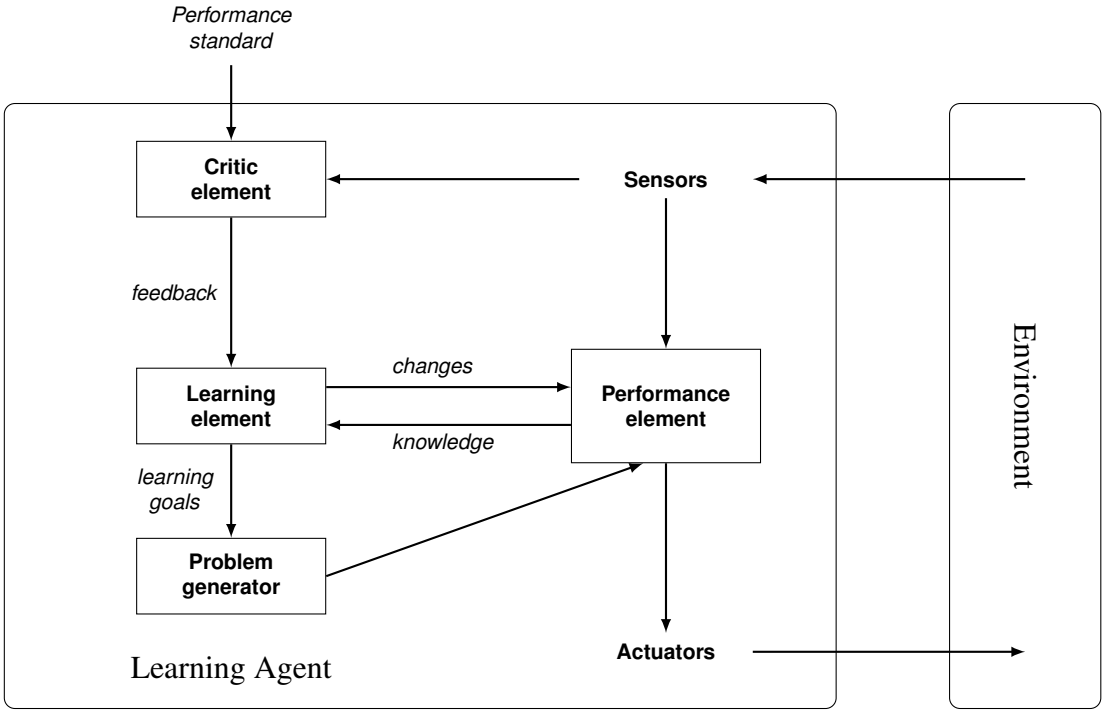


FIGURE 2.10 Learning Agent

Figure 2.10 introduces a general architecture of a Learning Agent. A Learning Agent can be decomposed into four components: a performance element, a critic element, a learning element, and a problem generator element.

In this architecture the performance element refers to an utility based agent. It reasons about the future, more specifically about how the world will evolve and what the effects of actions might be, in order to derive an appropriate action choice based on the currently available knowledge. This task is isolated from the learning part, which is concerned on how the currently available knowledge can be improved. The learning element of an agent can only improve the behavior of an agent if the current behavior is evaluated and criticized, which is done by the critic element.

The critic element evaluates the consequences of the agent's behavior and provides feedback on how the agent can improve its behavior. The evaluation is based on an external performance standard, as the percept itself does not convey how the agent is performing. For example, a print agent might receive data about a scuff mark on a sheet as perceptual input, but the fact of a scuff mark alone does not indicate if that is good or not. In fact, the agent requires a performance standard in order to know that scuff marks are not wanted. The percept itself does not indicate that. The performance standard is external to the agent to avoid any influence of the agent on it. In particular, it is the goal to avoid that the agent adapts the performance standard to its current behavior, instead of adapting its own behavior. The resulting feedback of the critic element is then provided to the learning element.

The learning element gathers feedback from the critic element based on which it improves its internal knowledge. In a more concrete example, a printer agent without learning might route the paper through many unnecessary parts of the system with the result that the overall throughput decreases. A learning agent can learn that unnecessary resource allocations slow down productivity and adapt its behavior to avoid those unnecessary allocations. Generally, performance feedback is represented by rewards and penalties, which can be leveraged to improve the utility function of the Utility-based Agent in the performance element. In order to be able to learn, an agent has to collect informative percepts, which is the responsibility of the problem generator.

The problem generator stimulates the agent to perform experiments in order to gather informative percepts. Those experiments are designed based on the current knowledge and the certainty about this knowledge. Sometimes it can be beneficial to choose sub-optimal actions in the short run, to improve the agents understanding of the world and therefore the performance over the long run. In the printer domain it might be useful to take different routes through the system in order to verify that those parts are functional before the technician leaves the site, even though some of those routes are sub-optimal in terms of resource allocation.

The critic element is responsible to evaluate the current behavior with respect to the performance standards and furthermore give its feedback criticism to the learning element. The task of the learning element is to take the feedback into account, to incorporate the feedback into the knowledge base, and to propagate changes to the model of the performance element. The problem generator receives learning goals from the learning element and transforms them into performance goals, which are forwarded to the performance element. The performance element takes those goals into account.

In summary, a learning agent can be used to learn knowledge, which is helpful to improve

the performance of an agent. The main advantage of a Learning Agent is the capability to learn on its own, without being manually programmed. That enables Learning Agents to operate in an unknown or evolving environment.

2.6 Planning: The Reasoning Side of Action

The previous section discussed various architectures of intelligent agents to enable agents that are capable to act autonomously, perceive the environment, act persistently over a period of time, adapt to changes, and take on predefined goals. In general, a Rational Agent should select an action that is expected to maximize its performance measure. In some situations a Rational Agent might have to reason about the different choices of available actions before committing to a specific action. This process is called planning or search, and is the reasoning side of acting. This section presents planning by building on some of the concepts introduced in (Nau, Ghallab & Traverso 2004).

In general, planning is concerned with the consideration of possible futures. A more technical definition of planning describes it, as:

Definition 2. *Planning is an abstract, explicit deliberation process to choose and organize actions before acting in order to maximize its expected performance measure, given the evidence provided by the percept history and whatever built-in knowledge an agent has.*

Automated planning is a subfield of artificial intelligence that studies this deliberation process of choosing and organizing actions computationally. Not all situations require an explicit planning process before acting. For example, an agent might choose not to plan, if it possesses the knowledge that an action or pre-stored plan leads directly to the goal or that acting can be freely adapted while executing actions. However, there are situations in which an agent chooses to simulate and evaluate potential action sequences before committing to a particular action. Usually planning is considered by an agent who faces new situations, has to achieve complex tasks or objectives, has to chose from less familiar actions, or if adaptation to actions during execution is constraint by high risk or high cost consequences. In general planning is understood as a costly and resource-consuming process. Therefore agents only plan if the trade-off between cost of planning and the resulting benefits suggest to plan.

There are many practical applications that motivate the field to build automated planning tools. The field of planning applications may be categorized based on the intended task to be solved and the corresponding action type. Roughly speaking, planning applications may be divided into path and motion planning, perception planning and information gathering, navigation planning, manipulation planning, communication planning, and several other forms of social and economic planning. The following section provides a way how one may think about the individual categories.

- *Path and motion planning* is concerned with finding a geometric trajectory through some configuration space that connects a start configuration and goal configuration while avoiding collisions or more general spaces of negative utility, given kinematics constraints and the internal system dynamics. A commonly cited problem is the Piano

Mover Problem (Natarajan 1986), in which a fleet of robots intends to move a piano from one room to another without bouncing into walls or furniture. The planner has to consider the kinematics and the dynamics of the piano as well as of the robots.

- *Perception planning and information gathering* is concerned with selecting and organizing actions to achieve the highest information gain to enrich some knowledge base. The planner has to trade-off the information value of different sensing-actions or actions that stimulate the system to determine the best strategy to gather data. Examples are active probing in diagnosis, localization and map building in navigation problems, or generally active state estimation or active system identification.

- *Navigation planning* is concerned with navigating a robot or more generally a mobile device to some goal location through some environment. Typically navigation planning combines perception planning, path planning, and motion planning. Fundamentally, navigation requires the ability to determine the current position (information gathering) and then to plan a path towards some goal location (path and motion planning). In order to navigate through some environment, an agent requires some representation of the environment i.e. a map and the ability to interpret that representation.

- *Manipulation planning* is concerned with choosing and organizing actions to manipulate an object into some goal configuration. Such actions might include grabbing, lifting, turning, modifying, constructing, or destroying an object.

- *Communication planning* is concerned with reaching a target audience using marketing communication channels such as advertising, public relations, experiences or direct mail for example. It is concerned with deciding who to target, when, with what message and how.

- *Social and economic planning* is concerned with planning economic activity in production, or in directing an economy towards specific goals, socially or economically.

Planning in AI aims to build agents with rational behavior. Planning is the reasoning side of acting and is therefore a key element to computational intelligence. For that reason it is important to understand the theory of planning as well as being able to build programs that plan.

2.6.1 Conceptual architecture of Planning

The following section formulates a conceptual design for planning. The conceptual architecture involves four components: a planner, a controller, a model, and a system. In order to provide a practical implementation for planning the following aspects have to be defined: the role of the individual components, an architecture that describes the interactions among them, and all individual components.

2.6.1.1 Components and how they interact

The four components in a model-based planning system are a planner, a controller, a model, and a system. Figure 2.11 illustrates the four components and their interactions.

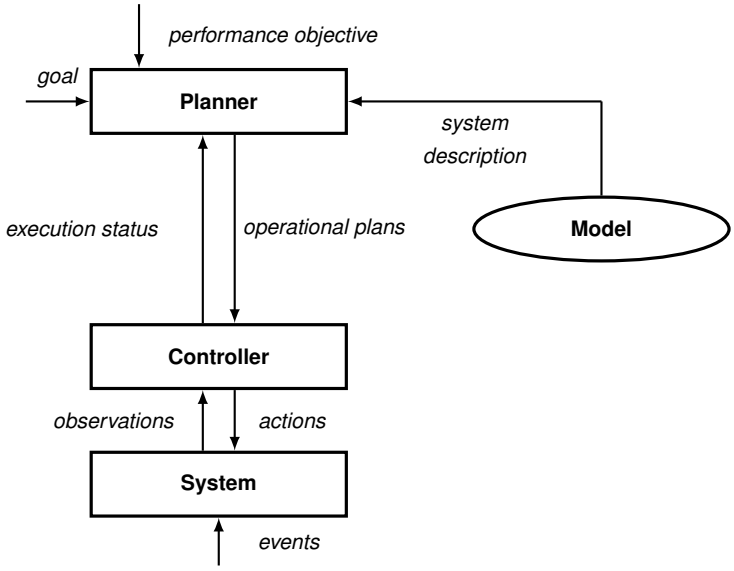


FIGURE 2.11 Plan System Architecture

The role of each individual component can be thought of as follows:

- *System:* A system evolves through state space driven by events and actions, which it receives as input. Based on its evolution, a system provides information about its current state as output.

- *Model:* A model is an abstract, declarative representation of a system, describing its capabilities and structure. It serves as a base for the selection process of actions for a planner.

- *Controller:* Given a plan and current state information as an input, a controller releases actions to the system, which are in accordance to the plan and the current state information.

- *Planner:* A planner, given a system description, a set of initial states, information about the current execution status, and a goal, synthesizes a plan that advises the controller which action is appropriate to execute in order to achieve a given goal.

The four components are common among different model-based planning systems, even if individual planning systems might differ in their detailed definition, e.g. what exactly a plan is, how much state information a controller has, or assumptions made about the system. The next section discusses the individual components of a planning system in more detail.

2.6.1.2 System

A common technique used in computer science to model a dynamic system, is the general model for state-transition systems.

Definition 3. *A state-transition system is a quadruple* $\Sigma =< \mathcal{S}, \mathcal{A}, \mathcal{E}, T >$*, where*

- $\mathcal{S}$ *is a finite set of states;*
- $\mathcal{A}$ *is a finite set of actions;*
- $\mathcal{E}$ *is a finite set of events; and*
- $T : \mathcal{S} \times \mathcal{A} \times \mathcal{E} \rightarrow \mathcal{S}$ *is a state-transition function.*

A state-transition system may be represented by a directed graph with labeled arcs. Given a state-transition system $< \mathcal{S}, \mathcal{A}, \mathcal{E}, T >$, the corresponding graph may be constructed by creating a vertex for each state $s \in \mathcal{S}$ and an arc from s to s' labeled (a, e) for each transition $s' \in T(s, a, e)$. It is convenient to define a neutral action $noop$ and a neutral event ϵ to account for state-transitions which are strictly due to events or actions. In the case of $T(s, a, \epsilon) \neq \emptyset$ there exists a set of transitions, which start from state s triggered only by action a. Accordingly if the set $T(s, noop, e) \neq \emptyset$ there exists a set of transitions, which start from state s triggered only by some event e.

Not all actions are necessarily applicable in all states. An action $a \in \mathcal{A}$ is applicable in state $s \in \mathcal{S}$ if $\bigvee_{e \in \mathcal{E}} T(s, a, e)$ contains at least one state-transition. One can introduce an applicability function $app : \mathcal{S} \times \mathcal{A} \rightarrow \{true, false\}$. Given a state $s \in \mathcal{S}$, an action $a \in \mathcal{A}$, and a state-transition function T the semantic of the applicability function app can formally be defined by

$$app(s, a) \Leftrightarrow \bigvee_{e \in \mathcal{E}} T(s, a, e) \neq \emptyset, \tag{2.1}$$

where $true$ indicates that action a is applicable in state s and $false$ indicates the opposite.

The system evolves through state space triggered by both, actions and events. The difference between an action and an event is whether they are intentionally controlled by a planner or not. An action is intentionally controlled by a planner as part of a goal achievement process. An event, on the other hand, is contingent; a planner has no control over events. However, events need to be considered by the planner as, they may cause state transitions in the system. Without defining an event formally, an event can be viewed as something which may lead to a state-transition. Events may occur, triggered by an action execution (*action-event*), as a result of several conditions, which are suddenly all met in a state, (*state-event*), or spontaneously dictated by nature (*nature-event*).

This can be illustrated with the concrete application *Shepafi* modeled as a state-transition system, as presented in Section 2.4.2. *Shepafi* consists of ten modules $\mathcal{M} = \{fe1, fe2, pt1, pt2, pt3, p4, p5, p6, fi1, fi2\}$, and twelve actions, $\mathcal{A} = \{a1, a2, \ldots, a12\}$, connecting the modules as shown in Figure 2.4. Further, assume that there exist two events $\mathcal{E} = \{e1, \epsilon\}$. Event $e1$ causes a fault if and only if action $a7$ is executed and event ϵ is the

neutral event. To model the occurrence of a fault a health variable $h = \{\neg AB, AB\}$ is introduced. The health variable is assigned to $\neg AB$ (not abnormal), if the fault has not occurred, and to AB (abnormal) otherwise. The states of the system are represented as pairs, consisting of a module and a health assignment. Note that a health assignment indicates that a fault has occurred, but does not indicate which particular module behaves abnormal. The transition function is defined by the directed graph shown in Figure 2.12. Summarized, this results in the state-transition system $\Sigma_{Shepafi} =< \mathcal{S}, \mathcal{A}, \mathcal{E}, T >$, where

- $\mathcal{S} = \{< m, h > \| m \in \{fe1, fe2, pt1, pt2, pt3, p4, p5, p6, fi1, fi2\}, h \in \{AB, \neg AB\}\}$;
- $\mathcal{A} = \{a1, a2, \ldots, a12\}$;
- $\mathcal{E} = \{e1, \epsilon\}$; and
- T is represented by the directed graph shown in Figure 2.12.

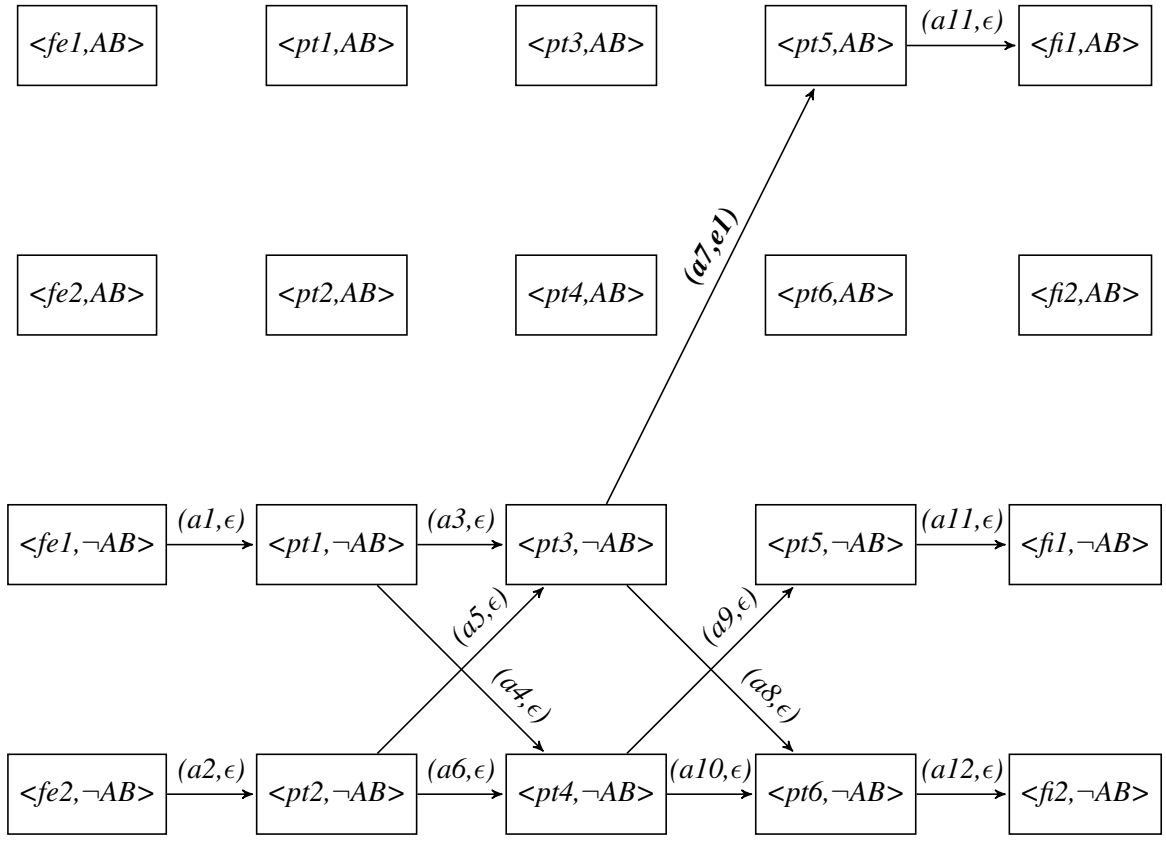


FIGURE 2.12 **She**et **Pa**th **Fi**nder (Shepafi): State-Transition System where each state represents a potential state of a sheet (not a module).

$\Sigma_{Shepafi}$ defines all states, actions, events, and state-transitions in *Shepafi*. The directed graph shown in Figure 2.12 captures how the system evolves, stimulated by actions and events. However, until now it was not described how systems can be observed. In order to model

how the system can be observed, the current state-transitions is extended by an observation function. The resulting system is a state-transition system with observation function, coined *observable state-transition system*.

Definition 4. *An observable state-transition system is a 6-tupel* $\Sigma^{obs} =< \mathcal{S}, \mathcal{A}, \mathcal{E}, T, \Phi >$*, where:*

- $\mathcal{S}$ *is a finite set of states;*
- $\mathcal{A}$ *is a finite set of actions;*
- $\mathcal{E}$ *is a finite set of events;*
- $T : \mathcal{S} \times \mathcal{A} \times \mathcal{E} \rightarrow \mathcal{S}$ *is a state-transition function;*
- $\Phi : \mathcal{S} \rightarrow 2^{\mathcal{S}}$ *is an observation function.*

In case of an observable state-transition system Σ^{obs}, the elements of a state-transition system Σ are extended by an observation function Φ. The observation function $\Phi(s) = \{s_1, \ldots, s_n\}$ maps state $s \in \mathcal{S}$ into a set of states, called belief state. A belief state represents all states the system could possible be in, given the current uncertainty. The uncertainty represented by the belief state might result from partial observability. In this case the observation function $\Phi(s)$ maps state s, the true underlying system state, to a set of states containing all states the system could possibly be in. In case that the system is fully observable the observation function results in the identity function. Continuing with the example, the state-transition system $\Sigma_{Shepafi}$ can be extended to an observable state-transition system by adding an observation function. According to Section 2.4.2, observations can only be collected at the two finisher modules. An observation at any other location does not lead to any information. The observation capabilities is modeled by an observation function that returns for all the following states $< fi1, \neg AB >, < fi2, \neg AB >, < fi1, AB >$, and $< fi2, AB >$ the state itself and for all other states the entire set of states, $\mathcal{S}$. This ensures that observations are only informative if they are made at one of the two finisher modules. After adding the observation function to $\Sigma_{Shepafi}$, the resulting system $\Sigma^{obs}_{Shepafi}$ is an observable state-transition system.

In general, the observable state-transition systems can be used to model planning systems.

2.6.1.3 Controller

A controller is an entity that releases actions to a system according to some plan, provided by a planner, and some belief state, given by an observation function of the system. The exact implementation varies greatly and depends on the assumptions made about a system, e.g. if the system is fully or partially observable. In general, a controller assumes that a plan combined with a belief state can be leveraged to extract the appropriate action, which is than executed next. In many planning systems, the planner is assumed to be off-line in the sense, that there is no information flowing back from the controller to the planner. However, since a real world system may evolve over time there is a chance that the orginal model does not correctly represent the system. To address those mismatchs between the model and the system,

the controller is assumed to be capable of handling such mismatch. Some more sophisticated systems incorporate an on-line planner to respond to exceptions during the planning process. In such systems, planners are not limited to only planning, but take on the burden of interleaving planning and acting in order to perform plan supervision, plan revision, and, if necessary, re-planning.

2.6.1.4 Planner

In general, planning is concerned with finding which action to apply to which set of states, in order to achieve a pre-stated goal, given a description of a state-transition system, a set of initial states, and information about the execution status. Typically, the planner organizes the solution to a particular problem in a data structure, called plan. Hence, the controller is advised which action is appropriate to be executed, given a belief of the current system state. The goals, a planner may be requested to achieve, can be categorized into the following goal types:

1. In the simplest case, the goal is specified by a set of goal states $\mathcal{S}_g$ (sometimes the set might contain only a single goal state s_g) and is achieved, if the plan execution ends in any of those goal state. Note that the plan execution has to terminate in a goal state, rather than evolving a system through a goal state. This type of planning is also called feasible planning.

2. The goal may also be specified by a set of goal states $\mathcal{S}_g$, an utility function attaching rewards or penalties to each state, a compound function, and an optimization criterion. In this case the goal is to end plan execution in a goal state, while optimizing the compounded utility over all the states the system evolved through based on the optimization criterion. Planning problems following this goal specification are called optimal planning.

3. Similar to the preceeding goal specification, a goal may be stated in the same way except that the utility function attaches rewards or penalties to actions. The goal is to terminate the plan execution in a goal state, while optimizing the compounded utility over all actions released to the system based on the optimization criterion. This type of planning is coined optimal planning.

4. Another way to formulate the goal, is to specify a task (e.g. a set of actions or recursively a set of tasks) and to require the system to perform the task in order to achieve the goal. Note that this specification does not require termination in a specific state, as long as the task is achieved.

This goal types conveys an idea on how planning goals may be specified. The specification of an initial situation is usually captured in either a single initial state, or in a belief state which is captured by a set of states, denoting all the states the system could be in initially.

The conceptual model introduced so far, is very loosely defined and can hardly serve as a reference for a practical implementation. The intent of this conceptual model, is to serve as a frame of reference throughout the discussion and to convey how planners may be thought

of. In general, planners rely on the restrictions made about the system and on the kind of information available. The next section outlines a set of relevant model restrictions, followed by a description of the common planning problems stated in the literature.

2.6.2 Classes of Planning

Consider some kind of planning problem with an associated system. Sometimes it is helpful to analyze the system, whether or not certain restrictions can be made. Usually, a more restricted problem representation leads to a more efficient solution. A common set of restrictions is presented, which tends to accelerate problem solving while simultaneously maintaining relevant aspects of the original problem. In particular, the following restrictions have been shown to be helpful:

2.6.2.1 Model Restrictions :

- **Restrictions A1 - Finite:** A system is finite if the state set, the action set, and the event set are finite.

- **Restrictions A2 - Fully Observable:** A system is fully observable, if the observation of any two states, $s, s' \in \mathcal{S}$, is not equal, $\phi(s) \neq \phi(s')$. The observation function results in the identity function. In this case, the controller has perfect knowledge about the current state of the system at any given time.

- **Restrictions A3 - Deterministic:** A system is deterministic, if the state-transition function T defines at most one state-transition for any combination of a state $s \in \mathcal{S}$ with either an event $e \in \mathcal{E}$ or an action $a \in \mathcal{A}$. Formally, a system is deterministic if the following term holds true: $\bigvee_{s \in \mathcal{S}} \bigvee_{a \in \mathcal{A}} \bigvee_{e \in \mathcal{E}} |T(s, a, e)| \leq 1$. In a deterministic system the controller has perfect knowledge about the successor state given the current state and the occurred event or action.

- **Restrictions A4 - Static:** A system is static if it remains in the same state unless the controller releases an action to the system. In a static system the set of events is empty.

- **Restrictions A5 - Restricted Goals:** Goal restricted problems may only specify the goal by either a single goal state s_g or a set of goal states S_g and may only require the plan execution to terminates wants a goal state is reached. This corresponds to the first goal type described earlier.

- **Restrictions A6 - Sequential Plans:** A system is controlled by a sequential plan if the plan is a linear ordered finite sequence of actions.

- **Restrictions A7 - Implicit Time:** A system meets the implicit time requirement if all state-transition are assumed to be instantaneous. No action nor event in the system implements the concept of duration.

- **Restrictions A8 - Offline Planning:** A planner is an off-line planner if it does not take any state information into account during planning. The planner synthesizes the plan purely based on the system description, the set of initial states, and the pre-stated goal. Therefore internal dynamics do not have any implication on the planning process.

The outlined model restrictions can be combined in various ways, which defines different classes of planning problems. Over the following sections classical planning, planning under uncertainty, and temporal planning are introduced.

2.6.2.2 Classical Planning

Classical planning assumes that all earlier presented restrictions (A1-A8) hold and that the system can be represented by a restrictive state-transition system. A restrictive state-transition system is a finite, deterministic, static system with restricted goals and implicit time representation. Classical planning further assumes that the initial state, denoted s_0, is a single state and known to the planner.

Such a system can be represented by the quadruple $\Sigma_{classic} =< \mathcal{S}, \mathcal{A}, \epsilon, T >$ instead of the 5-tuple $\Sigma_{obs} =< \mathcal{S}, \mathcal{A}, \mathcal{E}, T, \Phi >$ for two reasons: First, there are no contingent events, and secondly, since the system is assumed to be deterministic and the initial state s_0 is known, all other states can be predicted with certainty. This reduces the planning problem to the following:

Given a system as a triple $\Sigma_{classic} =< \mathcal{S}, \mathcal{A}, T >$, an initial state, $s_0 \in \mathcal{S}$, and a set of goal states $\mathcal{S}_g \subseteq \mathcal{S}$ the problem can be reduced to find a sequence of actions $< a_1, a_2, \ldots, a_k >$ corresponding to a sequence of state-transitions $< (s_0, a_1, \epsilon, s_1), (s_1, a_2, \epsilon, s_2)..., (s_{k-1}, a_k, \epsilon, s_k) >$ such that if the sequence is applied starting from the initial state s_0 the system evolves into some state s_k and s_k is a goal state, $s_k \in \mathcal{S}_g$.

In classical planning, conform to restriction A6, a plan can be represented as a linear ordered finite sequence of actions. The controller follows the sequence of actions unconditionally, and releases the actions to the system without considering any observations regarding the current state. This is also called open-loop planning.

Classical Planning appears rather trivial: The task of planning reduces to path search in directed finite graphs, which is well-studied and many well-known solutions already exist. Theoretically, for any system $\Sigma_{classic}$ the corresponding graph can be created and searched with known algorithms, like Bellman-Ford-Algorithm (Bellman 1958), Dikstra-Algorithm (Dijkstra 1959), A*-search-algorithm (Hart, Nilsson & Raphael 1968), or any other search algorithm. However, practically many of the problems quickly get to big to be represented explicitly. A good example is chess. With chess it is possible, in principle, to play a perfect game or construct a machine to do so as follows: One considers in a given position all possible moves, then all moves for the opponent, etc., until the end of the game. The end must occur, by the rules of the games, after a finite number of moves. Each of these variations ends in win, loss or draw. By working backward from the end, one can determine whether there is a forced win, the position is a draw, or it is a loss. In his 1950 paper, "Programming a Computer for Playing Chess", (Shannon 1950) has estimated the number of possible positions, "of the general order of $\frac{64!}{32!8!^2 2!^6}$, or roughly 10^{43}". Later Victor Allis also estimated the game-tree complexity to be

at least 10^{123}, "based on an average branching factor of 35 and an average game length of 80". As a comparison, the number of atoms in the observable universe is estimated to be between 4×10^{79} and 10^{81}.

Therefore, there is a need to have implicit representations and to construct solvers which can leverage those implicit representations without unfolding the problem explicitly for finding a solution. The Planning Domain Definition Language (PDDL) is a commonly used formalism to represent planning tasks implicitly. PDDL is an attempt to standardize how a planning task is described. It was developed mainly to make the International Planning Competitions possible, which started in 1998 (McDermott 2000). Ever since, PDDL is the dominating formalism to specify planning domains and tasks. The official PDDL specification provides more information (Ghallab, Nationale, Aeronautiques, Isi, Penberthy, Smith, Sun & Weld 1998) on the syntax and semantics. Other commonly used formalism are Strips (Fikes & Nilsson 1971) and ANML (Smith, Frank & Cushing 2008).

2.6.2.3 Planning under Uncertainty

Another area in planning is concerned with planning under uncertainty. This branch of planning is motivated by the desire to reflect some of the aspects of real-world systems more realistically. In the context of a real-world system, planning has to scope with contingent event occurrences, non-deterministic actions, or partial state information. Generally, in the area of planning under uncertainty, one or some of the restrictions are relaxed. In particular, one or more of the following restrictions are relaxed: A2 (Fully Observable), A3 (Deterministic), or A4 (Static). As a result, the field can roughly be divided into the following subfields: planning with partial observability, planning with non-deterministic actions, planning for non-static systems, and any combination of them.

- **non-deterministic**: In a non-deterministic system the outcome of some, or all of the actions can not be reliably predicted. A classic example of such a system is to throw a dice. Let's say, our example system consists of one dice and a dice throwing action. The system can be modeled with six states, indicating which of the six faces is currently facing up, and one action, which can be applied independently of the current state. Applying an action to the dice results in one of the six states. If the dice is assumed to be fair, than the outcome of a throw can not reliably be predicted. In planning this is called a non-deterministic action, an action with unpredictable outcome.

- **non-static**: A non-static system is a system with contingent events. An event is understood as a non-intentional occurrence of a state transition. In the case a system is non-static, the planner as well as the controller can not simply rely on the state information they have, as there might has been a contingent state-transition in the meantime. A planning system has to take eventual state-transitions into account and generate plans that can be applied independently of the occurrence of a contingent state-transition.

- **partial observability**: In the face of uncertainty about the behavior of the system, the ability to observe the current state of a system might be essential in order to control the system. This is true, if the selection of the next action depends on the current state

information and can not be done without it. However, there are systems which can be controlled even so one has only partial knowledge about the current state. The uncertainty about the current state is a combination of partial observability together with either non-deterministic actions or non-static system behavior. In the case partial operability combined with non-deterministic actions, the outcome of an action can neither be reliably predicted, nor determined by observation which leads to uncertainty regarding the current state. On the other hand, if the system evolves contingently, as in the case of a non-static system, and the system is not fully observable, there remains some uncertainty about how the system has evolved.

There are many planning systems that handle planning under uncertainty. In such systems the uncertainty is usually captured in the model. In the case of a non-deterministic system, the uncertainty is usually modeled by annotating different action outcomes with probabilities. A non-static system behavior is captured by modeling the probabilities of contingent state-transitions and in the case of a partial observable system an observation function maps observations into belief states.

The hyper-modular, multi-engine printer domain is a good example where planning under uncertainty is necessary. A failure event, e.g. a paper jam or scuff mark, might occur during action execution leading to unpredicted behavior. In addition the actual behavior might only be observable further down stream. In our running example *Shepafi* an failure event might happen and it is assumed that a failure can only be observed at one of the two finisher modules. This leads to a non-deterministic, non-static system with partial observability.

2.6.2.4 Temporal Planning

An important aspect of many real world planning applications is time. Actions and events take place over a period of time, and the possibility of taking actions may depend on events and other actions taking place simultaneously. This leads to a relaxation of restriction A6 (Sequential Plans) or restriction A7 (Implicit Time). The resulting differences separating temporal planning from the classical planning can be characterized mainly in terms of two aspects:

- In temporal planning actions do not sequentially follow each other, but may temporally overlap and interfere. The possibility of taking an action may depend on whether some other actions are being taken. In classical planning actions are taken in a sequence, and the possibility of taking an action is independent of earlier (and later) actions, given the current state.

- In temporal planning the effects of an action may be a complex function of the state and other simultaneous actions, whereas in classical planning they are independent of other actions.

In the domain of hyper-modular, multi-engine printers temporal behavior is every important. Multiple sheets might be routed through the system at any given time, which requires resource coordination. Temporal aspects are important to ensure that no collisions among simultaneously executed sheets occur. One important temporal aspect are action durations.

2.7 Autonomous Hyper-modular, Multi-engine Printer

This section gives a brief overview how agent technology can be used to control hyper-modular, multi-engine printer. The hyper-modular, multi-engine printer domain resemble a product manufacturing plant, with raw materials (blank sheets) entering at the plant's inputs, being routed through different machines that can change the properties of the materials, and the final products (printed sheets) being collected at the outputs. Multiple feeders allow blank sheets to enter the printer at a high rate and multiple finishers allow several jobs to run simultaneously.

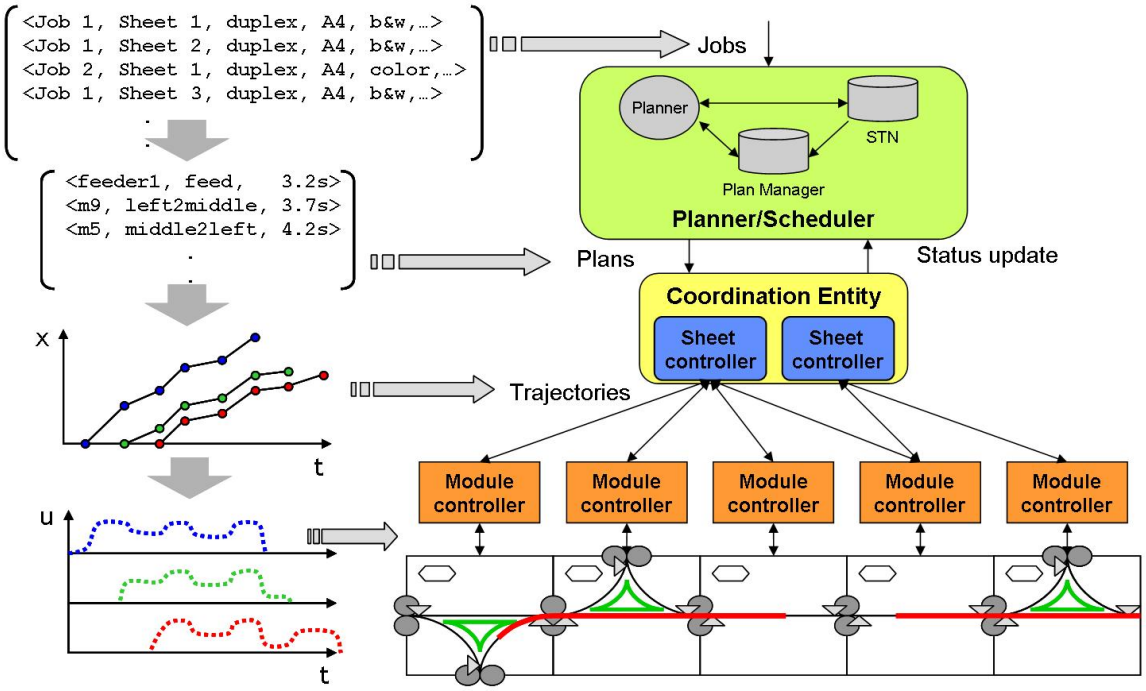


FIGURE 2.13 Architecture of the control software.

Figure 2.13 outlines our software architecture, which can be used to control manufacturing systems. As illustrated in Figure 2.13, job requests arrive asynchronously over time. A job request might consist of a set of sheets, which are described by several attributes. The Integrated Planner and Scheduler (Do, Ruml & Zhou 2008*b*), from now on just referred to as planner, translates the job requests into plans. A plan for printing an individual sheet is a linear sequence of actions, which routs each sheet through multiple modules. All sheets of a particular job have to arrive at the same finisher and sheet $n + 1$ is directly stacked on top of sheet n such that no other sheet of the same job or any different job is stacked between the two consecutive sheets. There may be many different plans that can be used to print a given sheet. Each module has a limited number of discrete actions it can perform, and for many of these actions the planner is allowed to control their duration. An action is identified by a triple: $\langle moduleID, actionID, start\ time \rangle$. The coordination entity consumes the plans and maintains a

virtual sheet controller for every plan or respectively for every sheet currently in execution. A sheet controller manages a dynamic group of module controllers currently acting on the plan e.g., signals downstream modules to get ready to handle the sheet. Figure 2.13 shows two sheets in the system (indicated as red lines along the paper path) and consequently there are two sheet controllers. Multiple sheet controllers might talk to the same module controller. e.g one sheet controller signals a module controller to get ready whereas another one talks to the same model controller to coordinate the speed of a sheet it currently acts on. The coordination entity translates discrete plans into continuous trajectories. A trajectory describes the speed of leading edge overtime. Given a trajectories, each module controller performs computation to adjust the motor speed to keep the sheet on its trajectory. In case the module controller is not able to achieve certain timing constraint, it propagates his information back to the sheet controller. The sheet controller then tries to compensate for the exception by adapting its trajectory. If that does not resolve the exception, the corresponding sheet stops and the planner is notified, which of the modules are jammed as result of the stoppage. In Section 2.7.5, exception handling is discussed in more detail and outline how it can be integrated into an overall framework. The intent of the brief introduction of the software architecture is to provide some background information on how the planner fits into the overall framework. The next section focuses on the planning task and planning technologies used in the manufacturing domain.

2.7.1 Planning and Scheduling

This section characterizes the manufacturing systems from a planning point of view and introduce the planning technologies used to control them. Those systems may run at high speed (up to several hundred pages per minute) and have to operate over a long period of time. A good example for such a manufacturing system is a high speed printer. At any given time, there might be multiple sheets of the same job or different jobs in the system. In addition, individual print engines might be of different engine types, e.g. a color engine can print both color and black&white images, whereas a mono engine can only print black&white images. Controlling these systems requires planning and scheduling of a series of job requests, which arrive asynchronously over time. This planning problem can be characterized using the model restriction introduced in Section 2.6.2.1. From a planning point of view, a printer system can be described as a finite, deterministic, static system with restricted goals and explicit time representation. Further it is assumed that the initial state is a single state and known to the planner. In addition, resource allocations are described, which incur from the constraint that prevents multiple sheets to be in the same location at the same time. This formulation is suitable to describe the nominal behavior of the system. Section 2.7.5 discusses exception handling in more detail and in this context the planning task representation is extended to capture non-deterministic system behavior, e.g. occurrence of a failure.

As a concrete example assume a print job is requested. The planner may decide to feed a blank sheet from any of the two feeders, then route it to any one of the four print engines (or through any two of the four engines in the case of duplex printing) and then send it to the correct output tray. This on-line planning problem is complicated by the fact that many sheets are in-flight simultaneously and the plan for the new sheet must not interfere with those sheets, as shown in Figure 2.14. Moreover, plan synthesis and plan execution are interleaved

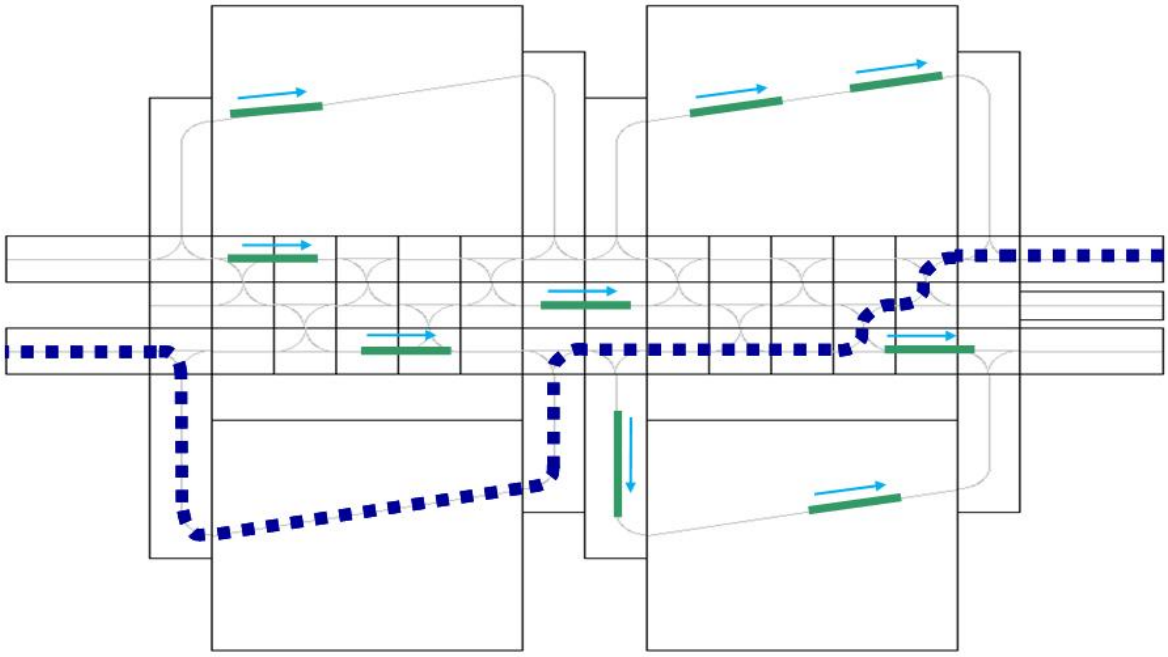


FIGURE 2.14 Planning and Scheduling Problem.

in real-time. Since the goal is to minimize wall clock end time, the speed of the planner itself affects the value of a plan.

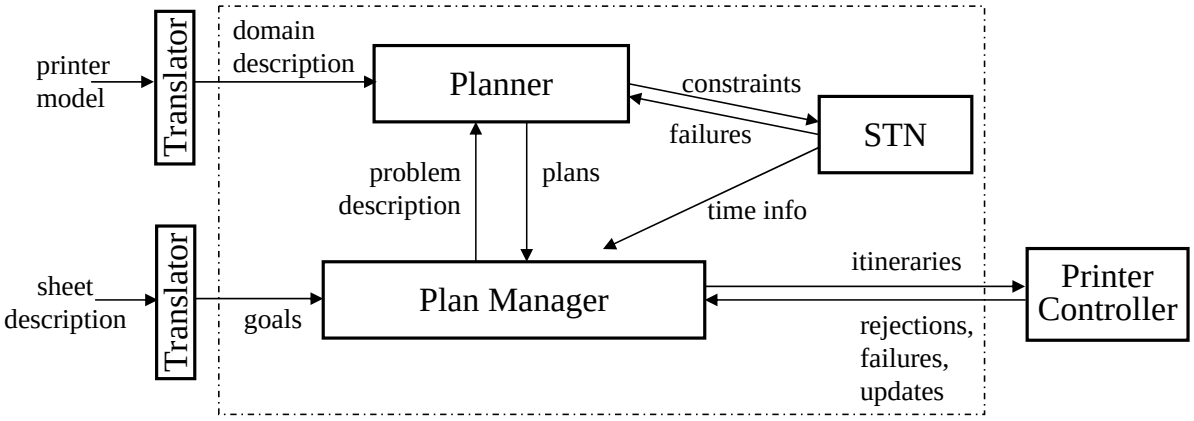


FIGURE 2.15 The system architecture, with the planning system indicated by the dashed box.

Figure 2.15 shows the core architecture of the Integrated Planner and Scheduler and how it communicates with the coordination entity. The major components are outlined below. The overall objective is to minimize the end time of all known sheets, ranging over all current print jobs. This can be approximated by optimally planning one sheet at a time. Figure 2.16 gives a sketch of the planning algorithm, which is referred to below.

On-linePlanner
1. plan the next sheet
2. if an unsent plan starts soon, then
3. foreach plan, from the oldest through the imminent one
4. freeze its time points to the earliest possible times
5. release the plan to the printer controller

PlanSheet
6. search queue ← {final state}
7. loop:
8. dequeue the most promising node
9. if it is the initial state, then return
10. foreach applicable operator
11. undo its effects
12. add temporal constraints
13. foreach potential resource conflict
14. generate all orderings of the conflicting actions
15 enqueue any feasible child nodes

FIGURE 2.16 Outline of the hybrid planner

2.7.2 Temporal Constraints and Plan Management

Given the rich temporal constraints in printer control, fast temporal constraint propagation, consistency checking, and querying are extremely important in our domain. Temporal constraints are maintained by using a Simple Temporal Network (STN) (Dechter, Meiri & Pearl 1991), represented by the box named STN in Figure 2.15. The network contains a set of time points t_i and constraints among them of the form $l_b \leq t_i - t_j \leq u_b$. The time points managed by the STN include action start and end times, as well as resource allocation start and end times, with the following constraints:

- (i) wall-clock action start time;
- (ii) the range of action start and end times;
- (iii) constraints between action start time and resource allocation by that action;
- (iv) conflicts for various types of resources.

For constraint propagation, a variation of the arc consistency algorithm (Cervoni, Cesta & Oddi 1994) is used.

The planner uses an A^* search strategy (Hart et al. 1968) that maintains multiple open search nodes. Each node is associated with a separate STN. Temporal constraints are added to the appropriate STN when a search node is generated. Whenever a new constraint is added, constraint propagation tightens the upper and lower bounds on the domain of each affected time point.

Lines 1–5 in Figure 2.16 correspond to the plan manager in Figure 2.15. After planning a new sheet, the plan manager checks the queue of planned sheets to see if there are any that could begin soon (line 2). If there are, those plans are released to the printer controller for execution. New temporal constraints are added that freeze the starting time of actions belonging to plans sent to the controller. These newly added constraints can cause significant propagations and in turn tighten the starting times of actions in the remaining plans.

The large number of potential plans for a given sheet and the close interaction through resource conflicts among plans for different sheets means that it is better to process scheduling constraints during the planning process. The planner uses state-space regression to plan each sheet, but maintains as much temporal flexibility as possible in the STN using partial orders between different actions in plans for different sheets. Therefore, it can be seen as a hybrid between state-space search and partial-order planning. Our approach is perhaps similar in spirit to that taken by the IxTeT system (Ghallab & Laruelle 1994).

2.7.3 Planning Individual Sheets

When planning individual sheets, the regressed state representation contains the (possibly partially-specified) state of the sheet. Best-first search (BFS) is used to find the optimal plan for the current sheet, in the context of all previous sheets. Because it must not interfere with existing plans, the state contains information both about the current sheet and previous plans. More specifically, the state is a 3-tuple $\langle$ *Literals, STN, R* $\rangle$, where

- *Literals* describes the regressed logical state of the current sheet. Those literals that are currently true are represented separately and those that are unknown, with false literals being represented implicitly.
- *STN* contains all known time points for the state and the current constraints among them. This includes constraints between different plans, between actions in the same plan, as well as against the wall-clock time.
- The resource profile *R* is the set of current resource allocations, representing the commitments made to plans of previous sheets and the partial plan of the current sheet.

After the optimal plan for a sheet is found, the resource allocations and *STN* used for the current plan are recorded passed back to the outer loop in Figure 2.16 and become the basis for planning the next sheet.

A major feature of our planner is that it seamlessly integrates planning and scheduling. Starting times of actions are not fixed but merely constrained by temporal ordering constraints in the STN. Note that line 14 of Figure 2.16 insists that any potential overlaps in allocations for the same resource are resolved immediately, resulting in potentially multiple children for a single action choice. This allows temporal propagation to update the action time bounds and guides plan search. While the plan for a single sheet is a totally-ordered sequence of actions, there are partial orders between actions that belong to plans of different sheets to represent the resource conflict resolutions.

2.7.4 Optimizing Productivity and Heuristic Estimation

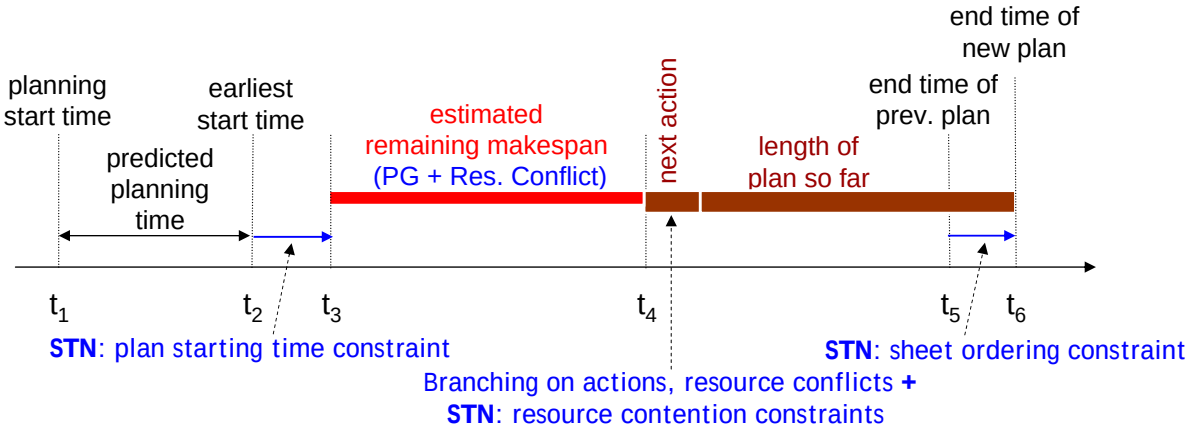


FIGURE 2.17 Important time points for evaluating a plan.

Our overall objective is to minimize the earliest possible end time for all planned sheets. In essence, this objective function minimizes the total time to finish all planned sheets and thus maximizes the printer's productivity. To support this, the primary criterion for evaluating the promise of a partial plan (line 8 in Figure 2.16) is the estimate of the earliest possible end time of the partial plan's best completion. To estimate this quantity, a simple lower bound on the additional makespan is computed, which is required to complete the current regressed plan. This heuristic value is indicated in Figure 2.17 by *estimated remaining makespan*. It is inserted before the first action in the current plan (t_4) and after the plan's earliest start time (t_2). By adding the constraint $t_2 < t_3$, the insertion may thus change the end time of the plan. It may also introduce an inconsistency in the temporal database, in which case the plan can safely be abandoned. Given that the current plan should end after the end time of all previous sheets in the same print job ($t_5 < t_6$), our objective function is to minimize t_6 without causing any inconsistency in the temporal database. Ties are broken in favor of smaller predicted makespan ($t_6 - t_3$) and then larger currently realized makespan ($t_6 - t_4$). This is analogous to breaking ties on $f(n)$ in BFS with larger $g(n)$, and encourages further extension of plans nearer to a goal. Because our heuristic is admissible, the plan found is optimal according to our objective function.

To estimate the duration required to achieve a given set of goals G from the initial state, dynamic programming is used over the explicit representation of the bi-level temporal on the planning graph (abbreviated as PG in Figure 2.17), in a manner similar to Temporal Graph-Plan (Smith & Weld 1999). To compute more accurate heuristic estimates in the presence of significant resource contention, resource conflicts are taken into account (Do & Ruml 2006*a*).

2.7.5 Exceptions during Operation

The previous sections discussed how autonomous technologies can be used to build autonomous printer. However, this discussion was limited to systems with nominal behavior. Many real-

world systems, such as hyper-modular, multi-engine printers, are not limited to nominal behavior. There are many reasons why a system might fail or behave abnormal, e.g. incorrect modeling, break down of system parts, unexpected interactions between different parts or with the environment, or simply wear and tear of components. All these reasons can be potential root-causes that disrupt nominal operation. The occurrence of abnormal behavior is called an exception.

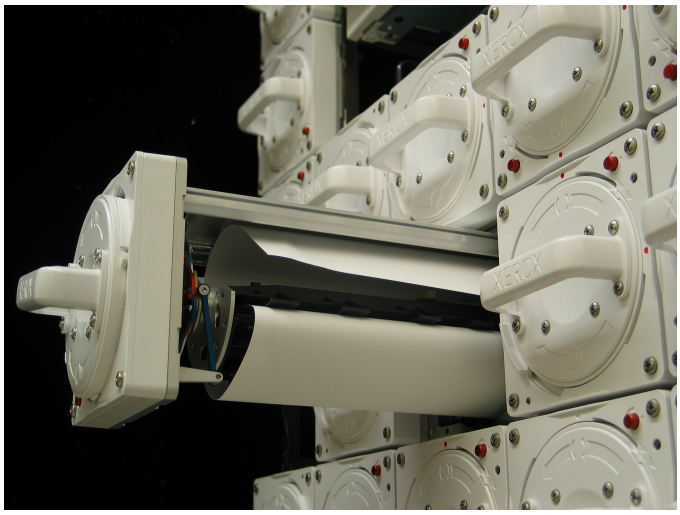


FIGURE 2.18 Clearance of a sheet after a paper jam occurred.

An exception can be characterized as a situation, in which predicted behavior deviates from actual behavior. Figure 2.18 shows an example of an exception, in which a sheet deviates from its trajectory by jamming within a system module. Usually after the appearance of an exception regular operation has to be disrupted for a special process to handle the exception. This process is called exception handling.

2.7.6 Exception Handling

The goal of exception handling is to automatically adapt the system to the current health conditions and to leverage the remaining parts of the system to achieve the highest performance possible. Ideally, exception handling is transparent to the user, minimize user intervention, and operates a system with minimal performance degradation. Exception handling introduces many hard problems, which are discussed in the remainder of this chapter. In general, exception handling strategies can be classified into two categories: reactive exception handling and proactive exception handling.

2.7.6.1 Reactive Exception Handling

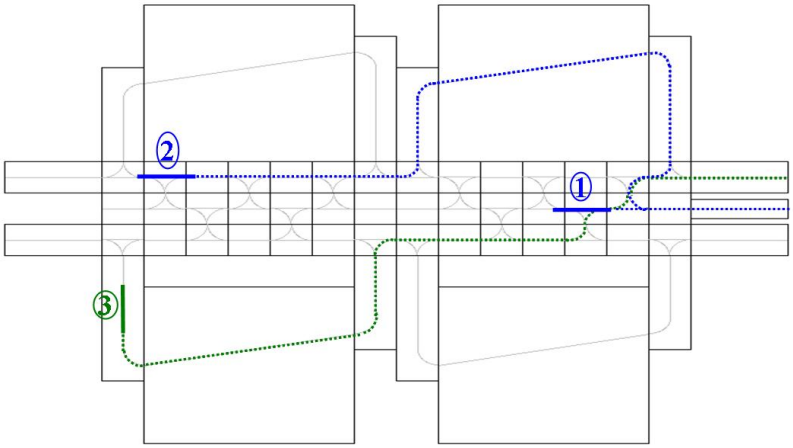
A Reactive Exception Handling Strategy reacts to symptoms by deciding the course of exception handling based on the symptoms. For example previous work in the context of hyper-modular, multi-engine printers (Fromherz, Bobrow & de Kleer 2003, Do et al. 2008*b*) describes a reactive exception handling that takes jammed modules offline and reroute consec-

utive sheets around the offline modules. Implementing such strategies is much harder than it appears to be. Since all plans tightly interact through various resource constraints an exception affecting any single plan can affect the executability of other plans. A plan that started out as a valid plan might become invalid due to the failing of another plan.

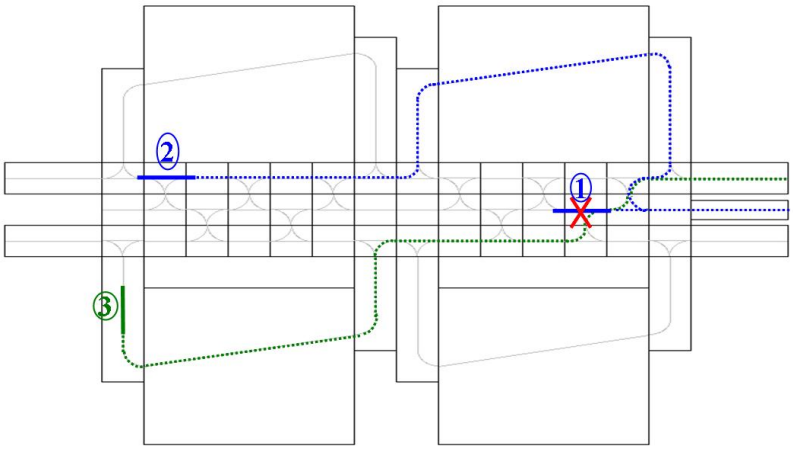
A concrete example is provided in Figure 2.19. Consider two print jobs. The first job consists of two sheets, sheet 1 and sheet 2. Both sheets are planned to go to the middle finisher. The dashed lines indicate the planned routes of the individual sheets. The second job consists of only one sheet, sheet 3, and is planned to arrive at the top finisher. The plans of both jobs are shown in Figure 2.19(a). The third finisher is a purge tray and can be used to exit sheets that not longer belong to any job. As discussed before, a job is correctly executed, if and only if all sheets of the job arrive at the same finisher and sheet $n + 1$ is directly stacked on top of sheet n with no other sheet in between sheet n and sheet $n + 1$. Consider that sheet 1 is jammed as indicated by the cross in Figure 2.19(b). Without intervention two further exceptions occur: sheet 2 arrives at the finisher before sheet 1, which violates the sheet order constraint of job 1; sheet 3 crashes into sheet 1, which fails job 2 and may cause more modules to be jammed. In order to avoid further exceptions the plans for sheet 2 and sheet 3 have to be changed. Reactive exception handling aims to reroutes consecutive sheets to prevent cascading exceptions and re-requests sheets or jobs that did not finish successful. Rerouting sheets online is a very hard problem, since the sheets do not stop or slow down until the original plan is replaced by a new plan. Since planning takes time, the position has to be predicted where the sheets will be at the end of the planning phase in order to determine the initial state for the planning phase. Earlier work describes how online re-planning can be done (Do, Ruml & Zhou 2008*a*). Given that new plans are found in the predicted time, a sheet can be rerouted starting from its predicted initial state. Figure 2.19(c) shows new plans for sheet 2 and sheet 3. Sheet 2 is rerouted to the purge tray to prevent violation of the sheet order constraint in job 1 and sheet 3 is rerouted around sheet 1 to avoid crashing. After the two sheets are rerouted, job 1 is re-requested and planned around the jammed module to the middle finisher. This leads eventually to a successful execution of both jobs.

The reactive exception handling strategy detects surface symptoms and adapts operation accordingly without performing detailed root-cause analysis. This strategy works well, if the root-cause is treated by reacting to symptoms. There are cases, however, where this fails. In those cases treating symptoms has no effect on underlying root-causes. In the printer domain, for example, the module that is causing an exception might differ from the module where symptoms are detected. In this case, the reactive strategy takes only those modules offline, where symptoms are detected, but leaves modules online, which causes the exception. In general, reactive exception handling treats symptoms independently from each other and assumes that treating the symptom also resolves the underlying root-cause. If this assumption is violated, for example in the case of symptoms with non-local root-causes, reactive exception handling fails.

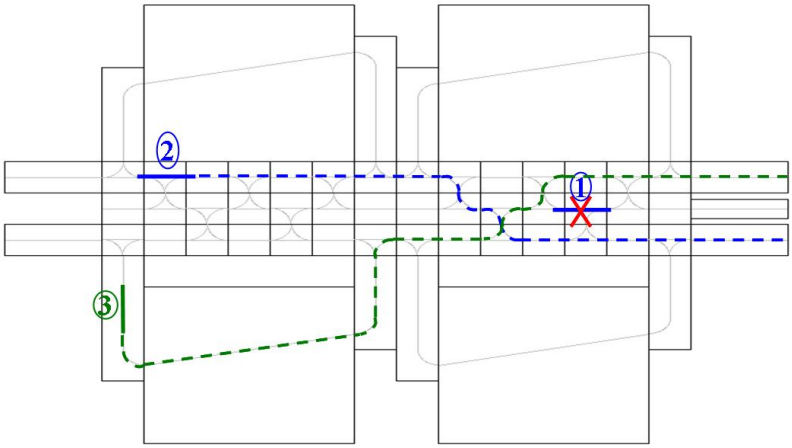
Consider the example provided in Figure 2.7.6.1. In this example reactive strategy fails to restore sustainable operation by simply reacting to surface symptoms. Consider three sheets, all originally planned to follow path 1. Assume that the module at location *CAUSE* creates a small scuff mark at the leading edge of a paper. Once a scuff mark exists, it builds up over time and causes a sheet to jam downstream. As a result, the first sheet jams along path 1 at location



(a)



(b)



(c)

FIGURE 2.19 Reactive Exception Handling realized by replanning and job re-requests.

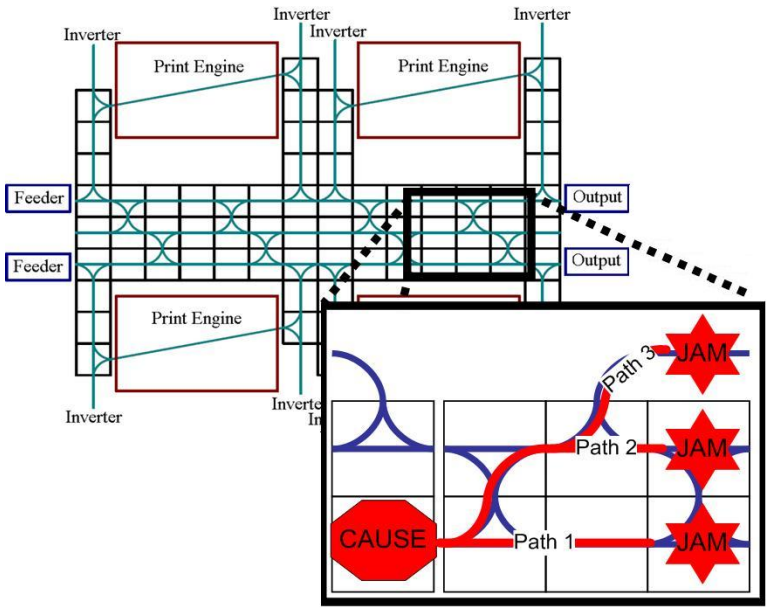


FIGURE 2.20 Reactive Exception Handling, exception handling without root-cause analysis, may fail given non-local root-causes scenarios.

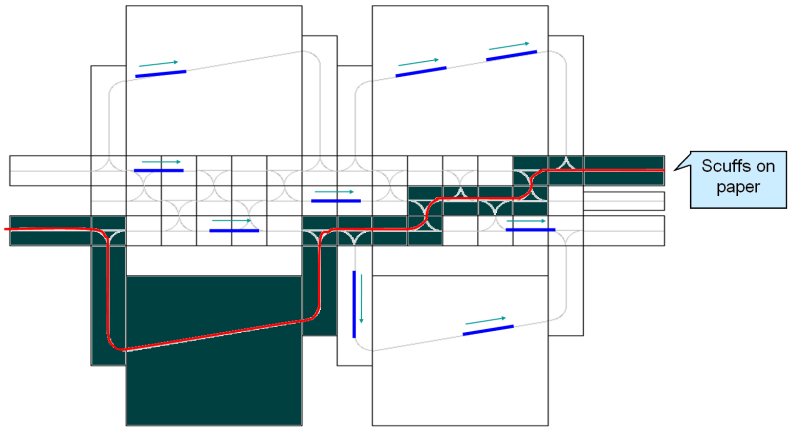
JAM. Following the reactive strategy, the jammed module is taken offline and the second sheet is rerouted around the jammed module, e.g. following path 2. However, path 2 still traverses the module causing the scuff mark. Therefore the paper jams along path 2. By now, two modules are jammed and taken offline. The reactive exception handling strategy continues by reacting just on the detected symptoms. Consecutive sheets are scheduled around the two jammed modules, but may still traverse the module causing the scuff mark. Consider that the third sheet then follows path 3 and jams. As a consequence the entire system is jammed. This scenario suggests that a purely reactive strategy might not always lead to high long-run performance.

The next section introduces an exception handling strategy that goes beyond reactive exception handling by performing root-cause analysis to isolate to true underlying root-causes of abnormal behavior.

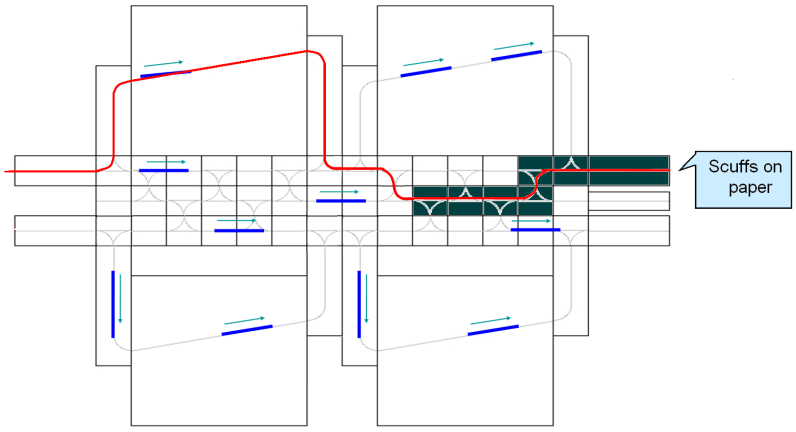
2.7.6.2 Proactive Exception Handling

Proactive Exception Handling differs from reactive exception handling as it not only reacts to exceptions, but additionally performs root-cause analysis to isolate the true underlying root-cause. Proactive Exception Handling reasons about the current health state in order to determine and eliminate the underlying root-cause of exceptions. Based on the gained knowledge the system then adapts future plans to compensate for the current conditions. This leads to higher long-term performance than a purely reactive strategy.

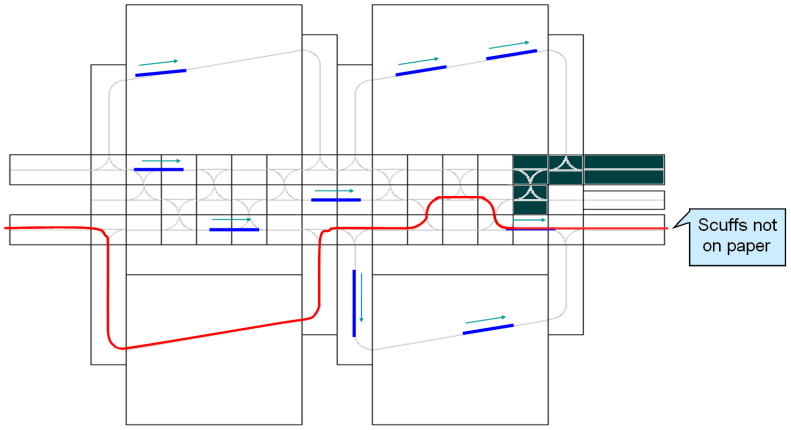
Figure 2.21 provides an example that illustrates the advantages of Proactive Exception Handling. Consider the example where a Hyper-modular, Multiple-engine Printer contains a sin-



(a) Plan execution with scuffed paper, thus all modules along the plan are suspected.



(b) Plan execution achieves suspected set reduction.



(c) Plan execution leads to valuable diagnostic information.

FIGURE 2.21 Proactive Exception Handling

gle, persistent fault, initially unknown to the system. The example is illustrated in Figure 2.21. The system prints a sequence of jobs with the overall objective to maximize long-run performance. Assume the plan illustrated in Figure 2.21(a) is executed and results in an observable fault, a scuffed paper. Without further root-cause analysis, one can only hypothesize, which of the modules has caused the damage. The scuff mark could have been cause by any module the sheet traversed. Even modules that interacted only indirect with the sheet could potentially be responsible for the scuff mark by triggering another module to scuff the paper. Such faults are called dependent faults (Weber & Wotawa 2008). For example a module can heat up and cause a neighboring module to fail. In general, any module could have triggered the scuff mark.

Reactive Exception Handling may decide to take all modules offline as response to the symptom. However, this seems not practical, since the system can not longer achieve goals. Alternatively, exception handling could take only those modules offline, which directly interacted with the sheet, assuming that some subset of them caused the scuff mark. Both suggested strategies do not perform root-cause analysis and may end up with a suboptimal long-term performance due to the lack of knowledge. Given an exception, the optimal response is to reconfigure the system such that the highest long-term performance can be achieved. A prerequisite of finding an optimal response strategy is to isolate the underlying root-cause of an exception. Only based on this knowledge the optimal response can be reliably determined. Even if the underlying root-cause can not be found, potentially reducing the set of hypotheses leads in average to higher long-term performance.

Proactive Exception Handling may chose to optimize further plans to gather additional information in order to isolate the underlying root-cause. Consider again the example in Figure 2.21(a). The set of suspected modules is highlighted in Figure 2.21(a). Since a fault was detected, Proactive Exception Handling optimizes future plans for long-run performance by increasing the potential information gain. Figure 2.21(b) illustrates an informative plan, which leads to a reduced suspected set. By continuing this strategy, the next plan execution, illustrated in Figure 2.21(c), leads again to a suspected set reduction. As a result only those modules need to be taken offline, which are still suspected. Even in the more general case of multiple faults, root-cause analysis can be used to reduce the number of suspected modules. As a consequence, Proactive Exception Handling takes in average less modules offline, which leads to higher long-term performance than a purely reactive strategy.

The process of root-cause analysis is called diagnosis and is the subject of the next chapter.

2.8 Conclusions

This chapter introduced the concept of a rational agent, an agent that does the right thing. It discusses in more detail how one can quantify what the right thing is and considers a performance measurement to address this question. It finally defines a rational agent to be an agent, that selects for each possible percept history an action such that the expected performance measure is maximized, given the evidence provided by the percept history and whatever built-in knowledge the agent has (Russell & Norvig 2009). After defining a rational agent it outlines various architectures to build such agents. The most sophisticated architecture is the general learning agent architecture (Russell & Norvig 2009), which enables an agent to gather and

leverage new knowledge from observations. Subsequently, an application of hyper-modular, mulit-engine printer illustrates a case in which the observations do not unambiguously suggest what an agent should learn. This leads to the consideration of a disambiguation process also referred to as diagnosis, which motivates the next chapter.

CHAPTER 3

Self-diagnosing Agent: The Integration of Active Diagnosis into a Planning Agent

In plan-driven systems, a planner uses a system model to create plans that achieve operational goals. The same system model can be used to diagnose exceptions during execution. Prior work has demonstrated that diagnosis can be used to adapt plans to compensate for component degradation. However, the sources of diagnostic information are severely limited. Diagnosis must either make inferences from observations during regular operation over which it has no control (passive diagnosis), or regular operation must be halted to introduce diagnostic-specific plans (explicit diagnosis). The declarative nature of model-based approaches allows the planner to potentially achieve operational goals in multiple ways. This chapter introduces a novel agent architecture, coined a Self-diagnosing Agent, which exploits this flexibility using a novel paradigm *pervasive diagnosis*. *Pervasive (active) diagnosis* constructs *informative operational plans* that simultaneously achieve operational goals, while uncovering additional diagnostic information about the condition of components. Section 3.1 introduces the general concept of a *Self-diagnosing Agent* and the overall framework. Section 3.2 defines diagnosis and outlines the diagnosis process, model-based diagnosis, and different paradigms of diagnosis such as passive and active diagnosis. Over the next sections, diagnosis is applied to planning agents. First Section 3.3 presents passive diagnosis for planning agents, followed by a probabilistic extension in Section 3.4. Active diagnosis is then described, first explicit diagnosis, in Section 3.5, and then pervasive diagnosis, in Section 3.6. To compare the different diagnosis paradigms, Section 3.7 develops a formal framework that evaluates the total response costs. Section 3.8 presents an overall framework to integrate active diagnosis with regular production. The theoretical results are verified in Section 3.9 on a hyper-modular, multi-engine printer. Finally, Section 3.10 concludes the chapter.

3.1 Introduction

Automated diagnosis is a key strategy for improving the reliability and maintainability of complex systems; it reduces labor overhead and leads to more rapid repairs. In remote applications, such as autonomous spacecraft operation, automated diagnosis can be used when human experts are unavailable. Autonomous systems perform tasks to achieve desired goals continuously over a long period of time without external guidance or intervention. This requires that autonomous systems know about their capabilities (model), reason about their course of action

with respect to their current conditions (planning), and reflect on their actual behavior to determine their current conditions (diagnosis). In general, autonomy can be seen as a combination of two processes:

- Diagnosis to determine the current conditions of the system.
- Planning a course of action to optimize system operation for the diagnosed conditions.

An autonomous system requires diagnosis to be integrated into regular operation. This chapter introduces a new architecture, coined a *Self-diagnosing Agent*, which realizes the integration by a novel diagnosis paradigm called *pervasive diagnosis*. Pervasive diagnosis actively manipulates the course of action during operation in order to gain diagnostic information without suspending operation. Consider a system where operational goals can be achieved in multiple ways. This flexibility can be exploited to generate operational plans that simultaneously gather information by trading off information gain with performance objectives. Therefore active diagnosis and regular operation occur at the same time leading to higher long-run performance than an integration of regular operation with passive diagnosis or alternating between explicit diagnosis and regular operation.

The conceptual framework of pervasive diagnosis is illustrated in Figure 3.1. It extends the planning framework shown in Figure 2.11.

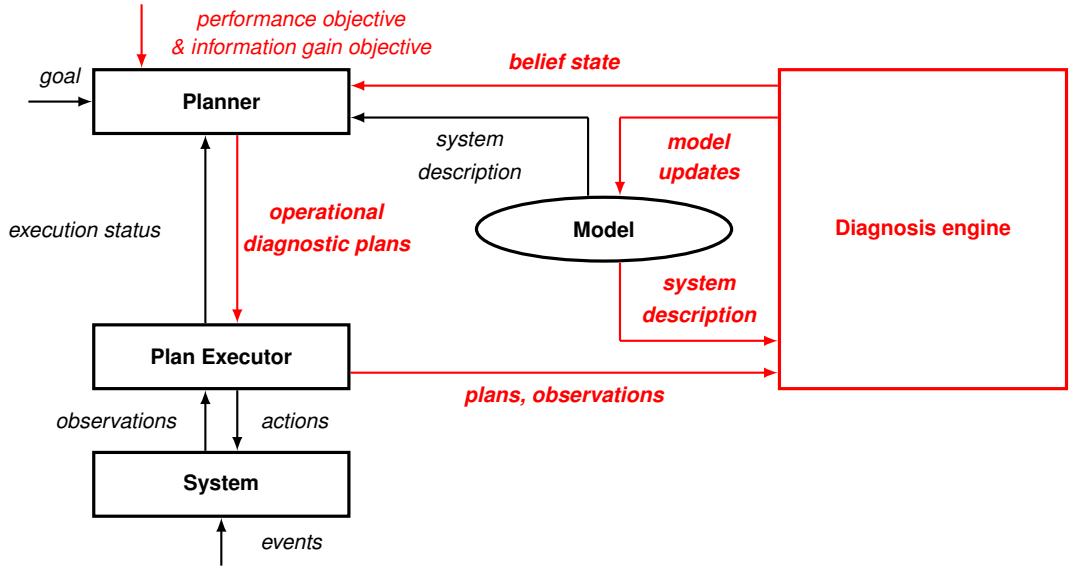


FIGURE 3.1 Pervasive Diagnosis: Integration of Operational Planning and Active Diagnosis.

The framework is implemented by a loop: The core idea of pervasive diagnosis is that plans simultaneously achieve operational goals and informative observations, coined informative operational plans. Those plans are executed and sent together with their corresponding observations to the diagnosis engine. The diagnosis engine updates its beliefs online to be consistent with the observations, updates the model, and forwards the beliefs to the planner. The planner then determines future plans based on current beliefs, performance objectives, and information

gain objectives. Systems that embody pervasive diagnosis benefit form high long-run performance by exploiting the overlap between operational plans and diagnostic plans. The space of all plans, operational plans, diagnostic plans, and their overlap is illustrated in Figure 3.2.

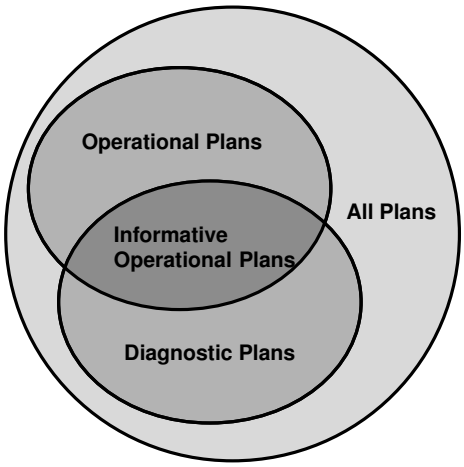


FIGURE 3.2 Pervasive diagnosis exploits the overlap between operational plans and diagnostic plans, so called *informative operational plans*.

However, not all systems can leverage pervasive diagnosis. In order to benefit form pervasive diagnosis the following three assumptions have to hold true:

- **Assumption 1:** A **sequence of plans** is executed and **observed**. Pervasive diagnosis is a sequential diagnosis technique where individual operational plans are optimized for information gain. For example, a continuous stream of print jobs can be optimized for root-cause analysis.
- **Assumption 2:** There are **multiple different ways** in which **operational goals can be achieved**. For example, consider that the same print job can be achieved by different routes through a print system. If no redundancy can be leveraged, it may be possible to alter action parameters such as execution speed.
- **Assumption 3:** The space of **operational plans** and **diagnostic plans intersect**. An observation process may not conflict with operational goals. A product, for example, that has to be cut open as part of the observation process is not longer a valid product.

A Self-diagnosing Agent extends the widely used framework of a Learning Agent by integrating pervasive diagnosis. The goal of a Learning Agent is to increase its understanding of itself and the world by learning to perform better than without learning (see Section 2.5.6). In general, a Learning Agent realizes this goal by identifying incorrect or incomplete knowledge and improving its knowledge by learning better predictive models directly from observations. In comparison, a Self-diagnosing Agent extends those capabilities by the ability to perform diagnosis. The core differentiator between a Self-diagnosing Agent and a Learning Agent is that a Self-diagnosing Agent has the ability to perform root cause analysis to improve its understanding to guide learning. Due to the integration of diagnosis an agent can determine the

root-causes of discrepancies between predicted and actual behavior before reasoning about how it learns. This analysis enables an agent to change its behavior with respect to the root-causes instead of continuously adapting to surface symptoms. This is particular important if the optimal reaction can only be inferred by reasoning over a set of symptoms. Suppose a scenario in which a root-cause can only be observed non locally. In this case, an agent that relies on pure learning without root-cause analysis may conclude that a local adaptation is needed. Only an agent that performs in-depth reasoning over a set of observations may correctly conclude that the root-cause is non local and therefore can only be compensated by a non local adaptation.

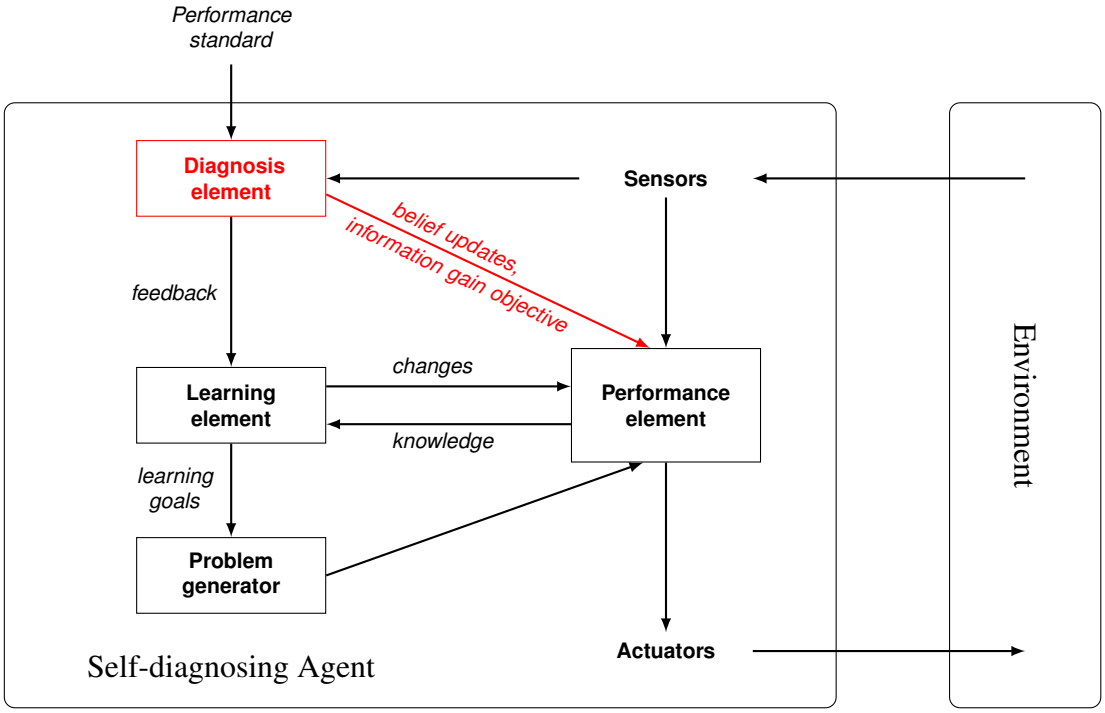


FIGURE 3.3 Self-diagnosing Agent

The overall framework of a Self-diagnosing Agent is illustrated in Figure 3.3. The framework complies with the general Learning Agent framework, except that the critics element is replaced by a diagnosis element, which leads to the following four components: a performance element, a diagnosis element, a learning element, and a problem generator. The ability to perform diagnosis enables the diagnosis element to provide diagnosis belief updates and information gain objectives to the performance element. The performance element can then generate informative operational plans to efficiently perform active diagnosis during regular operation to increase the agent knowledge about its current state. The main contribution of this work is an overall framework, which tightly integrates regular operation and active diagnosis. In particular, an information criterion is defined that quantifies how informative plans are, a plan generation algorithm is designed to derive informative operational plans, and a diagnosis framework is introduced to efficiently perform online diagnosis for systems that plan.

The core differentiator between a Self-diagnosing Agent and a Learning Agent is that a Self-diagnosing Agent performs root cause analysis and leverages the gain in understanding to improve the overall performance. An integral part of a Self-diagnosing Agent is the process of root-cause analysis, which is defined in the next section.

3.2 Diagnosis: Reasoning about Action

This section introduces the general concept of root-cause analysis, called diagnosis. The word diagnosis is derived through Latin from the Greek word $\delta\iota\acute{\alpha}\gamma\nu\omega\sigma\eta$ and is formed from $\delta\iota\acute{\alpha}$ (dia), meaning "apart", and $\gamma\nu\omega\sigma\eta$ (gnosi), meaning "to learn"'. In Greek, diagnosing means discriminating, distinguishing, or discerning between two or more possibilities. In general, the act of diagnosis describes a discrimination process, which is performed to determine the underlying root-cause for observed effects. The diagnosis process does not attempt to treat or cure, it is rather informational and exploratory in nature. The word diagnosis refers to a hypothesis that explains why a system differs from its nominal behavior. Consider a person with increased temperature. The hypothesis *person has fever* is one possible diagnosis, which explains the discrepancy between the nominal behavior and the observed behavior. The process of diagnosis can be viewed as the interaction between observations and predictions. Observations capture the actual system behavior, whereas predictions capture the nominal behavior deduced from some system model. A failure is presumed to be present if predictions and observations differ from each other.

In engineering, the process of diagnosis is a reasoning process, which explains discrepancies between predicted behavior and observed behavior. The system model, from which the nominal behavior is drawn, is an integral part of diagnosis. Without such a model, diagnosis is not able to perceive discrepancies from nominal behavior, as it lacks knowledge of how the system behaves nominally.

Conceptually, the process of diagnosis can be separated into three subtasks, which are common among all diagnosis systems: hypotheses generation, testing, and discrimination.

- The **hypothesis generation** task maps symptoms to a set of hypotheses. Given the discrepancies between predicted behavior and observed behavior, this subtask generates a set of possible hypotheses.
- The second subtask is **hypothesis testing**. After a set of hypotheses is generated, the testing process filters out any hypotheses, which can not explain the gathered observations. This leads to a subset of hypotheses, where each is able to account for all observed observations.
- The task of **hypothesis discrimination** is concerned with discriminating among the remaining set of hypotheses using some kind of discrimination technique, e.g. process of elimination.

In a practical setting, it is likely that diagnosis subtasks are interweaved to improve efficiency. However, an interweaved approach is conceptually similar to an approach where the subtasks are performed independently.

3.2.1 Model-based Diagnosis

Model-based diagnosis refers to a subfield of artificial intelligence that studies diagnosis computationally. It seeks to perform diagnosis on systems, possible complex systems, without human intervention. Model-based diagnosis has a long history in artificial intelligence and engineering, including logic based frameworks (Reiter 1992), continuous non-linear systems (Rauch 1995), and hybrid logical probabilistic diagnosis (Poole 1991). It has been deployed in the context of spacecrafts (Williams & Nayak 1996), xerographic systems (Zhong & Li 2000), automotive systems (Cascio, Console, Guagliumi, Osella, Panati, Sottano & Dupré 1999) and to determine optimal placement of sensors during design (Mauss, May & Tatar 2000).

The remainder of this section discusses the general concepts and definitions of model-based diagnosis. In order to exposit the key concepts consider a simple logic example. After the basic concepts of model-based diagnosis are introduced with the logic example, the printer example is used to outline how diagnosis can be integrated into systems which use planning.

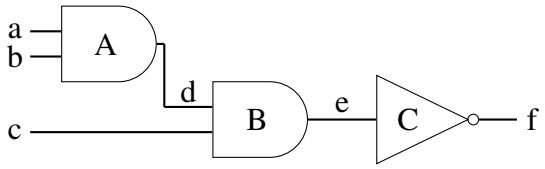


FIGURE 3.4 Example circut, SMALLY, with two and-gates and one inverter-gate.

Consider the logic circuit, *SMALLY*, illustrated in Figure 3.4. It can be modeled using a component set, a system description, and observations. The component set specifies the components of the overall system. The system description, on the other hand, specifies the behavior of the individual components and how they interact with each other. Observations describe the actual behavior of the system. Formally:

Definition 5. *An observable system is a triple* $(SD, COMPS, OBS)$ *where*

- SD*, system description, is a set of first-order sentences,*
- $COMPS$*, components, is a set of constants,*
- OBS*, observations, is a set of first-order sentences.*

Typically, the system description SD organizes the knowledge by maintaining a component library CL and a system topology ST. The component library CL captures the behavior of the individual components and the system topology ST describes the system's components, their type, and how they are connected. Generally, it can not be assumed that a system description SD is organized in any particular way, but it can be assumed that the SD intends to capture the behavior and the structure of the system. Note, that it says 'intends', as a system description might be incomplete. The case of diagnosing systems with incomplete models is addressed in Chapter 6.

SMALLY contains two and-gates A and B, and one inverter C. In Figure 3.4 solid lines illustrate how the components are connected. Consider there is a single fault in the system, causing the system to behave abnormal if and only if component C is involved. For example, if A real world scenario could be that component C is late in propagating its signal. Further assume that there are no intermittent faults, which means exercising a faulty component always leads to an observable abnormality. This is not a general restriction to model-based diagnosis (de Kleer 2007*a*), but it makes the example more comprehensible.

The set of components $COMPS$ consists of three gates, as shown in Figure 3.4, therefore $COMPS = \{A, B, C\}$. The system topology ST for the example is shown in Equation 3.1.

$$\begin{aligned} ST \quad = \{&And(A) \wedge And(B) \wedge Inv(C), \\ &a \equiv in(A,1) \wedge b \equiv in(A,2) \wedge out(A) \equiv d, \\ &d \equiv in(B,1) \wedge c \equiv in(B,2) \wedge out(B) \equiv e, \\ &e \equiv in(C,1) \wedge out(C) \equiv f\} \end{aligned} \tag{3.1}$$

To indicate the health state of a component the concept of an abnormal component is defined using an AB-literal. Those AB-literals are used to formalize the component behavior in component library CL.

Definition 6. *An AB-literal indicates the health of a component. Given a component $x \in COMPS$, the AB-literal can be either $AB(x)$ or $\neg AB(x)$, where $AB(x)$ represents that component x is ABnormal (faulted) and $\neg AB(x)$ indicates that x is not ABnormal, thus normal.*

The component library CL describes the behavior of the individual components. The component library CL for the example is shown in Equation 3.2.

$$\begin{aligned} CL \quad = \{&And(x) \rightarrow [\neg AB(x) \rightarrow [in(x,1) \wedge in(x,2) \equiv out(x)]], \\ &Inv(x) \rightarrow [\neg AB(x) \rightarrow [in(x,1) \equiv \neg out(x)]]\} \end{aligned} \tag{3.2}$$

The system description SD is the union of the component library CL and the system topology ST, as shown in Equation 3.3.

$$SD \quad = \quad CL \cup ST \tag{3.3}$$

A complete assignment over all components or respectively over all corresponding health AB-literals, to either abnormal (true, T) or not abnormal (false, F), is called a health assignment. A special case is the assignment that assigns not abnormal to all AB-literals, denoted $\neg AB^*$. The $\neg AB^*$ is defined in Definition 7.

Definition 7. *The $\neg AB^*$ assigns not abnormal to all AB-literals. Formally,*

$$\neg AB^* \quad = \quad \{ \bigwedge_{c \in COMPS} \neg AB(c) \}. \tag{3.4}$$

In the absence of failures, the $\neg AB^*$ together with system description SD and observations OBS is consistent, as defined in Definition 8.

Definition 8. *A set of observations OBS is consistent with a system description SD if and only if the following sentence is satisfiable:*

$$SD \cup OBS \cup \neg AB^* \tag{3.5}$$

Assume observation obs_1, observing a, b, c, and f, is collected

$$obs_1 \quad = \quad [a \equiv 1 \wedge b \equiv 1 \wedge c \equiv 1] \rightarrow f \equiv 1. \tag{3.6}$$

Given observation obs_1 and the system description SD, one can evaluate if the predicted behavior is consistent with the observed behavior. In our example, the predicted behavior is not consistent with observation obs_1. System description SD together with assignment $\neg AB^*$ imply that from $a \equiv b \equiv c \equiv 1$ follows $d \equiv 1$, $e \equiv 1$, and $f \equiv 0$. The predicted value for f is therefore 0, but the observed value is 1. The difference is called a discrepancy. Each discrepancy between a predicted and observed behavior indicates one or more components behaving abnormal. Intuitively, a conflict indicates a set of components, which can not all be functional at the same time, given some observation. Based on system description SD and observation obs_1 one can infer the present of a conflict. Before the concept of a conflict is defined, AB-clauses are introduced to be able to express dependencies among AB-literals.

Definition 9. *An AB-clause is a disjunction of AB-literals containing no complementary pair of AB-literals.*

Given Definition 9 a conflict is defined as follow:

Definition 10. *A conflict in $(SD, COMPS, OBS)$ is any AB-clause, which is entailed by $SD \cup OBS$.*

In our running example, the following AB-clause is a conflict:

$$SD \cup \{obs_1\} \models AB(A) \vee AB(B) \vee AB(C). \tag{3.7}$$

The task of diagnosis is concerned with finding a set of health assignments that make SD and OBS consistent. Intuitively, a diagnosis is a health assignment that explains the observed symptoms or logically speaking makes SD and OBS consistent. Formally, a diagnosis is defined by the following two definitions:

Definition 11. *Given two sets of components, B and G, $D(B, G)$ is defined to be the conjunction:*

$$\left[\bigwedge_{c \in B} AB(c)\right] \wedge \left[\bigwedge_{c \in G} \neg AB(c)\right] \tag{3.8}$$

where $AB(x)$ is the AB-literal of x.

Definition 12. *Let $\Delta \subseteq COMPS$. A diagnosis for $(SD, COMPS, OBS)$ is $D(\Delta, COMPS - \Delta)$ such that the following is satisfiable*

$$SD \cup OBS \cup \{D(\Delta, COMPS - \Delta)\} \tag{3.9}$$

For brevity, diagnoses are denoted by their Δ so $\Delta = \{A, B\}$ represents $D(\Delta, COMPS - \Delta) = D(\{A, B\}, COMPS - \{A, B\})$. List 3.10 shows all valid diagnoses based on observation obs_1 ordered by cardinality. The cardinality of a diagnosis is defined in Definition 13.

Definition 13. *The cardinality of a diagnosis $D(\Delta, COMPS - \Delta)$, denoted $|D(\Delta, COMPS - \Delta)|$ or in short $|D|$, is the number of elements in Δ.*

single fault diagnoses:

$\Delta_1 = \{A\}, \quad \Delta_2 = \{B\}, \quad \Delta_3 = \{C\},$

double fault diagnoses:

$\Delta_4 = \{A, B\}, \quad \Delta_5 = \{A, C\}, \quad \Delta_6 = \{B, C\},$

triple fault diagnoses:

$\Delta_7 = \{A, B, C\}$ (3.10)

The set of diagnoses can be reduced by a more constrained definition of diagnosis, coined minimal cardinality diagnosis:

Definition 14. *A diagnosis $D(\Delta, COMPS - \Delta)$ for $(SD, COMPS, OBS)$ is a minimal cardinality diagnosis if and only if there exists no other diagnosis $D'(\Delta', COMPS - \Delta')$ such that $|\Delta'| < |\Delta|$.*

List 3.11 shows all valid minimal cardinality diagnoses. The minimal cardinality in our example is currently 1, yet it can not be concluded that there is only one failure in the system. The only conclusion that can be drawn is that there exists at least one failure in the system.

single fault diagnoses:

$\Delta_1 = \{A\}, \quad \Delta_2 = \{B\}, \quad \Delta_3 = \{C\}$ (3.11)

Suppose another observation obs_2 is collected:

$$obs_2 \quad = \quad [a \equiv 1 \wedge b \equiv 1] \rightarrow d \equiv 1. \tag{3.12}$$

Based on the two collected observations, obs_1 (Equation 3.6) and obs_2 (Equation 3.12), it can be deduced that a fault in component A individually can not explain the discrepancy. Recall, there are no intermittent faults in our example. This reduces the set of minimal cardinality diagnoses to:

$$\begin{array}{c} \textit{single fault diagnoses:} \\ \Delta_2 = \ \{B\}, \ \ \Delta_3 = \ \{C\}, \end{array} \tag{3.13}$$

To illustrate the diagnosis framework, assume that another two observations obs_3 and obs_4 are made, as shown in Equation 3.15.

$$\begin{array}{lcl} obs_3 & = & [d \equiv 1 \wedge c \equiv 1] \rightarrow e \equiv 1 \\ obs_4 & = & [e \equiv 1] \rightarrow f \equiv 1 \end{array} \tag{3.14}$$

Based on the observations obs_1, obs_2, and obs_3, it can be derived that neither component A individually nor component B individually can explain the discrepancy. This reduces the set of all minimal cardinality diagnoses to the list:

$$\begin{array}{c} \textit{single fault diagnoses:} \\ \Delta_3 = \ \{C\}, \end{array} \tag{3.15}$$

Once our reasoning includes all available observations, obs_1, obs_2, obs_3, and obs_4, the current diagnosis is verified. At this point the diagnosis framework converges to the following health diagnosis:

$$\begin{array}{rl} D(\{C\}, COMPS - \{C\}) & = \left[\bigwedge_{c \in \{C\}} AB(c)\right] \wedge \left[\bigwedge_{c \in COMPS - \{C\}} \neg AB(c)\right] \\ & = AB(C) \wedge \neg AB(A) \wedge \neg AB(B) \end{array} \tag{3.16}$$

Diagnosis, as defined in this section, is a formal framework using first-order logic as described in (Reiter 1987) and (de Kleer, Mackworth & Reiter 1992). The size of the initial diagnosis space is exponential in the number of system components. Any component can either behave abnormal or not abnormal, thus the diagnosis space in our example consists initially of $2^3 = 8$ diagnoses. As with all model-based frameworks, model-based diagnosis is computationally explosive if the implementation follows naively the definitions. Chapters 5 and 4 show how the computational complexity can be drastically reduced. The goal of diagnosis is to isolate a set of health assignments, which are consistent with the collected observations. Recall, that each conflict is a set of components, which contains at least one abnormal component. Thus each diagnosis must have a non-empty intersection with every single conflict. This observation is a key element to the algorithms used to develop efficient diagnosis algorithms.

The next section introduces different diagnosis strategies based on how proactive observations are gathered. After that the individual strategies are compared based on a theoretical evaluation framework.

3.2.2 Classes of Diagnosis

The task of diagnosis is to detect and isolate the underlying root-causes from observed sensor data. In that process the information content of gathered observations is crucial to effective diagnosis. In general, diagnosis can be categorized into two classes based on how proactive they gather information: passive diagnosis and active diagnosis.

3.2.2.1 Passive Diagnosis

In passive diagnosis π_{pass} the diagnostic reasoner gathers information purely through passively observing (or monitoring) the system. Passive diagnosis performs diagnosis without actively participating in the process of selecting actions. The system is assumed to generate a sequence of actions according to some objective. This sequence of actions results in observations which are consumed by the diagnostic reasoner. Note, that a passive diagnosis engine has no influence on the plan generation process. As a result the course of action is typically optimized for some performance objective, but not to maximize diagnostic information. The remote agent project (Muscettola et al. 1998) which is responsible for diagnosing, planning and repairing spacecraft is one of the most sophisticated and well developed examples of passive diagnosis used to inform planning, search and repair.

3.2.2.2 Active Diagnosis

Active diagnosis π_{act} differs from passive diagnosis, which purely monitors the system, by the ability to influence the action selection process. In more detail, the action selection process depends at least partly on the current belief state. As a result the course of action can be optimized for information gain leading to more informative observations. From an active diagnosis point of view, this imposes three challenges on the action selection process:

- **Information criterion:** Which criterion determines how informative an action is?
- **Selecting informative actions:** How to select an informative course of action?
- **Integrating diagnosis:** How to integrate active diagnosis into regular operation?

The first challenge is to define an information criterion. The second challenge is to generate plans with maximum information gain. The information criterion is briefly discussed in Section 3.5.1. The generation of informative plans motivates Section 3.5.2 and Chapter 7. The third challenge addresses how active diagnosis and system operation are combined, which motivates the rest of this section. There are two active diagnosis strategies explicit diagnosis and pervasive diagnosis, which differ in the way on how diagnosis is integrated into operation. Both strategies are outlined over the next two sections.

3.2.2.3 Explicit Diagnosis

In the case of explicit diagnosis π_{exp}, the action selection is purely optimized for information gain. Actions are selected based on how much diagnostic information they contribute independently of any operational goal. In explicit diagnosis, the optimal sequence of actions maximizes the diagnostic information and therefore minimizes the ambiguity among the hypotheses. Calculating such probe sequences has been demonstrated for static circuit domains (de Kleer & Williams 1987). This can be extended to sequential circuits with persistent state and dynamics through time-frame expansion methods described in (Bushnell & Agrawal 2000) or (de Kleer 2007*c*). Finding explicit diagnosis tests can be done as a SAT formulation (Ali, Veneris, Safarpour, Abadir, Drechsler & Smith 2004). The use of explicit diagnosis jobs has been suggested for model-based control systems (Fromherz 2007). The combination of explicit diagnosis to obtain information and model-based control conditioned on the information has appeared in many domains including automatic compensation for faulty flight controls (Rauch 1995), choosing safe plans for planetary rovers (Dearden & Clancy 2002), maintaining wireless sensor networks (Provan & Chen 1999) and automotive engine control (Kim, Rizzoni & Utkin 1998). Section 3.5 goes into more detail on how explicit diagnosis can be realized in the context of a planning agent.

3.2.2.4 Pervasive Diagnosis

Pervasive diagnosis π_{per} is a new paradigm, in which regular operation is actively manipulated to maximize diagnostic information. Active diagnosis and regular operation can therefore occur simultaneously leading to higher long run performance than passive diagnosis or alternating active diagnosis with regular operation. The integration of diagnostic goals in an operational strategy results in informative operation. The primary objective in informative operation is to continue operation. In the case that there are various ways to achieve operational goals, informative operation simultaneously maximizes diagnostic information. Regular operation might be optimized towards time efficient operation, cost efficient operation, robust operation, or any combination of those. All of those share the primary objective of achieving operational goals, but differ in the way how they approach a goal. In all named approaches the set of goal achieving strategies are ranked by an objective function and the highest ranking strategy dominates. For example, in time efficient operation, strategies are ranked by time and the most time efficient strategy dominates. Similar to other operation strategies, informative operation, ranks the set of goal achieving plans by their potential of information gain and selects the most promising strategy. Details on how this can be realized are outlined in Section 3.6.

3.3 Passive Diagnosis for Planning Agents

A first step towards a Self-diagnosing Agent is to formulate how a planning agent can be diagnosed. A planning agent uses a system description to generate goal achieving plans. The same description can be leveraged by diagnosis to infer the underlying diagnostic state. A planning system is usually represented as an observable state-transition system (see Section 2.6.1.2,

Definition 4). An observable state-transition system is a 6-tupel $\Sigma^{obs} =< \mathcal{S}, \mathcal{A}, \mathcal{E}, T, \Phi >$, where:

- $\mathcal{S}$ is a finite set of states;
- $\mathcal{A}$ is a finite set of actions;
- $\mathcal{E}$ is a finite set of events;
- $T : \mathcal{S} \times \mathcal{A} \times \mathcal{E} \rightarrow \mathcal{S}$ is a state-transition function;
- $\Phi : \mathcal{S} \rightarrow 2^{\mathcal{S}}$ is an observation function.

Generally, a planning agent attempts to derive a plan p such that executing plan p evolves the system from an initial state s_0 to some state s_k and s_k is a goal state, $s_k \in \mathcal{S}_g$. A not abnormal plan execution evolves a system into a goal state. In the case of a plan execution that does not terminate in a goal state, the system has deviated from its normal execution behavior. To determine if a plan execution behaved not abnormal or abnormal an observation can be performed. The observation function Φ maps the underlying system state $s \in \mathcal{S}$ to a set of states $obs \in 2^{\mathcal{S}}$, called observation. An observation represents all states an underlying system can possibly be in. If the system is fully observable, the observation function Φ maps a state s to a set of states that contains only state s. Executing a plan results in an execution observation, this indicates if an execution behaves not abnormal or abnormal. If the execution behavior is not abnormal, an observation obs contains at least one goal state, otherwise the execution behavior is abnormal. This leads to the following definition:

Definition 15. *Assume a system as a 6-tuple $\Sigma_{obs} =< \mathcal{S}, \mathcal{A}, \mathcal{E}, T, \Phi >$, an initial state $s_0 \in \mathcal{S}$, a set of goal states $\mathcal{S}_g \subseteq \mathcal{S}$, and a plan p are given. Executing plan p behaves not abnormal if and only if the execution evolves the system from the initial state s_0 to some state s_k such that the corresponding observation $obs = \Phi(s_k)$ contains at least one goal state, $\exists s_g \in obs$ s.t. $s_g \in \mathcal{S}_g$, otherwise the execution of plan p behaves abnormal.*

Based on Definition 15 an execution observation function $\mathcal{Y}$ can be defined as follows:

Definition 16. *Given the set of all plans $\mathcal{P}$, an execution observation function $\mathcal{Y} : \mathcal{P} \rightarrow \{0, 1\}$ is a function that maps a plan $p \in \mathcal{P}$ to a binary observation, where $\mathcal{Y}(p) = 1$ if and only if plan p executed not abnormal and $\mathcal{Y}(p) = 0$ if and only if plan p executed abnormal.*

In order to diagnose a planning system descript as a 6-tuple $\Sigma^{obs} =< \mathcal{S}, \mathcal{A}, \mathcal{E}, T, \Phi >$ it remains to be shown how it can be formulized as an observable system. As defined in Section 3.2.1, an observable system is a triple $(SD, COMPS, OBS)$, where

- SD, system description, is a set of first-order sentences,
- $COMPS$, components, is a set of constants,
- OBS, observations, is a set of first-order sentences.

The idea of the presented formulation is to model a planning system as a signal processing system. In general, executing a plan evolves a system from an initial state into some state and the corresponding execution observation indicates if the execution behaved not abnormal or abnormal. Figuratively, a plan execution can also be viewed as propagating a signal from an initial state to some state and the corresponding execution observation indicates if the signal propagation behaved not abnormal or abnormal. Assuming that not abnormal signal propagation does not manipulate the signal, a propagation behaved abnormal if the signal differs between the initial state and the state the plan execution terminates in. In planning, an abnormal plan execution is cause by one or multiple abnormal action executions. Hence, if a plan execution behaved abnormal, at least one action execution behaved abnormal. Figuratively, if a signal got manipulated during propagation at least one link (action) must have manipulated the signal. The task of diagnosis is to infer a complete health assignment over all actions. The set of components $COMPS$ is the set of actions $\mathcal{A}$, formally:

$$COMPS = \{a | a \in \mathcal{A}\}. \tag{3.17}$$

The system description SD is decomposed into a component library CL and a system topology ST.

$$SD = CL \cup ST \tag{3.18}$$

The component library CL captures the behavior of the individual components. In the here introduced formulation, there exists only one type of component, an action, formally:

$$CL = \{action(a) \rightarrow [\neg AB(a) \rightarrow in(a) \equiv out(a)]\} \tag{3.19}$$

Note, that the component library CL describes an action as a signal propagating component that outputs the same signal as inputted if the component behaves not abnormal.

The system topology ST describes the actual existing components and their type. Typically, the system topology also describes the connections between the individual components. In the context of planning, connections represent the order in which actions can be executed in. In planning there are often to many combinations in which actions can be executed in to be represented explicitly (see Section 2.6.2.2). Therefore, the connections are only captured if this information is relevant for diagnosis. In the context of diagnosis, a connection is relevant if it was part of an observed plan execution. In the case where a plan has been executed and observed, the connections are represented as part of the observation formulation. Given that there is only one type of component, that the set of components $COMPS$ is known, and that connections are not described as part of the system topology, the resulting system topology ST is formularized as follows:

$$ST = \left\{ \bigwedge_{a \in COMPS} action(a) \right\} \tag{3.20}$$

The next step is to formulise observations. Consider a plan $p =< a_1, a_2, \ldots, a_k >$ and a corresponding execution observation $\mathcal{Y}(p)$, the corresponding observation in the context of an observable system can be formulized as:

$$obs(p) = \left\{ 1 \equiv in(a_1) \wedge out(a_k) \equiv \mathcal{Y}(p) \wedge \bigwedge_{a_i,a_j \in p, s.t. j=i+1} out(a_i) \equiv in(a_j) \right\} \quad (3.21)$$

Note, that observations describe the structure of the executed plan p and set the input signal of the first action to 0, and the output signal of the last action to the corresponding execution observation $\mathcal{Y}(p)$. The set of observations results from a sequence of executed plans and their corresponding execution observations.

A planning system can therefore be mapped to an observable system as defined in Definition 17:

Definition 17. *Given a system as a 6-tuple $\Sigma_{obs} =< \mathcal{S}, \mathcal{A}, \mathcal{E}, T, \Phi >$, a sequence of executed plans $\mathcal{P} =< p_1, p_2, \ldots, p_n >$, and an execution observations function $\mathcal{Y}$ the resulting observable system is defined as a triple $(SD, COMPS, OBS)$, where*

- $SD = CL \cup ST$, *where*
  $CL = \{action(a) \rightarrow [\neg AB(a) \rightarrow in(a) \equiv out(a)]\}$
  $ST = \left\{\bigwedge_{a \in COMPS} action(a)\right\}$
- $COMPS = \{a | a \in \mathcal{A}\}$
- $OBS = \bigcup_{p_i \in \mathcal{P}} obs(p_i)$, *where*
  $obs(p) = \left\{1 \equiv in(a_1) \wedge out(a_k) \equiv \mathcal{Y}(p) \wedge \bigwedge_{a_i,a_j \in p, s.t. j=i+1} out(a_i) \equiv in(a_j)\right\}$.

Given the introduced mapping, diagnosing a planning system is straightforward by following Section 3.2.1. In summary, this section proposes a fundamentally new approach to diagnose a planning system. As with all model-based frameworks, it is computationally explosive if directly implemented as described in the definitions. Many systems have a set of actions, which is to big to be represented explicitly (see Section 2.6.2.2). This problem can be avoided by generating the set of components $COMPS$ incrementally. An incremental approach starts with an empty component set $COMPS$, hence $COMPS = \emptyset$. Executing a plan implies executing actions. An action is dynamically added to the set of components $COMPS$ once it was executed. This ensures that all necessary actions are represented in the component set, but prevents a computational explosion.

Up to this point only logical diagnosis has been content of the discussion. The next section extends diagnosis to a probabilistic framework.

3.4 Probabilistic Diagnosis for Planning Agents

The task of diagnosis, as defined in Section 3.2.1, is to infer a health assignment that explains the observed symptoms or logically speaking makes a system description SD and a set of observations OBS consistent. At any given time, there might exist multiple diagnoses and

not all of those diagnoses must have the same likelihood of being true. This section considers probabilistic diagnosis for planning systems.

Let X be the underlying diagnosis state and $\mathcal{Y}(p)$ be the observation resulting from executing plan p. For example, the bit-vector $X = 011000$ indicates that only the second and third component are faulted. The observation is a single bit indicating an abnormal execution if $\mathcal{Y}(p) = 1$ and a not abnormal execution if $\mathcal{Y}(p) = 0$. In the context of the diagnosis framework introduced in Section 3.2.1, a hypothesis is a health assignment and the set of all possible hypotheses, denoted $\mathcal{X}$, is the set of all possible health assignment. A special case is the *no fault hypothesis* x_0, which assigns not abnormal to all AB-literals, also referred to as not abnormal health-assignment $\neg AB^*$.

Probabilistic diagnosis reasons about the likelihood of hypotheses and assigns a likelihood probability $Pr(X)$, called diagnosis belief, to each hypothesis X. The probability distribution over all possible hypotheses $\mathcal{X}$ defines the current diagnosis beliefs $Pr(X)$. The diagnosis beliefs at time t are denoted $Pr_t(X)$. The introduced framework updates the diagnosis beliefs using past observations and Bayes' rule to obtain a posterior distribution over the unknown diagnosis state X given plan p and the corresponding execution observation $\mathcal{Y}(p)$:

$$Pr_t(X|p, \mathcal{Y}(p)) = \alpha Pr_t(\mathcal{Y}(p)|X, p)Pr_t(X)$$

Chapters 4 and 5 present the overall probabilistic diagnosis framework in more detail. In the context of a planning systems, the introduced probabilistic framework is used to infer the current failure probabilities of actions. In diagnosis the information content of gathered observations is crucial to effective diagnosis, which motivates the next section.

3.5 Explicit Diagnosis for Planning Agents

Explicit diagnosis is an active diagnosis paradigm with the ability to influence the action selection process. More specifically explicit diagnosis optimizes the action selection process strictly for information gain. Actions are selected based on how much diagnostic information they contribute independently of any operational goal. As a result the course of action is optimized for information gain leading to more informative observations than passive diagnosis. This imposes two challenges on the action selection process:

- **Information criterion :** Which criterion determines how informative an action is?
- **Selecting informative plans :** How to select an informative course of action?

The first challenge is to define an information criterion. Given an information criterion, the second challenge is to generate plans with maximum information gain.

3.5.1 Information Criterion for Informative Planning

A plan execution is said to be *informative* if it contributes to clarifying ambiguities in the diagnosis. However, not all plan executions are equally informative regarding diagnosis. The information content of a plan can be measured using mutual information. Mutual information

is a metric from information theory which is commonly used for characterizing the performance of classification and data compression (Cover & Thomas 1991).

Remember, X is the underlying diagnosis state and $\mathcal{Y}(p)$ is the observation resulting from executing plan p. The mutual information between X and $\mathcal{Y}(p)$ is defined as (Cover & Thomas 1991):

$$I(X;\mathcal{Y}(p)) \stackrel{def}{=} \Sigma_{x\in X}\Sigma_{y\in\mathcal{Y}(p)}Pr(x,y)\left[\log_2\frac{Pr(x,y)}{Pr(x)Pr(y)}\right]. \tag{3.22}$$

Conceptually, it measures the bits of information observation $\mathcal{Y}(p)$ tells about the underlying diagnosis state X. It is non-negative, and is equal to zero if and only if X and $\mathcal{Y}(p)$ are independent, in which case, measuring any value of $\mathcal{Y}(p)$ has no implication on refining the underlying diagnosis state X, hence has zero information content. In practice an irrelevant observation should be avoided, but rather an observation should be made that reveals as much information as possible regarding underlying diagnosis state X.

In diagnosis for planning, observations are made from plan executions. The goal for plan selection is to find a plan p such that $I(X;\mathcal{Y}(p))$ is maximized. The first task is to be able to predict the failure probability of a plan p. Let $\mathcal{A}_p$ be the set of unique actions in plan p:

$$\mathcal{A}_p = \bigcup_{a_i\in p}\{a_i\}. \tag{3.23}$$

Given the assumption, that a plan execution behaves abnormal $AB(p)$ if at least one action in the plan behaves abnormal, the following equation holds:

$$AB(p) \;\Leftrightarrow\; AB(a_1)\vee\cdots\vee AB(a_n) \tag{3.24}$$

The predicted probability of a plan being abnormal is a function of the probabilities assigned to all relevant hypotheses. The set of hypotheses that bear on the uncertainty of the outcome of plan p is denoted $\mathcal{X}_p$ and is defined as:

$$\mathcal{X}_p = \{x | a\in x,\; a\in\mathcal{A}_p,\; x\in\mathcal{X}\}. \tag{3.25}$$

Therefore plan p fails whenever any hypothesis in $\mathcal{X}_p$ is true.

$$AB(p) \;\Leftrightarrow\; x_1\vee x_2\vee\cdots\vee x_m \quad \text{where } x_j\in\mathcal{X}_p \tag{3.26}$$

The probability of a plan failure $Pr_t(AB(p))$ is defined as the sum of all probabilities of hypotheses in which plan p fails:

$$Pr_t(AB(p)) \stackrel{def}{=} \sum_{x\in\mathcal{X}_p} Pr_t(x) \tag{3.27}$$

Given the failure probability of a plan, it is straightforward to compare plans in terms of their information content. For example, given two plans p_1 and p_2, p_1 is more informative and preferable to plan p_2 if:

$$I(X;\mathcal{Y}(p_1)) > I(X;\mathcal{Y}(p_2)). \tag{3.28}$$

Mutual information $I(X;\mathcal{Y}(p))$ measures the reduction of uncertainty in the underlying diagnosis state X when observing $\mathcal{Y}(p)$, i.e.,

$$I(X;\mathcal{Y}(p)) = H(X) - H(X|\mathcal{Y}(p)), \tag{3.29}$$

where $H(X)$ is the entropy of X, and $H(X|\mathcal{Y}(p))$ is the entropy of X conditioned on observing $\mathcal{Y}(p)$, i.e., the "remaining uncertainty" after the observation. Maximizing $I(X;\mathcal{Y}(p))$ is equivalent to minimizing $H(X|\mathcal{Y}(p))$. This is equivalent to selecting the plan, which leaves as little uncertainty as possible. Mutual information can be calculated by taking advantage of its symmetry, formally

$$I(X;\mathcal{Y}(p)) = I(\mathcal{Y}(p);X). \tag{3.30}$$

The amount of information that $\mathcal{Y}(p)$ tells about X is equal to the amount that X tells about $\mathcal{Y}(p)$. Exchanging X and $\mathcal{Y}(p)$ in (3.29), results in:

$$I(X;\mathcal{Y}(p)) = H(\mathcal{Y}(p)) - H(\mathcal{Y}(p)|X). \tag{3.31}$$

Although (3.29) and (3.31) are equivalent, the latter is often easier to compute. In a diagnostic problem there is a current diagnosis $Pr(\mathbf{x})$ and an observation likelihood $Pr(\mathcal{Y}(p)|\mathbf{x})$, hence the second term $H(\mathcal{Y}(p)|X)$ is easy to compute. The other way, $H(X|\mathcal{Y}(p))$ in (3.29), involves the posterior belief $Pr(\mathbf{x}|\mathcal{Y}(p))$, which is much harder to compute. For the rest of this discussion (3.31) is used to evaluate the information content of a plan.

Mutual information has been used as an evaluation and selection criterion in a number of applications. For example, (Liu, Reich & Zhao 2003) uses mutual information to decide which sensors to activate in the context of tracking a moving target. Similarly, (Hoffmann, Waslander & Tomlin 2006) uses mutual information to control a fleet of robots, sending robots to most advantageous locations. In the context of this work, the framework is extended to enable the integration of planning and diagnosis.

The question is how to diagnose a system when a fault has been observed? A common divide-and-conquer scheme is to devise a plan p, which includes only half of the actions. If a fault is observed by executing plan p, that means p contains the fault, and the other half that p excludes is cleared of suspicion. If the plan is successful, then p is cleared, and the fault must be in the other half. In this way, every plan dissects the diagnosis space by half.

The following section focuses on the task of diagnosing a single intermittent fault to illustrate the idea of choosing the most informative plan. A more general formulation can be found in (Liu, de Kleer, Kuhn, Price & Zhou 2008). In the single-fault example, the diagnosis space is linear in the number of actions, i.e., $X = \{1, 2, \ldots, M\}$. Further assume that if action a_i has a fault and contributes to some plan, it will exhibit aberrant behavior with some probability q_{a_i}, the *intermittency rate*, resulting in plan failure.

The divide-and-conquer scheme can be generalized via the mutual information criterion. Assume that an faulty action a_i can fail a plan execution with an intermittent probability q_{a_i} if

it is included in the plan p, i.e., with the observation likelihood. For single faults,

$$Pr(\mathcal{Y}(p)|X = a_i) = \begin{cases} 0 & \text{if } \mathcal{Y}(p) = 1 \text{ and } a_i \notin p \\ 1 & \text{if } \mathcal{Y}(p) = 0 \text{ and } a_i \notin p \\ q_{a_i} & \text{if } \mathcal{Y}(p) = 1 \text{ and } a_i \in p \\ 1 - q_{a_i} & \text{if } \mathcal{Y}(p) = 0 \text{ and } a_i \in p. \end{cases} \tag{3.32}$$

Define $H_q^{(a_i)}$ as the entropy corresponding to the binomial distribution q_{a_i}, i.e. as in

$$H_q^{(a_i)} \stackrel{def}{=} -[q_{a_i} \log q_{a_i} + (1 - q_{a_i}) \log(1 - q_{a_i})]. \tag{3.33}$$

The mutual information can be evaluated as,

$$I(X; \mathcal{Y}(p)) = [-y_0 \log y_0 - y_1 \log y_1] - \sum_{a_i \in p} Pr(X = a_i) H_q^{(a_i)}, \tag{3.34}$$

where y_0 is the probability of observing a success, and y_1 is the probability of observing a failure. The derivation follows from (3.31). The first term (the bracketed term) is $H(\mathcal{Y}(p))$, and the second term is $H(\mathcal{Y}(p)|X) = \sum_{a_i \in p} Pr(X = a_i) H_q^{(a_i)}$.

An interesting special case is when all faults are persistent, i.e., $q_{a_i} = 1$ for all a_i. In this case, all $H_q^{(a_i)} = 0$, and the second term in (3.34) vanishes. The mutual information is hence only $-y_0 \log y_0 - y_1 \log y_1$, maximized when $y_0 = y_1 = 0.5$. Thus the optimal explicit diagnosis plan $p^*_{\pi_{exp}}$ is the one that touches closest to half the probability mass:

$$p^*_{\pi_{exp}} = \underset{p \in \mathcal{P}_{obs}}{\operatorname{argmin}} \mid Pr_t(AB(p)) - T \mid \tag{3.35}$$

where the target $T = 0.5$, $\mathcal{P}_{obs}$ being the set of all observable plans. This is a generalization of the divide-and-conquer strategy above.

In the intermittent fault case, the second term is non-zero and can be considered as a "correction" term due to the intermittency. When all the actions have the same q_{a_i} value, the mutual information can be further simplified. It can be evaluated as the function of a single variable $w = \sum_{a_i \in p} Pr(X = a_i)$:

$$-[wq \log wq + (1 - wq) \log(1 - wq)] - wH_q \tag{3.36}$$

Given any plan p, the accumulative probability w and the corresponding mutual information value can now be evaluated. Similarly, the most informative plan p can be found by optimizing the search to determine the plan closest to the optimal w value. The optimal w follows through calculus as:

$$w = \frac{1}{q(2^{H_q/q} + 1)} = \frac{1}{\left(\frac{1}{1-q}\right)^{\frac{1-q}{q}} + q} \tag{3.37}$$

When $q \to 0$, w is asymptotically approaching $\frac{1}{e}$. This can be verified from the limit

$$\lim_{q \to 0} \left(\frac{1}{1-q} \right)^{\frac{1}{q}} = e. \tag{3.38}$$

For $q \in (0, 1]$, w takes values from $\frac{1}{e}$ to $\frac{1}{2}$. Thus, in the case of intermittent faults the target value T is determined by the q_{a_i}'s of the actions and lies between $\frac{1}{e}$ to $\frac{1}{2}$, unlike $T = \frac{1}{2}$ for persistent faults. The concept can be generalized to the diagnosis of multiple faults (Liu et al. 2008). Given the information criterion, it remains to be shown how informative plans can be selected.

3.5.2 Selecting Informative Plans

The previous section introduced an information criterion to select informative plans. Therefore, an optimal explicit diagnosis plan $p^*_{\pi_{exp}}$ is the one that touches closest to some target probability mass:

$$p^*_{\pi_{exp}} = \underset{p \in \mathcal{P}_{obs}}{\operatorname{argmin}} \mid Pr_t(AB(p)) - T \mid \tag{3.39}$$

where $\mathcal{P}_{obs}$ is the set of all observable plans and T is the target probability mass. Given a persistent fault scenario $T = 0.5$ and in the case of an intermittent fault scenario T takes values from $\frac{1}{e}$ to $\frac{1}{2}$ depending of the intermittency rate q.

This section considers the selection process of informative plans, given the derived information criterion. It introduces a search algorithm to find informative plans, which can be generalized to the Problem of finding a path which is as close as possible to some target value. This problem is referred to as the *Target-Value Search Problem*, which is discussed in more detail in Chapter 7. The idea is based on an efficient estimation of the value $v = \mid Pr_t(AB(p)) - T \mid$ during the search. To estimate the value of v, dynamic programming is used to incrementally update the estimate of the upper and lower bound values on $Pr_t(AB(p))$, using the action failure probabilities $Pr_t(a)$ provided by the diagnosis reasoner. This incremental process can be illustrated with an example.

Example: Consider the graph in Figure 3.5 which represents legal action sequences or possible plans that can be executed. A plan starts in the initial state *I* and follows the arcs through the graph to reach an observable state, denoted as goal state *G*.

Suppose that a plan is executed that moves the system through the state sequence $[I, A, C, G]$. The sequence is shown as a shaded background on the relevant links in the figure. Assume the plan resulted in an abnormal execution observation. Unknown to the diagnosis engine, the abnormal outcome was caused by action $a_{A,C}$. Assume the fault is persistent. A diagnosis engine would now suspect all of the actions along the plan path, knowing that at least one of them has contributed to the fault. This results in three hypotheses corresponding to the suspected actions: $\{\{a_{I,A}\}, \{a_{A,C}\}, \{a_{C,G}\}\}$. In the absence of additional information, all hypotheses are assigned with equal probability (see Table 3.1).

The graph structure and probability estimates can be used to construct heuristic bounds on the uncertainty that can be contributed to a plan by any plan suffix. The heuristic can be build backwards from the goal state G (right side of figure). Consider action $a_{D,G}$ leading from state

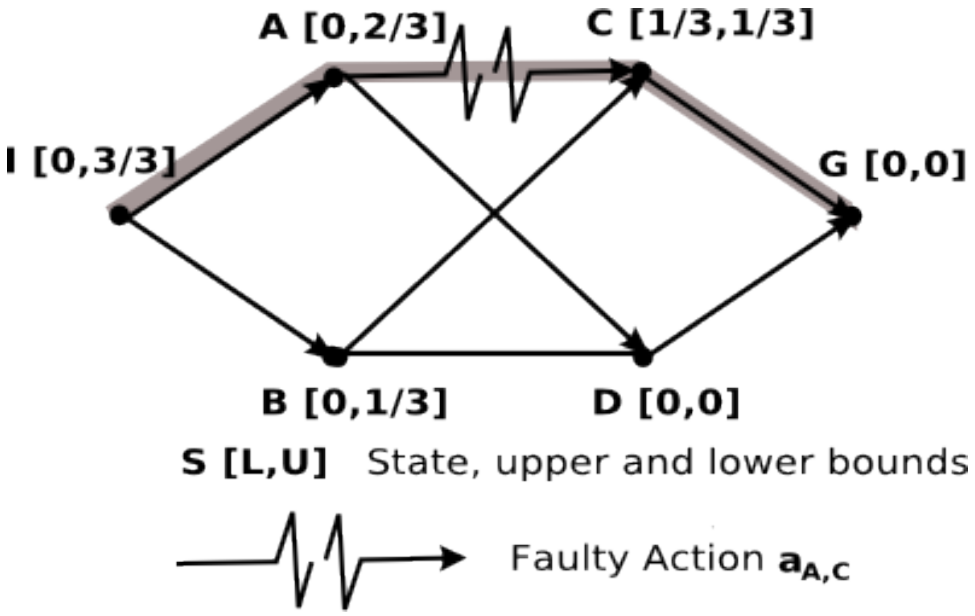


FIGURE 3.5 Machine topology places limits on what can be learned in the suffix of a plan.

Hypothesis	$\{a_{I,A}\}$	$\{a_{A,C}\}$	$\{a_{C,G}\}$
Probability	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$

TABLE 3.1 Probability of Hypotheses (single fault)

D to the goal state G in Figure 3.5. Action $a_{D,G}$ was not part of the plan that was observed to fail, so it is not a candidate hypothesis. Under the single fault hypothesis it has probability zero of being faulted. Extending a prefix plan ending in state D with action $a_{D,G}$ will not increase the failure probability of the extended plan, because the action $a_{D,G}$ has probability zero of being abnormal. There are no other possible plan completions from D to G so both the upper and lower bound for any plan completion from state D is zero.

Similarly, it can be determined that state B also has a lower bound of zero. Since a plan prefix ending in B can be completed by an action $a_{B,D}$ which does not use a suspected action and ends in state D which has lower bound zero. State B has an upper bound of $\frac{1}{3}$ since it can be completed by an unsuspected action $a_{B,C}$ to state C which has both an upper and lower bound of $\frac{1}{3}$ probability of being abnormal.

Once the bounds on the probability of a suffix being abnormal have been built up recursively, these bounds can be used to guide forward search for a plan that achieves the target probability T. Consider a plan starting with action $a_{I,A}$. Action $a_{I,A}$ was part of the plan that was observed to be abnormal. Adding $a_{I,A}$ to a partial plan, increases the probability of failure as it is a candidate itself. After $a_{I,A}$ the system would be in state A. A plan could be completed through D. Action $a_{A,D}$ itself, has zero probability of being abnormal, and given the heuristic bound, it is known that a completion through D must add exactly zero probability of being abnormal. Alternatively, from node A, a plan could also be completed through node C. Action $a_{A,C}$

immediately adds probability of failure $Pr_t(a_{A,C})$ to our plan and given the heuristic bound completion through C is known to increase the probability of being abnormal by exactly $\frac{1}{3}$. The pre-computed heuristic therefore allows predicting abnormality probability for any plan completion. The lower bound of the total plan is $\frac{1}{3}$. This comes from $\frac{1}{3}$ from $a_{I,A}$ plus 0 from the completion $a_{A,D}, a_{D,G}$. The upper bound is $\frac{3}{3}$ equal to the sum of $\frac{1}{3}$ from $a_{I,A}$ plus $\frac{1}{3}$ from $a_{A,C}$ and $\frac{1}{3}$ from $a_{C,G}$. Completing this plan through $[a_{A,C}, a_{C,G}]$ results in a total plan that will fail with probability 1. Note that this plan follows the same path as originally taken. Since it is already known that this plan fails, it adds no new knowledge under the persistent fault assumption. Completing this plan through the suffix $[a_{A,D}, a_{D,G}]$ results in a total plan failure probability of $\frac{1}{3}$, which is closer to $T = 0.5$. If it fails, then $a_{I,A}$ was the failed action. Note, that there is no guarantee that a plan exists for an arbitrary value strictly between the bounds.

Intuitively, the algorithm for computing heuristic bounds is as follows: The bounds are calculated recursively starting from all goal states. A goal state has an empty set of suffix plans $P_{G \to G} = \emptyset$ therefore the lower bound L_G and upper bound U_G are set to zero. Each new state S_m calculates its bounds based on the bounds of all possible successor states $\mathcal{S}_{succ(S_m)}$ and on the failure probability of the connecting action a that causes the transition from S_m to S_n. A successor state S_n of S_m is any state that can be reached in a single step starting from S_m. In the single fault case, if the faults applicable to the suffix and prefix are disjoint $H_{p_{I \to S_n}} \cap H_{a_{S_n,S_m}} = \emptyset$, the failure probability of the action can simply be added to the upper bound U_{S_m} on the failure probability of the suffix. The lower bound L_{S_m} can be updated the same way:

$$\begin{aligned} L_{S_m} &= \min_{a \in \mathcal{A} \wedge app(S_m,a)} (Pr_t(AB(a)) + L_{succ(S_m,a)}) \\ U_{S_m} &= \max_{a \in \mathcal{A} \wedge app(S_m,a)} (Pr_t(AB(a)) + U_{succ(S_m,a)}) \end{aligned} \tag{3.40}$$

For actual implementation, the L_S and U_S values are updated by running a pair of individual uniform cost searches, one for the L_S value and one for the U_S value. Uniform cost search is the form of best-first-search whose only ranking function is the accumulated cost on the best path from initial node to the current node S. For updating the L_S value, the best path is the shortest path found so far, and for updating the U_S value the best path is the longest no-loop path explored so far.

After finding the lower bound L_S and upper bound U_S on the remaining probability mass starting from S, the bounds can be used to define an objective function $f_{\pi_{exp}}$ to search for informative diagnosis plans. The objective function leverages the bounds to estimate $v = |Pr_t(AB(p)) - T|$ by estimating the remaining probability mass starting from S using the following estimate $\hat{v}(p_{I \to S})$:

$$\hat{v}(p_{I\to S}) = \begin{cases} 0 & \text{if } U_S \geq T(p_{I\to S}) \geq L_S \\ L_S - T(p_{I\to S}) & \text{if } L_S > T(p_{I\to S}) \\ T(p_{I\to S}) - U_S & \text{if } U_S < T(p_{I\to S}). \end{cases} \tag{3.41}$$

where $T(p_{I\to S})$ is the adjusted target value for the partial plan $p_{I\to S}$. The information gain objective $f_{\pi_{exp}}$ is then defined using the estimate $\hat{v}$.

$$f_{\pi_{exp}}(p_{I\to S}) = g(p_{I\to S}) + \hat{v}(p_{I\to S}) \tag{3.42}$$

where $T(p_{I\to S})$ is the adjusted target value for the partial plan $p_{I\to S}$, $g(p_{I\to S})$ is the accumulated cost of the plan prefix $p_{I\to S}$, and $\hat{v}(p_{I\to S})$ bounds the remaining probability mass starting from S.

Improving Efficiency through Search Space Pruning

Besides ranking functions, the upper and lower bounds on the total plan failure probability can be used to prune dominated nodes (i.e., nodes that are guaranteed to be less informative than some other visited nodes). Consider a search space with lower and upper bounds that do not straddle the target value T. If the bounds are calculated exactly by dynamic programming, then the best possible plan in this search space will be on one of the two boundaries that lies closer to the target value T. Let L_{S_n} and U_{S_n} be the lower and upper bound of the search space $p_{I\to S_n}$, then let $V_{p_{I\to S_n}}$ represents a guaranteed upper bound on the closeness of a total plan $p_{I\to G}$ to T starting with the partial plan $p_{I\to S_n}$ as prefix:

$$V_{p_{I\to S_n}} = \min(|L_{S_n} - T(p_{I\to S_n})|, |U_{S_n} - T(p_{I\to S_n})|) \tag{3.43}$$

is the closeness of the best plan to T, where $T(p_{I\to S_n})$ is the adjusted target value for the partial plan $p_{I\to S_n}$. A prefix $p_{I\to S_i}$ dominates every plan $p_{I\to S_j}$ if:

$$\bigwedge_{n\in\{i,j\}} \neg(L_{S_n} \leq T \leq U_{S_n}) \ \wedge \ (V_{p_{I\to S_i}} < V_{p_{I\to S_j}}) \ \Rightarrow prune(S_j) \tag{3.44}$$

The proposed diagnosis strategy is a fundamentally new approach to diagnose a planning system. As with many frameworks, it is computationally explosive if directly implemented as described in the definitions. Chapter 7 introduces a set of search algorithms to search for informative plans more efficiently.

3.6 Pervasive Diagnosis for Planning Agents

The previous section introduced explicit diagnosis for planning agents. The optimal explicit diagnosis plan $p^*_{\pi_{exp}}$ maximizes the diagnostic information by touching as close as possible to

some target probability mass:

$$p^*_{\pi_{exp}} = \underset{p \in \mathcal{P}_{obs}}{\operatorname{argmin}} \mid Pr_t(AB(p)) - T \mid \tag{3.45}$$

where $\mathcal{P}_{obs}$ is the set of all observable plans and T is the target probability mass.

This section introduces pervasive diagnosis for planning agents, which is realized by extending explicit diagnosis with an additional constraint that guarantees that plans simultaneously achieve operational goals while gathering information. As a result, the optimal pervasive diagnosis plan $p^*_{\pi_{per}}$ is an operational plan that touches closest to some target probability mass:

$$p^*_{\pi_{per}} = \underset{p \in \mathcal{P}_{I \to G}}{\operatorname{argmin}} \mid Pr_t(AB(p)) - T \mid \tag{3.46}$$

where $\mathcal{P}_{I \to G}$ is the set of all plans at achieve an operational goal and T is the target probability mass. Similar as in explicit diagnosis, T is 0.5 in a persistent fault scenario and a value between $\frac{1}{e}$ and $\frac{1}{2}$ in an intermittent fault scenario. Given this modified objective, the plan selection framework for explicit diagnosis can be directly applied to search for pervasive diagnosis plans.

Over the previous sections, three different diagnosis paradigms were introduced. To compare those strategies, the next section develops a formal framework that evaluates the total responce costs. This theoretical results are then verified in Section 3.9 by experiments presented.

3.7 Choosing an Diagnosis Paradigm

In this section a formal framework is developed to compare the presented diagnosis paradigms. The comparison is based on a simple cost model of expected unrealized production. In this context, production refers to realized operational goals, e.g. printed sheets in the case of a printer. The expected production loss is due to effort of isolating the faulty component (diagnosis costs) and exchanging this component (repair costs). The cost in this model represents the expected total amount of lost production due to the fault, called expected response cost $C^\pi(t)$:

$$C^\pi(t) = c^\pi_d(t) + c_r(\mathrm{Pr}_t) \tag{3.47}$$

where $c^\pi_d(t)$ is the diagnosis costs of policy π as a function of the time t spent on diagnosis and $c_r(\mathrm{Pr}_t)$ is the expected repair cost as a function of the current belief state represented as a probability distribution Pr_t.

Assume that some diagnosis policy is chosen and time t is spent on his diagnosis policy. At the end of time t, the system will have beliefs about the condition of its components Pr_t. While the system could continue diagnosis until Pr_t unambiguously assigns the failure probability to a single component, it may be cheaper to instantly repair all currently suspected components. Diagnosis is still valuable, however, as repairs will typically cost less when Pr_t is more specific about the nature of the fault.

For a given diagnosis policy π, the response cost can be minimized by choosing the diag-

nosis time t that minimizes the response cost:

$$C^{\pi *} = \min_t C^{\pi}(t) \tag{3.48}$$

$$= \min_t [c_d^{\pi}(t) + c_r(\mathrm{Pr}_t)]. \tag{3.49}$$

To better understand the tradeoffs between passive, explicit and pervasive diagnosis, consider a simplified opportunity cost model. Our model assumes that under normal conditions the machine produces output at its nominal rate r_{nom} and that diagnosis efforts begin once some abnormal outcome is observed (i.e., a paper jam, dog ear, etc.). Under the diagnosability assumption, it is assumed that the system failure is immediately detectable and that the only goal is to isolate and repair the problem. For simplicity, a single, intermittent fault is assumed, which causes the system to produce faulty output with probability q_{nom}. Therefore if regular production is continued (without any adaptation to a detected fault) the system might still produce output with a success probability of $1 - q_{nom}$. This can be due to the nature of the fault intermittency or due to the internal multi-way redundancy.

With the passive diagnosis policy the machine continues to produce output without any adaptation to the detected fault. Therefore the cost can be estimated by:

$$c_d^{pass} = t \times r_{nom} - t \times r_{nom} \times (1 - q_{nom}). \tag{3.50}$$

where the successfully produced output $t \times r_{nom} \times (1 - q_{nom})$ is subtracted from the potential lost output $t \times r_{nom}$.

For the explicit approach, assume production is suspended during diagnosis (reasonable as chances of producing the correct output are negligible), thus diagnosis cost is equal to the nominal rate of production that would have occurred:

$$c_d^{exp} = t \times r_{nom}. \tag{3.51}$$

Finally, for pervasive diagnosis it is assumed that the machine produces at a reduced rate $r_{perv} \leq r_{nom}$ during diagnosis. The reduced rate results from maximizing information gain while still obeying the production constraints. Similar to the passive policy, production is faulty during pervasive diagnosis. The probability of producing a faulty output increases due to the effort of increasing the information gain. The most information can be gained by executing plans with maximum uncertainty. That leads to an increased production failure probability $q_{perv} \geq q_{nom}$ during production, thus the cost is,

$$c_d^{perv} = t \times r_{nom} - t \times r_{perv} \times (1 - q_{perv}). \tag{3.52}$$

For all three policies, a simple repair cost model is introduced: The technician receives a list of suspected printer modules in decreasing order of their fault probability. He then follows a very simple procedure: step through the list, exchange the next module and test the machine

until the machine is working properly. Using this model, repair costs can be estimated by

$$c_r(\mathrm{Pr}_t) = c_{exc} \sum_{i=1}^{N} i \times Pr_t(i) \tag{3.53}$$

where i indicates the amount of modules that have to be exchanged with probability $Pr_t(i)$. Each exchange takes is assumed to take a constant amount of time t_{exc} and thus has a constant cost $c_{exc} = r_{nom} \times t_{exc}$.

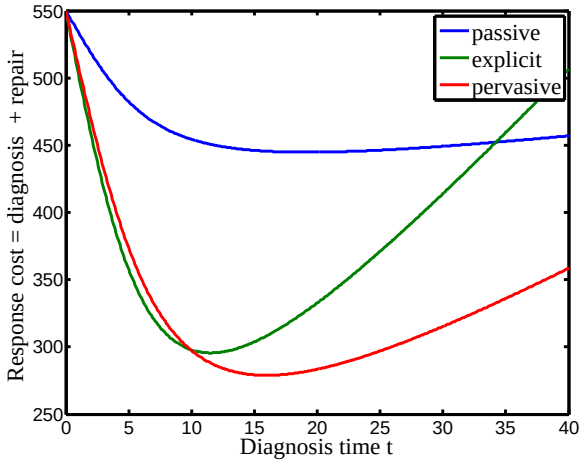
The simplified opportunity cost model is used to compare the different diagnosis policies. To simulate the information gained at any time of a particular diagnosis policy a simple abstraction can be chosen by selecting a family of distributions. Recall, components are ordered in decreasing posterior probability order. Thus, Pr_t can be approximated with a monotonically declining function. The "slope" of this decreasing function is assumed to depend on how much information gain can be achieved per unit time. This can be represented with a "slope" parameter α. Further assume that each policy will reach a point where it can not gain more information. That point might be different among the policies due to the amount of internal redundancy they can exploit. For example, the passive policy optimizes for efficient production and will not explore a particular production plan if there exists a more efficient alternative production plan. The pervasive policy on the other hand explores alternative production plans to gain information. Explicit diagnosis usually gains more information, since it does not have to obey the production constraint. The residual uncertainty is captured by the parameter β^{π}. Imagine that the distribution Pr_t is given by a family of the form:

$$\mathrm{Pr}_t^{\pi} = \gamma(e^{-\alpha t} + \beta^{\pi}) \tag{3.54}$$

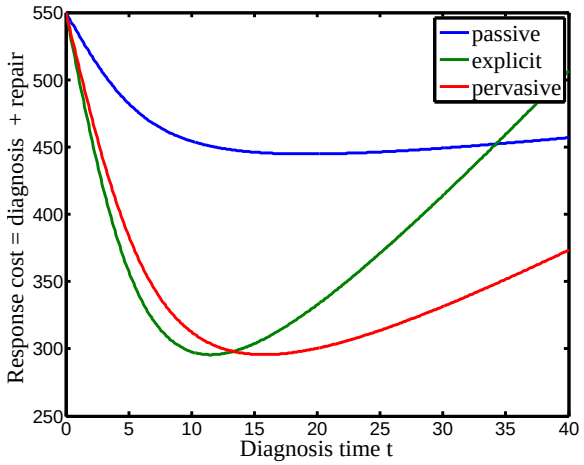
where γ is a normalizing constant to ensure that the elements of Pr_t sum to 1. Of course, this distribution does not reflect a real system, but it will allow us to see how different diagnosis methods respond given their residual uncertainty β^{π}.

The response cost model allows us to understand how the cost varies with the diagnosis policy chosen and with the amount of time spent in diagnosis. The response cost over time is plotted for passive, explicit, and pervasive diagnosis in Figure 3.6. Subfigure 3.6(a), 3.6(b) and 3.6(c) show each of the three policies as a separate curve. For short diagnosis times repairing of the machine is quickly started, but there exists only poor information about the underlying fault. This leads to high repair cost. If additional time is spent in diagnosis, potential time for production is lost, but information is gained, which reduces repair cost. At some point, cost for diagnosis outweighs the reduction in repair cost, and any additional time spent for diagnosis increases the response cost. Therefore each policy results in a U-shaped curve where the bottom of the U indicates the optimal amount of time spent for diagnosis using a particular policy.

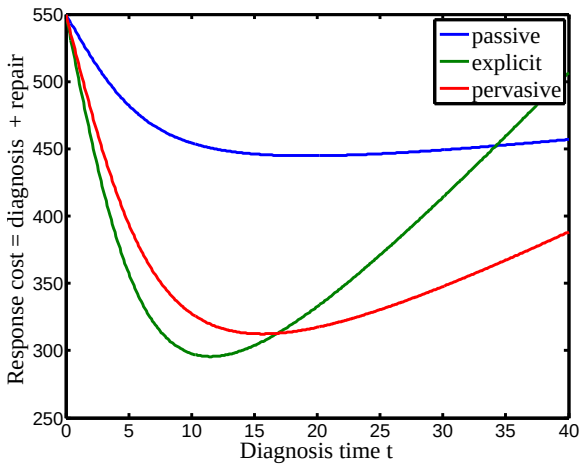
Once the optimal amount of time for each policy is determined, one must still select which of the policies to employ. The effectiveness of pervasive diagnosis depends on whether the space of production plans is sufficiently informative to isolate the fault. Figure 3.6(a), 3.6(b) and 3.6(c), show that changing the residual uncertainty β_{perv} for pervasive diagnosis from 0.1 to 0.15 to 0.2 causes pervasive diagnosis to change from the diagnosis method with lowest



(a) β_{perv}=0.1 Low uncertainty, pervasive diagnosis achieves lowest cost



(b) β_{perv}=0.15 Medium uncertainty, pervasive diagnosis ties with explicit



(c) β_{perv}=0.2 High uncertainty, explicit diagnosis achieves lowest cost

FIGURE 3.6 Responce costs as a function of time, plotted with β_{perv}=0.1, β_{perv}=0.15, and β_{perv}=0.2. Pervasive diagnosis is favorable when operational plans exist, which are informative.

cost, to the method with costs equal to explicit diagnosis to being more expensive than explicit diagnosis.

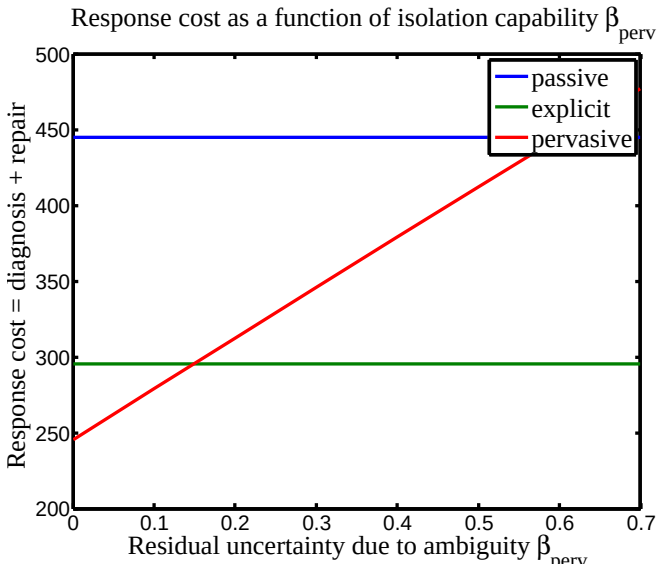


FIGURE 3.7 Lowest minimal response cost as a function of β_{perv}.

Subfigure 3.7 plots the optimal cost for pervasive diagnosis as a function of residual uncertainty β_{perv} against the constant costs for explicit and passive diagnosis. There is a distinct region in the lower left corner where the low cost of pervasive diagnosis dominates explicit and passive approaches. Systems in this region can use pervasive diagnosis to lower diagnosis costs without reengineering the system or adding sensors.

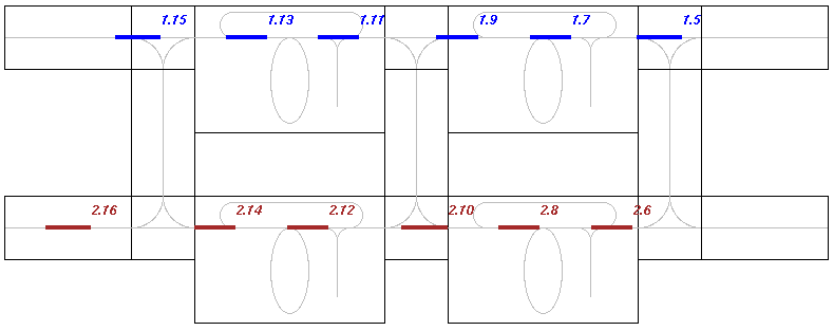
3.8 Integration of Pervasive Diagnosis and Regular Operation

This section outlines how pervasive diagnosis can be integrated into a planning agent such that the resulting Self-diagnosing Agent is able to optimize simultaneously for the following two objectives:

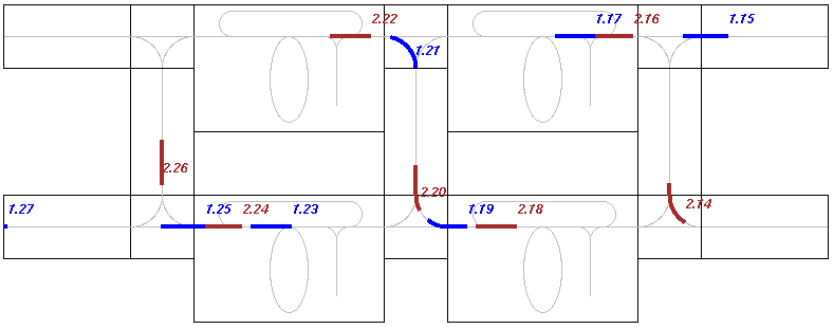
- $f_{\pi_{op}}$: maximizing operational performance (e.g. minimizing plan finishing time), and
- $f_{\pi_{per}}$: maximizing information gain information (e.g. minimizing ambiguity).

In general, the integration of two objective functions is not trivial as improving the final plan quality along one dimension can worsen the quality according to the other. Therefore a trade off parameter, denoted γ, is introduced, which specifies the "weights" for each objective function and the integration is realized by a weighted sum. As a result, the top level objective function f of a Self-diagnosing Agent defined by Equation 3.55.

$$f = \gamma f_{\pi_{per}} + (1 - \gamma) f_{\pi_{op}} \tag{3.55}$$



(a) The printer consists of 4 print engines (large rectangles) connected by controllable paper handling modules. Sheets enter on the left and exit on the right. There are 48 actions controlling feeders, paper paths, print engines and finisher trays.



(b) The flexibility of the architecture can be exploited to choose paper paths that use different subsets of components. A sequence of these paths can be used to isolate the fault.

FIGURE 3.8 A schematic of a modular printer used in the experiments.

where the value γ is a trade off parameter. The weighted objective function f (Equation 3.55) combines the information gain objective (e.g. minimizing ambiguity) (denoted $f_{\pi_{per}}$) with the performance objective (e.g. minimizing plan finishing time) (denoted $f_{\pi_{op}}$) to guide the search towards the plan, which is optimal with respect to the trade off between information gain and performance.

The next section outlines experimental results that suggest that the theoretical improvement can be realized in the context of a manufacturing system, in particular on a hyper-modular, multi-engine printer.

3.9 Experiments

This section evaluates the practical benefits of a Self-diagnosis Agent in the context of a manufacturing system, in particular on a hyper-modular, multi-engine printer. Pervasive Diagnosis has been implemented and combined with an existing model-based planner and diagnosis engine. The overall system has been tested on a model of a modular printer (Ruml, Do & Fromherz 2005) or (Do & Ruml 2006*b*). Multiple pathways allow the system to parallelize

production, to use specialized print engines for specific sheets, and to reroute around failed modules. A schematic diagram showing the available paper paths in the machine is presented in Figure 3.8(a).

A test run is done for each possible abnormal action. The planner then receives a print job request from the queue. It generates a plan and sends it to a simulation of the printer. The simulation models the physical dynamics of the paper moving through the system. Plans that execute on this simulation can be executed unmodified on our physical prototype machine in the laboratory. The simulation determines the outcome of the job. If the job is completed without failure, e.g. a dog ear (bent corners) or a scuff mark and deposited in the requested output tray, the plan has succeeded or in the language of diagnosis, the plan was not abnormal, otherwise the plan was abnormal.

The original plan and the outcome of executing the plan are sent to the diagnosis engine. The engine updates the fault hypothesis probabilities. As soon as a fault occurs, the planner searches for the most informative plan. Since there is a delay between submitting a plan and receiving the outcome, production jobs from the job queue are planed without optimizing for information gain until the outcome is returned. This keeps the performance of the system high.

The performance of passive diagnosis (only regular operation), explicit diagnosis (alternates between explicit diagnosis and regular operation) and pervasive diagnosis (regular operation modified to obtain additional diagnostic information) are evaluated.

In the experiments an exchange time for a single module is set to be $t_{exc} = 150$ sec. The nominal rate of the system is $r_{nom} = 3.1$ sheets/sec. The experiments have shown that the reduced rate of pervasive diagnosis is $r_{perv} = 1.9$ sheets/sec.

Based on the introduced model, the performance of passive diagnosis, explicit diagnosis and pervasive diagnosis is compared for three levels of fault intermittency, represented by the probability q. When $q = 1$, a faulty action always causes the plan to fail. When $q = 0.01$ a faulty action only causes the plan to fail with a statistical mean of 0.01.

The summary of the experimental results is in Table 3.2. A more detailed visualization of the results is presented in Figures 3.9, 3.10,and 3.11 for the intermittency rates $q = 0.01$, $q = 0.1$, and $q = 1$. Given the probabilistic intermittency rates, the experiments are averaged over 1000 runs to reduce statistical variation.

Figure 3.9, 3.10, and 3.11 show the expected costs of repair relative to repair start time in terms of lost production in relation to a healthy machine. The figures plot the costs for the following fault intermittency rates: $q = 0.01$, $q = 0.1$, and $q = 1$. Cost of repair is computed by estimating the repair time based on the current probability distribution over the fault hypothesis (see Section 3.7) and pricing this downtime according to the nominal machine production rate. Cost of diagnosis at time t is the accumulated production deficit in relation to a healthy machine producing at its nominal rate r_{nom} (see Section 3.7). The x-axis is the amount of time (relative to the first occurrence of the fault) after which one chooses to stop diagnosis and start repairing the machine. The minimum of the sum of these costs denotes the optimal point in time to switch from diagnosis to repair and gives the minimal expected total loss of production due to the fault.

In all experiments the optimal response costs of pervasive diagnosis was below those of the other two approaches. As expected, the respective optimal durations of diagnostic processes are in the order (shortest to longest) of explicit, pervasive and passive. This corresponds to the

q=0.01	min. response cost	time t	# of exchanged modules
Passive	1010.22	214.1	2.01
Explicit	947.25	76.6	1.41
Pervasive	768.80	204.6	1.01
q=0.1	min. response cost	time t	# of exchanged modules
Passive	1055.25	35.0	2.10
Explicit	591.31	29.1	1
Pervasive	547.77	37.2	1
q=1.0	min. response cost	time t	# of exchanged modules
Passive	1012.00	7.78	2.01
Explicit	515.43	3.1	1
Pervasive	509.76	3.78	1

TABLE 3.2 Pervasive diagnosis has the lowest rate of lost production.

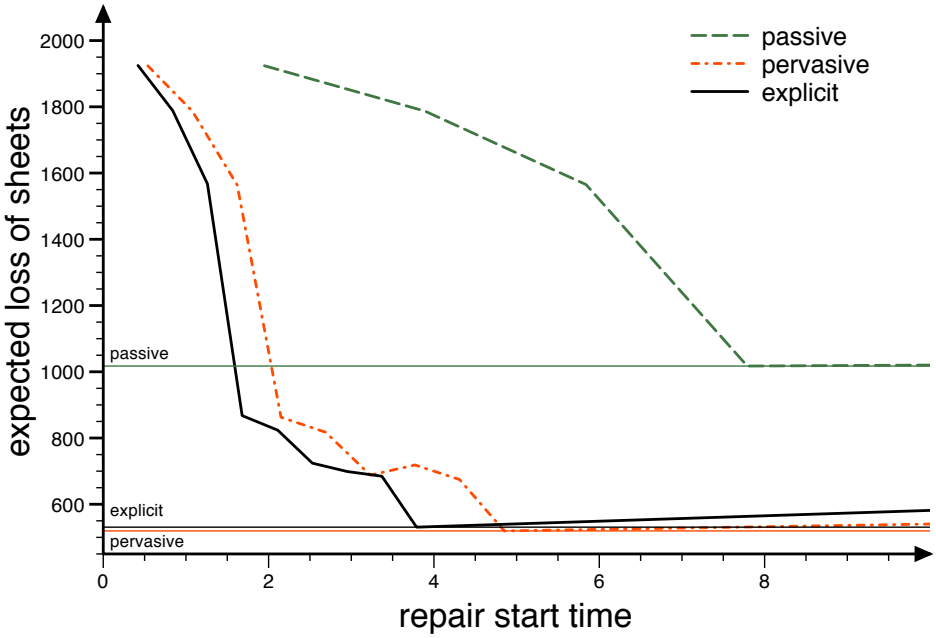


FIGURE 3.9 Experimental results for intermittency rate $q = 1.00$. The curves show the responds cost in expected loss of sheets as a function of diagnosis time t. For each diagnosis policy there is a horizontal line indicating the minimal response cost.

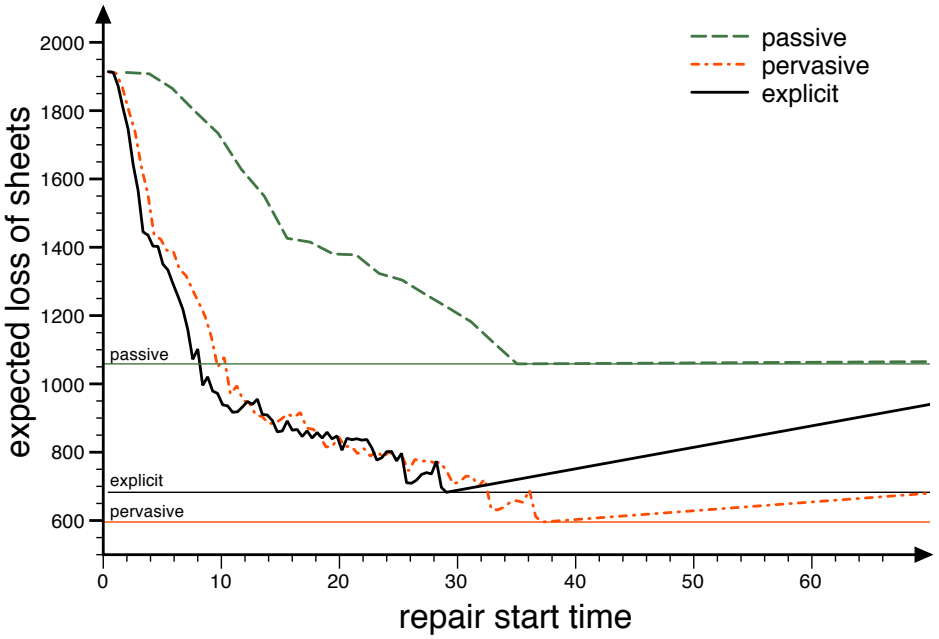


FIGURE 3.10 Experimental results for intermittency rate $q = 0.10$. The curves show the responds cost in expected loss of sheets as a function of diagnosis time t. For each diagnosis policy there is a horizontal line indicating the minimal response cost.

fact that explicit diagnosis focuses solely on the diagnosis task. Therefore explicit diagnosis is able to select plans with maximal diagnosis information gain and can isolate the faulty component in the shortest amount of time. However, due to the high production loss (production is halted), explicit diagnosis does not result in minimal response costs. Passive diagnosis has the lowest rate of lost production, but incurs the highest expected repair costs due to its lower quality diagnosis. This corresponds to the fact that the plans executed during passive diagnosis are optimized for production regardless of diagnosis needs. Pervasive diagnosis intelligently integrates diagnosis goals into production plans by using planning flexibility. Passive diagnosis works well for faults with low intermittency rates and explicit diagnosis for high intermittency, pervasive diagnosis combines the benefits of both. This leads to a lower total expected production loss in comparison to passive and explicit diagnosis.

3.10 Conclusions

This chapter introduced a new architecture, coined a *Self-diagnosing Agent*, which realizes the integration of acitve diagnosis and regular operation by a novel diagnosis paradigm called *pervasive diagnosis*. Pervasive diagnosis actively manipulates the course of action during operation in order to gain diagnostic information without suspending operation. Consider a system where operational goals can be achieved in multiple ways. This flexibility is exploited to generate operational plans that simultaneously gather information by trading off information

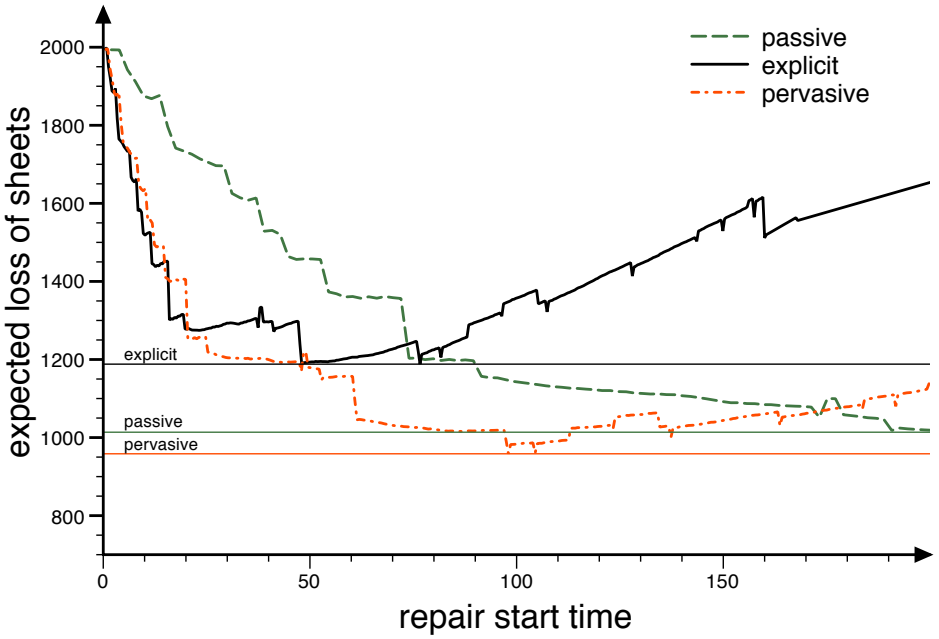


FIGURE 3.11 Experimental results for intermittency rate $q = 0.01$. The curves show the responds cost in expected loss of sheets as a function of diagnosis time t. For each diagnosis policy there is a horizontal line indicating the minimal response cost.

gain with performance objectives. Therefore active diagnosis and regular operation occur at the same time leading to higher long-run performance than an integration of regular operation with passive diagnosis or alternating between explicit diagnosis and regular operation. While a Self-diagnosing Agent has interesting theoretical advantages, the experiments have shown that the theoretical benefits can be realized on a real time applications. Throughout this chapter an application of a Self-diagnosing Agent to a hyper-modular, multi-engine printer has been presented. The techniques generalize to a wide class of domains ranging from space crafts to robotics to smart networks.

CHAPTER 4

Tiered-Partitioned Inference for Multiple Fault Diagnosis

Diagnosing multiple-component systems is difficult and computationally expensive, as the number of fault hypotheses grows exponentially with the number of components in the system. This chapter describes an efficient computational framework for statistical diagnosis featuring two main ideas:

- **Tiered Inference:** Structuring fault hypotheses into tiers, starting from low cardinality fault assumptions (e.g., single fault) and gradually escalating to higher cardinality (e.g., double faults, triple faults) when necessary.
- **Partitioned Inference:** Dynamically partitioning the overall system into subsystems, within which there is likely to be a single fault. The partition is based on correlation between the system components and is dynamic: when a particular partition is ruled out, a new one is constructed based on the updated belief. When no viable partition remains, the search proceeds to the next tier.

This approach enables the use of single-fault diagnosis, which has only linear complexity, to the subsystems avoiding exponential hypothesis explosion. The performance is analyzed and shows that for practical systems where most components are functioning properly, the proposed scheme achieves a desirable tradeoff between computational cost and diagnosis accuracy.

4.1 Introduction

Troubleshooting a system to isolate faulted components can be difficult. This is especially true, when diagnosing multiple faults as the number of fault combinations grows exponentially in the number of components. In diagnosis literature, various ideas have been proposed to address the computational challenge. The general diagnosis engine (GDE) work (de Kleer & Williams 1987) finds minimal diagnoses, isolating not the complete fault combination, but a minimal subset of faulted components that can explain the observations. Another example is the production plant diagnosis work (Kuhn & de Kleer 2008), which extends model-based diagnosis (Reiter 1987, de Kleer & Williams 1987) to production systems such as food processing plants, oil refineries, and printers. The diagnosis engine (Kuhn & de Kleer 2008)

discriminates fault assumptions based on their complexity. Diagnosis starts with simple fault assumptions (e.g., single, persistent, and/or independent faults) for computationally efficient diagnosis, and escalates to more complicated fault assumptions (e.g., multiple, intermittent, and/or interaction faults) when necessary. This progression of diagnosis greatly reduces computation complexity.

The minimal diagnosis idea and the progressive diagnosis work are qualitative in nature. This chapter extends this approach from qualitative reasoning to statistical inference. The main challenge is to perform Bayesian updates in the context of multiple fault diagnosis computationally efficient.

Statistical inference is widely adopted in diagnosis. The basic idea is to evaluate hypotheses (fault combinations) based on their probability given the observation data (Berger 1995). For illustration, the standard notation is used with uppercase symbols denoting random variables and lowercase symbols denoting a particular realization. Mathematically, the probability for any hypothesis X in the hypotheses space $\mathcal{X}$ can be updated via the Bayes rule:

$$Pr(X|Y) = \alpha Pr(Y|X)Pr(X), \tag{4.1}$$

where $Pr(X)$ is the initial probability (prior) for the hypothesis X, $Pr(Y|X)$ is the likelihood probability of observing Y given that X is true, and α is the normalization factor such that $\sum X \in \mathcal{X} Pr(X|Y) = 1$. The resulting $Pr(X|Y)$ is the posterior probability that X is true given the observation Y. The diagnosis that best explains the data is the maximum a posterior (MAP) estimate

$$X_{MAP} = \arg\max_{X \in \mathcal{X}} Pr(X|Y). \tag{4.2}$$

While Bayesian update offers a coherent and quantitative way of incorporating observation data, it faces the same need to search through all hypotheses in $\mathcal{X}$. In practice, a system with k components has the hypothesis space

$$X = \{000000, 000001, \ldots, 111111\} \tag{4.3}$$

Each hypothesis $X \in \mathcal{X}$ is a bit vector, where i-th bit is an indicator whether the i-th component has fault. (0 for not having fault, 1 for having fault). The computational complexity of the Bayesian update is $O(2^k)$. When k is large, the update is prohibitively expensive.

The proposed approach introduces two ideas to mitigate the computational difficulty. The first is tiered inference, illustrated in Section 4.2. The basic idea is to organize the hypothesis space $\mathcal{X}$ into tiers with increasing fault cardinality. Inference is restricted to lower tiers (fewer defective modules) until the lower tiers have been ruled out by the observation data. This idea is implicit in many diagnosis engines such as GDE (de Kleer & Williams 1987) and MBD (Thiebuax, Cordier, Jehl & Krivine 1996), but here it is develop as part of a more general framework. The main contribution is the second idea: a divide-and-conquer strategy presented in Section 4.3. It partitions system components into single-fault subsystems. This partitioning enables utilizing single-fault diagnosis, which only has linear complexity, to diagnosing a multiple-fault system.

Figure 4.1 illustrates these basic concepts. In the diagram, the hypothesis space $\mathcal{X}$ is repre-

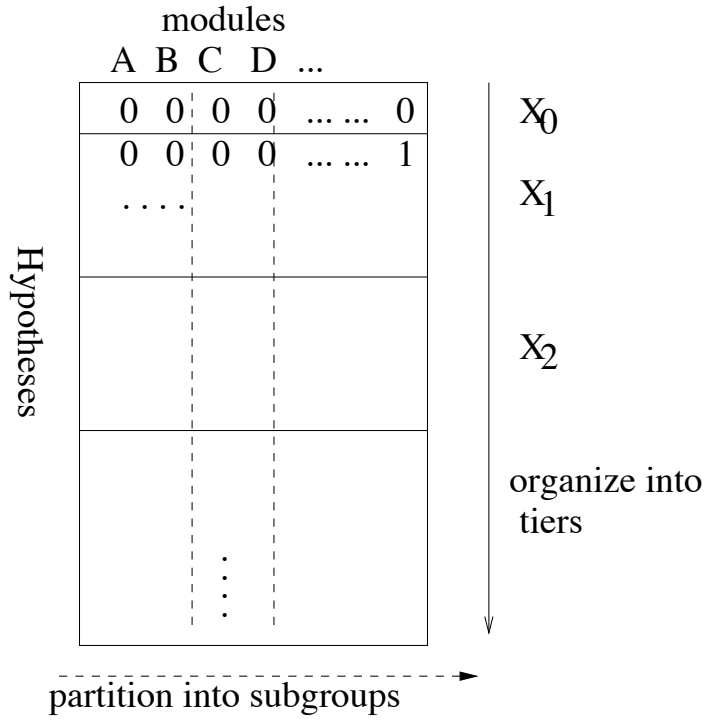


FIGURE 4.1 Basic idea: (1) organize hypothesis into tiers (along the vertical direction), and (2) partition components into subgroups (along the horizontal direction), for instance, AB are a group, and CD are a group.

sented as a matrix, with columns representing components, and rows representing the different fault assumptions. Organizing hypotheses into tiers is shown as dividing the hypothesis into vertically stacked blocks. Inference starts from the top block (no-fault tier), and progresses down to the single-fault tier $\mathcal{X}_1$, then to the double-fault tier $\mathcal{X}_2$, and so on. The second idea is to organize modules into groups, for instance, AB form a subsystem, and CD forms a subsystem. This forms a horizontal partition in the figure.

While partitioning a multiple-fault system into single-fault subsystems is a neat idea, how to partition is actually a tricky problem. Consider the following best-effort approach: given the posterior belief $\{Pr(X)\}$, a partition is preferred, which results in subsystems that are single-fault with maximum probability. Section 4.4 describes a computationally efficient greedy algorithm based on the intuition that modules within a subsystem must be negatively correlated so that the total number of faults remains constant (single-fault). The partitioning idea and algorithm are the main novelty of this work.

Many diagnosis approaches have taken advantage of the hierarchical structure of the system being diagnosed (Pravan 2001) (Srinivas 1994). For example, all possible combinations of faults in a subsystem can be represented as a single component, as done in (Siddiqi & Huang 2007). Similarly, if two distinct faults are indistinguishable they can be represented as one fault. These approaches greatly reduce computational cost. However, they depend on a single decomposition determined a priori. The approach presented in this chapter is quite different: it dynamically constructs and modifies the decomposition as diagnosis proceeds and is complementary to these fixed approaches.

Section 4.5 demonstrates the application of tiered-partitioned inference to manufacturing plant diagnosis. The presented examples are drawn from hyper-modular, multi-engine printers (Section 2.4.1). A reprographic system receives a continuous stream of print jobs and each print job consists of a sequence of sheets of paper. The planner constructs an optimal plan for each sheet which specifies a full trajectory through potentially dozens of modules. Each module type has a set of actions it can perform. One of those actions may be faulty, but the module may always succeed at other actions. Therefore, the proposed approach applies the framework to actions, not to modules. Each capability fails approximately independently. Figure 4.2 illustrates a three way module with six capabilities. Figure 4.3 illustrates five modules of the two types connected together. Circles indicate rollers, triangles indicate sensors, and two sheets of paper are indicated in red. Note that three modules can be acting on the same sheet of paper at one time.

It is possible to design machine configurations where a failure in the output capability of one module cannot be distinguished from a failure in the input capability of the connected module. In the presented framework, this will show up as a double fault when in fact only one of the two modules if faulted. This confusion can be avoided by applying an idea from digital circuits to collapse indistinguishable faults. In addition, multiple faults are allowed. Experiments with printers have shown that most equipment contains multiple, low frequency, intermittent faults. High-end reprographic systems operate more-or-less continuously providing a constant stream of observations and exceptions. Consider a system, where raw material (e.g. sheets of paper) is transported through a sequence of modules by executing a sequence of actions (known as a "plan") and modified to produce a product (e.g. printed paper). A plan execution results in an execution observation, a not abnormal execution or an abnormal exe-

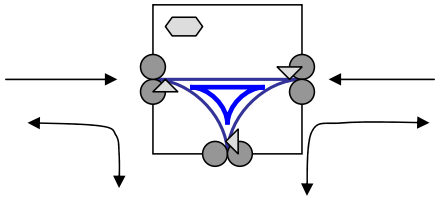


FIGURE 4.2 A more detailed figure of a three way module. The 6 possible paper movements (capabilities) are indicated on the diagram.

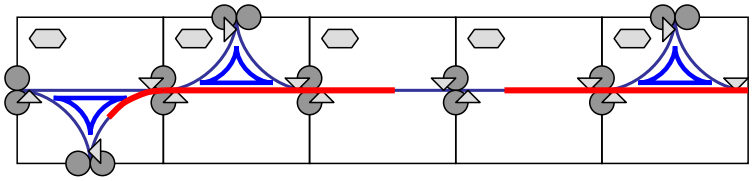


FIGURE 4.3 A more detailed figure of five connected modules moving two sheets of paper.

cution. The execution is abnormal if any of the actions in the plan malfunctions. Furthermore assume that an execution failure caused by a defective action cannot be repaired by subsequent actions. In this paradigm, diagnosis aims at isolating abnormal actions based on the executed plans and their corresponding execution observations. In this context, the tradeoff between computational cost and inference accuracy is analyzed for this diagnosis problem. While production plant diagnosis is used as an illustration, the presented ideas are more general and can be extended to other diagnosis problems.

4.2 Tiered-partitioned inference

To mitigate the computational difficulty, our prior work in (Kuhn & de Kleer 2008) has been advanced and the notion of *tiered-partitioned inference* is now proposed. The basic idea is to restrict posterior computation to a subset of hypotheses, and broaden the scope of inference only when necessary. In the tiered-partitioned inference framework, the overall hypothesis space is partitioned into tiers, i.e.,

$$\mathcal{X} = \mathcal{X}_0 \cup \mathcal{X}_1 \cup \mathcal{X}_2 \cup \cdots \cup \mathcal{X}_k, \tag{4.4}$$

where each tier $\mathcal{X}_j$ is defined as the collection of hypotheses assuming a total of j faults in the system, i.e., hypotheses with cardinality j ($\sum_i x_i = j$). Once the system is observed to be malfunctioning, the need for diagnosis arises. Inference starts with the single-fault tier

$\mathcal{X}_1$, assuming that the system has only one fault. At this tier, the inference only updates the posterior for the hypotheses in $\mathcal{X}_1$ and ignores all other hypotheses. This drastically reduces the computational complexity from $O(2^k)$ to $O(k)$. However, the single-fault assumption is an approximation, as the system can have multiple faults. When a conflict is detected, i.e., all the hypotheses in $\mathcal{X}_1$ conflict with the observation data. As a consequence the inference is escalated to the next tier $\mathcal{X}_2$, assuming a total of two faults in the system. The inference then updates all hypotheses in $\mathcal{X}_2$. The process repeats until observation data or the hypothesis space is exhausted.

Before diving into technical details, an example is used to provide some intuitions. Figure 4.4 shows the computation structure in the tiered-partitioned inference framework. The hypothesis space $\mathcal{X}$ is partitioned into non-overlapping tiers $\mathcal{X}_1, \mathcal{X}_2, \ldots, \mathcal{X}_k$ as shown in Figure 4.4a. Figure 4.4b shows the computation in the tiered-partitioned inference algorithm. Imagine a sequence of observations as follows:

1. The first batch of observations is used to update all hypotheses in $\mathcal{X}_1$, hence the computation is linear in $|\mathcal{X}_1|$. In Figure 4.4b, this is shown as vertical solid lines in the first tier (the upper-left corner). The length of the lines symbolizes the amount of computation, in this case proportional to the size of $\mathcal{X}_1$.

2. The last observation of the first batch rules out all hypotheses in $\mathcal{X}_1$. In this case, an escalation is forced to the double tier $\mathcal{X}_2$. The observations now need to be re-applied. This corresponds to the solid lines in the second tier. The computation is linear in $|\mathcal{X}_2|$.

3. The second batch of observations are applied to all hypotheses in $\mathcal{X}_2$. The computation is shown as the dashed lines in the second tier.

4. The last observation of the second batch further rules out all hypotheses in $\mathcal{X}_2$. That forces an escalation to $\mathcal{X}_3$ and all the previous observations have to be re-apply (solid and dashed lines in the third tier). As more observations are accumulated, the update computation (dotted lines in the figure) is restricted to $\mathcal{X}_3$.

In contrast, Figure 4.4c shows the computation where all observations are applied to all hypotheses. Notice that the total vertical lines are much shorter in Figure 4.4b than in Figure 4.4c. The computational savings are clear. The savings are primarily due to the fact that the higher tier hypotheses are not updated until necessary.

In this tiered-partitioned inference framework, what is the price to pay in return for the inference computational savings? First bear in mind that this is an approximation — the higher tiers have been ignored when the lower tiers remain consistent with the observations. Therefore tiered-partitioned inference loses optimality, for instance, the maximum a posterior (MAP) diagnosis is only optimal within the tiers that had been worked on. It can no longer be claimed optimality in the overall hypothesis space. Secondly, the tiered-partitioned inference framework needs to store *all past observations*. In the case where the current tier is ruled out, the past observations will be re-applied to the new tier. This means the system should have enough memory. The comparison is as follows: If the computation is done sequentially each time a new observation is made, the memory storage requirement for updating the whole hypothesis

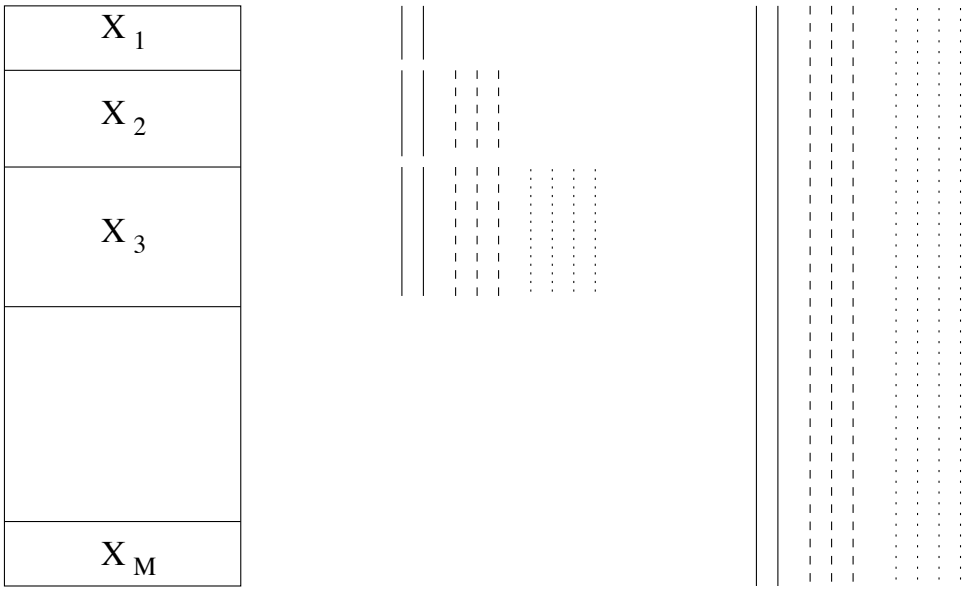


FIGURE 4.4 Computational structure: (a) partition hypothesis space into tiers, (b) computation in the tiered-partitioned inference framework, (c) computation in the whole hypothesis space.

space is 2^k — only the posterior probabilities need to be stored, the observation itself does not need to be stored. In contrast, the memory requirement for the tiered-partitioned inference method is $|\mathcal{X}_j| + O(|observations|)$, i.e., the probability of hypotheses in the current tier needed to be stored, as well as all observations in the past. When the observation history is long, the memory requirement is high. In essence, the tiered-partitioned inference framework reduces the burden on computation, but shifts the burden to memory storage. In practice one can to compress the observation history into some aggregated form as demonstrated in (Kuhn & de Kleer 2008). Another possibility is to store only distinct plans and their failure/success counts as outlined in Chapter 5.

It is important to characterize when this tiered-partitioned inference framework is advantageous. In practical systems, most actions are likely to be good, and the total number of faults is likely to be small. In this case, the single-fault tier can be much more probable than the double-fault tier, and even more so than the triple-fault tier, and so on. Hence it makes sense to focus computational resources to the single-fault tier, and escalate to the higher tiers only when necessary. The higher tier hypotheses are safely ignored because they have minimal probability to start with. The computational savings are tremendous. On the other hand, a pathological case would be the situation where each action has a high (close to 1) probability of having a fault. From the computational point of view, starting from the low cardinality tiers is less attractive, since the low cardinality hypotheses are likely to be ruled out by the observations, and the reduction in inference computation is less significant. Furthermore, the tiered-partitioned inference framework incurs an overhead cost of defining the next subset or tier of hypotheses

to work on every time an existing tier is ruled out. This overhead cost can be high in this pathological case, making the tiered-partitioned inference framework less attractive. On the flip side, this pathological case is rare.

4.3 Partition into single-fault subsystems

Diagnosing a single-fault is computationally efficient. If a k-action system is assumed or known to have a single-fault, only k hypotheses need to be compared, rather than the 2^k hypotheses in the multi-fault case. Given that single-fault inference is computationally efficient, it would be nice to apply this technique whenever applicable. This motivates us to find single-fault subsystems although the overall system can have multiple faults.

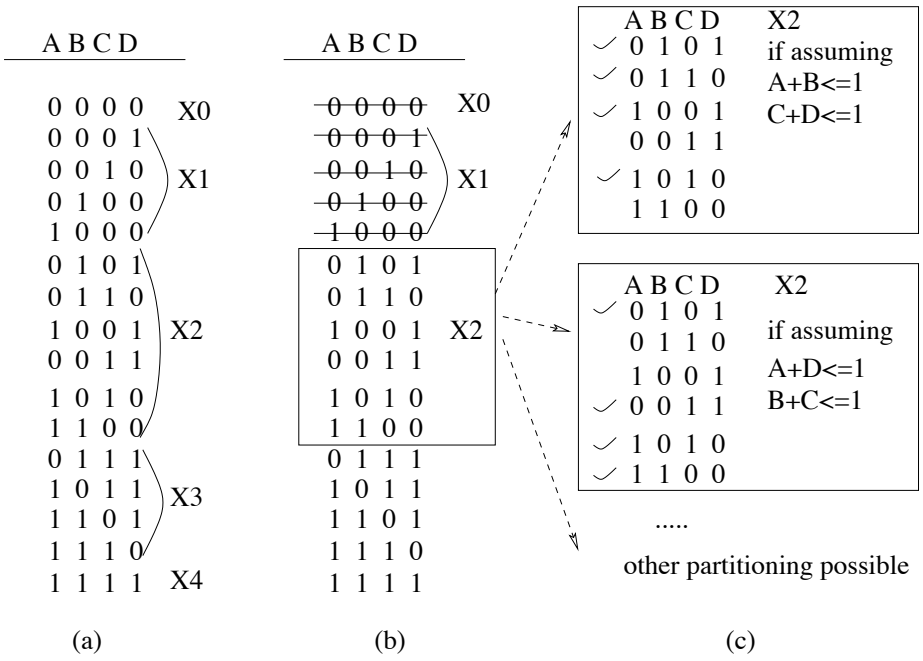


FIGURE 4.5 Example of tiered-partitioned inference: (a) hypothesis space and tiers; (b) escalating to tier $\mathcal{X}_2$; (c) partition $\mathcal{X}_2$ into two groups, each with (at most) a single fault: top box — partition into $\{(AB), (CD)\}$; second box — partition into $\{(AD), (BC)\}$. Other partitions are also possible.

The tiered-partitioned inference idea in the previous section suggests that single-fault diagnosis can be used in the first tier $\mathcal{X}_1$ until data conflict arises. Figure 4.5 shows a simple example system with only four actions $(ABCD)$. Figure 4.5a arranges the hypotheses based on their cardinality. This defines the tiers $\mathcal{X}_0$, $\mathcal{X}_1$, $\mathcal{X}_2$, and so on. The tiered-partitioned inference framework starts from $\mathcal{X}_0$ and $\mathcal{X}_1$. When data suggests that the system $(ABCD)$ has more than one fault, the tiered-partitioned inference escalates to the double-fault tier, $\mathcal{X}_2 \stackrel{def}{=} \{X \mid \sum_i x_i = 2\}$, as shown in Figure 4.5b. At this point, it is know that the overall system $(ABCD)$ has at least two faults, but it is possible that subsystems, for instance,

(AB) and (CD) each have a single fault. In this case, single-fault diagnosis can be applied to the subsystems (AB) and (CD) separately to isolate the faults. The computation with in both subsystems is efficient, as it is single-fault diagnosis. With this partition, the update is restricted to hypotheses into the subset $\mathcal{X}^t = \{X \mid x_A + x_B \leq 1, x_C + x_D \leq 1\}$, shown as the hypotheses marked with check-marks in the top box in Figure 4.5c. The computation is restricted to $\mathcal{X}^t$, hence fast.

The question now is to seek a good partitioning such that the partitioned subsystems are most likely to have single fault. Formally, the partitioning problem is as follows: given an overall system with a set of components $COMPS$, the partitioning divides $COMPS$ into two groups $COMPS_1$ and $COMPS_2$ such that $COMPS_1 \cup COMPS_2 = COMPS$ and $COMPS_1 \cap COMPS_2 = \emptyset$. For instance, in the example in Figure 4.5, $COMPS_1 = (AB)$ and $COMPS_2 = (CD)$ is a valid partition. Note that this partitioning is not unique: $(ABCD)$ can be partitioned into $\{(AB), (CD)\}$ (top box in the figure), or $\{(AD), (BC)\}$ (second box in the figure)[1] or other combinations. The next section addresses the question of which partition to use. The basic idea is to examine the correlation between system components to find those subsets which collectively contain only a single fault with maximum probability.

Given a subsystem partition and the corresponding subset of hypotheses $\mathcal{X}^t$ assuming at most a single fault within each subsystem, the algorithm restricts the posterior updates to the subset, until the observation data conflicts with $\mathcal{X}^t$. In this case, the current partitioning has to be replaced by a more suitable partitioning within the existing tier $\mathcal{X}_2$. When the whole tier $\mathcal{X}_2$ is ruled out by observation, the framework escalates to the third tier $\mathcal{X}_3$ (the collection of hypotheses with 3 faulty actions) and partitions the overall system into three subsystems, each of which hopefully contains a single fault. The whole process repeats as more observations are made.

4.4 How to partition

4.4.1 Criterion for partitioning

As mentioned in the previous section, when the single-fault assumption fails, the framework escalates to $\mathcal{X}_2$ and assumes that the overall system has two faults. In that process the k-action system is partitioned into two subsystems, or two groups, within which there is likely to be at most one fault.

There are many ways of partitioning a system into two groups. For example, $(ABCD)$ can be partitioned into $C_4^1 + C_4^2/2 = 7$ ways. Which one is more preferable? What optimality criteria should be used? The intuition is clear: The partition which has only single-fault subsystem with maximal probability is preferred.

> **Criterion:** The partition (of the action set) which captures maximal probability mass, i.e., maximizing the probability $\sum_{X \in \mathcal{X}^t} Pr(X)$.

[1] The bracket are used to denote a group within which there is believed to be only single-fault, and the curly bracket for a collection of groups.

For instance, in Figure 4.5, partitioning into subsystems $\{(AB), (CD)\}$, shown as the top block on the right hand side, captures hypotheses $\{0101, 0110, 1001, 1010\}$. There are two hypotheses $\{0011, 1100\}$ that violates the single-fault assumption in (CD) and (AB) respectively. If the probabilities $Pr(0011)$ and $Pr(1100)$ are small, this means (AB) and (CD) are likely to have single-fault, and the partition is advantageous. On the other hand, if $Pr(0011)$ and $Pr(1100)$ are big, this mean the single-fault subsystem assumption is questionable. To compare the two partitions $\{(AB), (CD)\}$ and $\{(AC), (BD)\}$, the probability mass of missed hypotheses can be compared, in this case, $Pr(0011) + Pr(1100)$ and $Pr(0110) + Pr(1001)$. The partition with a lower (not captured) probability mass is more favorable.

4.4.2 A partitioning algorithm

Now with the optimality criterion, how should the partitioning algorithm be designed? The straightforward solution is to compare all partitions and see which partition captures the largest probability sum, but this is too expensive with complexity 2^k. The question is can a partitioning be found which is good (maybe suboptimal) with much less computation time? First the case of partition into two groups is discussed.

> **Intuition:** For a group of actions to have a single fault, i.e., $\sum_{i \in P} x_i = 1$, the x_i's would have to be negatively correlated.

In other words, when one member x_i increases, there must be another x_j which decreases in order to maintain the constant sum. This means, the algorithm should look for actions with significant negative correlation and group them into a group. In contrast, if two members are positively correlated, i.e., when one increases/decreases, the other one increases/decreases too, then these two actions should not be grouped into the same group.

Using this heuristics an algorithm can be designed, which examines the correlation coefficient between actions. The correlation coefficient is defined as

$$\begin{aligned} \rho(i,j) &\stackrel{def}{=} \frac{Cov(x_i, x_j)}{\sigma_i \sigma_j} \\ &= \frac{E\left[(x_i - \mu_i)(x_j - \mu_j)\right]}{\sigma_i \sigma_j} \end{aligned} \tag{4.5}$$

For any two actions i and j, x_i and x_j are the indicators of their respective health (0 if the action is good, and 1 if the action is bad), μ_i and μ_j are the respective mean of x_i and x_j, and σ_i and σ_j are their respective standard deviations. The correlation coefficient $\rho(i, j)$ measures the dependency between x_i and x_j. It has the following properties: (a) $-1 \leq \rho \leq 1$; (b) the sign of ρ shows whether the two random variables are positively or negatively correlated; (c) $\rho = 1$ if $x_i = x_j$, and $\rho = -1$ if $x_i = -x_j$; (d) symmetry: $\rho(i, j) = \rho(j, i)$. Given a set of hypotheses $\mathcal{X}$ and their respective probability values, one can easily compute the mean $\{\mu_i\}_{i=1,\ldots,k}$, the standard deviation $\{\sigma_i\}$, the covariance matrix $\{Cov(x_i, x_j)\}_{i,j=1,\ldots,k}$, and the correlation coefficient $\rho(i, j)$ for any i and j. The computational complexity is linear in the number of hypotheses.

The algorithm is the following:

1. From the hypotheses and their respective probabilities, evaluate the correlation coefficient $\rho(i,j)$ for any (i,j). The result is a correlation coefficient matrix of size $k \times k$.

2. Find the two group seeds i_1 and i_2 as the action which have the highest correlation $E(x_i^2)$ values. This indicates that these two actions are more likely to have a fault than the others. In the case of a tie, the seeds are selected randomly. The two groups "grow" around the seeds. Previously a random selection scheme has been used: randomly select the first seed i_1, and then find the second group seed i_2 as the action which has the highest correlation with i_1. Since these two are positively correlated, they should not be in the same group. The max-correlation scheme works best in simulations performed as part of the experiments.

3. For any remaining action j, compare the correlation coefficients $\rho(i_1,j)$ and $\rho(i_2,j)$. The action is assigned to group 1 if $\rho(i_1,j) < \rho(i_2,j)$ and to group 2 if otherwise.

Computational complexity : The computation is primarily on the computation of $\{\rho(i,j)\}$. The complexity if $O(k^2 \cdot |\#\text{ of hypotheses}|)$ — there are k^2 correlation coefficients, and computing each need to go through all hypotheses in the current tier. In contrast, the "oracle" scheme of comparing all partitioning combinations has complexity $O(2^k \cdot |\#\text{ of hypotheses}|)$.
Performance: Despite its simplicity, this greedy algorithm works well. In our simulation, a large number (100) of random simulations were repeated and the partitioning scheme was compared against the enumeration of 2^k possible partitions. The proposed partitioning selection scheme has the following performance:

- Against the missing probability metric: the partition selection method is at about the 85% percentile among all 2^k partitions, i.e., around 15% partitions are better than the proposed solution, and 85% are worse. But the computational complexity is significantly less.

- Compared to the "oracle" — the partition with smallest missing probability, the partition scheme produces a slightly larger missing probability, on average 3–5% larger.

Example — Consider a 5-action production system $(ABCDE)$. The observations are as follows: (1) observing a fault with plan $(ABCDE)$; (2) observing a fault with plan (ABC); (3) observing a fault with (DE). At this point, the single fault assumptions are eliminated. Assume each action is defective with a prior probability $r = 0.1$. Further assume all faults are persistent. In this case, the covariance coefficient matrix is:

$$\rho = \begin{pmatrix} 1 & -0.5 & -0.5 & 0 & 0 \\ -0.5 & 1 & -0.5 & 0 & 0 \\ -0.5 & -0.5 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & -1 & 1 \end{pmatrix} \tag{4.6}$$

The partitioning algorithm selects B and D as group seeds and partitions actions into two subsystems (ABC) and (DE), which agrees with earlier intuition.

A similar problem is optimal number partitioning (Korf 1995), which partitions a set of integer numbers into two groups with equal sums. However, there is a fundamental difference: the optimal number partitioning is deterministic, while our partitioning problem is inherently statistical and must work with uncertainties. As a result, the algorithms for the two problems are quite different.

4.4.3 Preparing probability distribution for partitioning

The algorithm above requires the computation of correlation coefficients $\{\rho(i,j)\}_{i,j=1,2,\ldots,k}$. They are computed based on a set of hypotheses and their respective probability values. Should this hypothesis set be the entire hypothesis space ($\mathcal{X}$, size 2^k)? or a smaller subset? It may be sufficient to compute the distribution for a subset. For instance, if the first tier (the single fault hypotheses tier $\mathcal{X}_1$) is ruled out, and the framework must escalate to double faults, only the double fault hypothesis tier $\mathcal{X}_2$ needs to examined , since other hypotheses are out of the representation of two-group partition anyway. Therefore the other hypotheses will not be covered by the partitioning. In our tiered-partitioned inference framework, tier $\mathcal{X}_2$ is used for partitioning into two groups. Likewise, if $\mathcal{X}_2$ is ruled out by observations, an escalation to the triple-fault tier $\mathcal{X}_3$ has to be performed, and partition the k-action system into three groups. The partitioning is computed based on the probability values of all hypotheses in $\mathcal{X}_3$.

The algorithm described above can be modified to partitioning components into any number of groups. The extension is straight-forward: more group seeds can be selected in Step 2, and the seeds can then grow into groups.

4.5 Implementation and simulation

As an example to illustrate the advantages and drawbacks of the tiered-partitioned inference approach, consider diagnosis of a production plant. Assume that actions are independent, and each action is defective with a known prior probability r. All faults are intermittent, i.e., a defective action damages any product passing it with a known probability q, known as the intermittency probability. In practice, each action may have its own r and q, different from the others. For simplicity, assume that all actions share the same r and q value.

Mathematically, the prior probability is

$$Pr(X) = \left(r^{\sum_i x_i}\right) \cdot \left((1-r)^{k-\sum_i x_i}\right).$$

Given an plan p, the likelihood of observing an output Y (0 for good, and 1 for damaged) is

$$Pr(Y|X) = \begin{cases} (1-q)^{n(P=p,X)} & \text{if } Y=0 \\ 1-(1-q)^{n(P=p,X)} & \text{if } Y=1 \end{cases}$$

Here the exponent $n(P=p,X)$ is the number of defective actions involved in the production

plan p given the hypothesis X. This is actually quite intuitive: a product is undamaged only when none of the defective actions malfunctions, hence the probability is the action-wise good probability $(1 - q)$ raised to the power $n(p, X)$.

Now with prior and likelihood probabilities specified, Bayesian updates (Equation 4.1) can be performed. Two diagnosis schemes are compared: (a) a baseline scheme applying all observations sequentially to update the posterior belief $Pr(X|Y)$ for all $X \in \mathcal{X}$ that has not been ruled out by previous observation data; and (b) the tiered-partitioned inference scheme described in Sections 4.2– 4.4. To evaluate the performance, 300 random trials were simulated, each with an observation sequence of 400 randomly generated production plans and corresponding execution observations. Performance is assessed based on cost and accuracy:

- Computational cost: for the baseline scheme, computational cost is measured as the accumulative number of posterior updates, i.e., how many times (Equation 4.1) is executed. For tiered-partitioned inference, the cost is the sum of two parts: (i) the inference cost, i.e., the number of posterior updates, and (ii) the overhead cost of partitioning actions into subsystems, measured as the number of hypotheses sieved through to compute the correlation coefficient (Equation 4.5). Table 4.1 reports the two terms, separated by a ";" in the third column.

- Diagnosis accuracy, measured as the total number of bits that X_{MAP} differ from the ground truth. Ideally, if X_{MAP} recovers the ground truth, this term should be 0. However, this is often not achieved, even in the baseline inference scheme. This is due to the fact that the observations may not be sufficient, for instance, if some defective actions are never used in production, and/or the faults are intermittent, hence the defects are never observed.

- Partition count, measured as the total number of partitions. The baseline inference scheme does not perform partitioned inference, hence it partitions the system into one partition.

Table 4.1 reports the results for a 10-action production system, averaged over 300 random trials. Each row corresponds to a value of r, ranging from 0.05 to 0.9. Small r implies a healthy system, while $r = 0.9$ corresponding to an extremely shaky system where all actions are likely to fail. The extremes are used to provide insights. Note the following:

(1) The computational cost saving using the tiered-partitioned inference scheme is significant. For instance, with $r = 0.05$, the tiered-partitioned inference scheme has a computation cost less than 1% of the baseline scheme. With $r = 0.9$, the tiered-partitioned inference computation is around 10% of the baseline computation.

(2) The baseline scheme is on average more accurate than the tiered-partitioned inference. This is expected, since the tiered-partitioned inference is an approximation.

(3) Tiered-partitioned inference is most advantageous when r is small. The inference accuracy is almost as good as the baseline scheme for $r \leq 0.2$, and the computation cost is 1–2 magnitudes order lower. This shows the benefit of tiered-partitioned inference. The good performance is not surprising, as a system with small r is what tiered-partitioned inference was originally designed for.

(4) As r increases, tiered-partitioned inference incurs an increasingly heavy partitioning overhead cost (second number in the third column). This is due to the fact that the system has

	Computation cost		Diagnosis accuracy		Partition count	
	baseline	tiered-part	baseline	tiered-part	baseline	tiered-part
$r = 0.05$	285779.7	2268.5; 63.3	0.03	0.03	1	1.05
$r = 0.1$	265003.1	2051.9; 448.3	0.17	0.13	1	1.35
$r = 0.2$	236468.3	2705.2; 1435.7	0.47	0.59	1	2.03
$r = 0.5$	175757.8	6293.7; 4610.0	1.51	2.36	1	3.85
$r = 0.9$	141973.8	7875.2; 6470.0	1.16	5.07	1	4.90

TABLE 4.1 Tradeoff between computational cost and diagnosis accuracy. This table is generated assuming the intermittency probability of $q = 0.1$. The second column reports computation cost of the baseline scheme measured as the number of hypotheses updated; the third column reports the computation cost for tiered-partitioned inference for Bayesian update and partitioning overhead, and the last two columns report diagnosis accuracy of the two schemes, measured as the number of bits that MAP estimate differs from the ground truth.

more defective actions, and the single-fault assumption within subsystems is often ruled out by observation data. In this case, the partitioning operations are frequently repeated. The overhead cost makes computational savings less dramatic. Furthermore, tiered-partitioned inference becomes less accurate. For instance, in the last row ($r = 0.9$), the tiered-partitioned inference diagnosis has roughly 5 bits flipped. It misses to detect 5 defective actions. In comparison, the baseline has 1.16 bits flipped on average. Note that this is due to their different strategies: the baseline scheme seeks exact inference and optimal diagnosis, while tiered-partitioned inference favors low-cardinality diagnosis. Tiered-partitioned inference stays at lower tiers as long as the lower tiers can explain the data. This is similar to a minimal diagnosis: the minimal candidate set can be quite different from the underlying ground truth, especially when the faults are intermittent and the number of observations is limited.

4.6 Conclusion

This chapter has presented a new framework for efficiently computing multiple fault diagnoses. This framework introduces the generic notion of tiered-partitioned inference which focuses search and inference on the set of hypotheses most likely to contain the fault(s). Past approaches which focus on most probable, subset-minimal, or minimum cardinality approaches are all instances of the more general tiered-partitioned approach. In addition, this chapter introduced the notion of partitioning the actions such that efficient, linear, single fault inference can be used (and never requires the usual multiple-fault inference scheme). By performing single fault diagnosis on each partition, the potential computational inference on each partition is avoided. For smaller cardinality diagnoses, the inference saving outweighs the cost of computing partitions (including recomputing partitions when they are discovered to be unsuccessful).

CHAPTER 5

Continuously Estimating Persistent and Intermittent Failure Probabilities

Almost all previous work on model-based diagnosis has focused on persistent faults. However, some of the most difficult to diagnose faults are intermittent. It is very difficult to isolate intermittent faults which occur with low frequency but yet at high enough frequency to be unacceptable. For example, a printer which prints one blank page out of a thousand or a computer that spontaneously reboots once per day is unacceptable. Accurate assessment of intermittent failure probabilities is critical to diagnosing and repairing equipment. This chapter presents an overall framework for estimating component failure probabilities which includes both persistent and intermittent faults.

5.1 Introduction

Almost all previous work on model-based diagnosis has focused on persistent faults where the prior probability of component failure is provided by the manufacturer or estimated from fleet-wide service records (Section 3.2.1). However, some of the most difficult to diagnose faults are intermittent. It is very difficult to isolate intermittent faults which occur with low frequency, but yet at high enough frequency to be unacceptable. For example, a printer which prints one out of one thousand pages blank or a computer that spontaneously reboots once per day is unacceptable. Accurate assessment of intermittent failure probabilities is critical to diagnosis and sustainable system operation. This chapter presents an overall framework for estimating component failure probabilities, which includes both persistent and intermittent faults. These estimates are constantly updated while the system is operating.

The presented examples are drawn from hyper-modular, multi-engine printers (Section 2.4.1). A reprographic system receives a continuous stream of print jobs and each print job consists of a sequence of sheets of paper. The planner constructs an optimal plan for each sheet which specifies a full trajectory through potentially dozens of modules. Each module type has a set of actions it can perform. One of those actions may be faulty, but the module may always succeed at other actions. Therefore, the proposed approach applies the framework to actions, not modules. Each capability fails approximately independently. Figure 5.1 illustrates a three way module with six capabilities. Figure 5.2 illustrates five modules of the two types connected together. Circles indicate rollers, triangles indicate sensors, and two sheets of paper are indicated in red. Note that three modules can be acting on the same sheet of paper at one time.

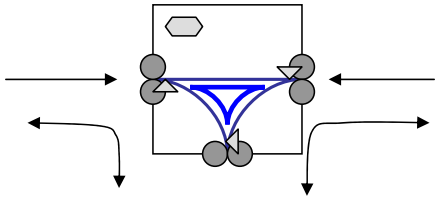


FIGURE 5.1 A more detailed figure of a three way module. The 6 possible paper movements (capabilities) are indicated on the diagram.

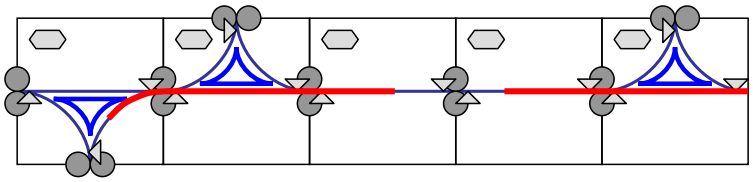


FIGURE 5.2 A more detailed figure of five connected modules moving two sheets of paper.

It is possible to design machine configurations where a failure in the output capability of one module cannot be distinguished from a failure in the input capability of the connected module. In the presented framework, this will show up as a double fault when in fact only one of the two modules if faulted. This confusion can be avoided by applying an idea from digital circuits to collapse indistinguishable faults. In addition, multiple faults are allowed. Experiments with printers have shown that most equipment contains multiple, low frequency, intermittent faults. High-end reprographic systems operate more-or-less continuously providing a constant stream of observations and exceptions.

Consider a system as a 6-tupel $\Sigma^{obs} =< \mathcal{S}, \mathcal{A}, \mathcal{E}, T, \Phi >$ (Section 2.6.1.2, Definition 4), which is controlled by a Self-diagnosing Agent (Chapter 3). Further assume that a system operates more-or-less continuously over a long period of time providing a constant stream of executed plans and corresponding execution observations. Each plan p consists of a sequence of actions, denoted $p =< a_1, \ldots, a_k >$. Following Definition 15 and 16, in Section 3.3, a plan execution results in $\mathcal{Y}(p) = 1$ if and only if plan p executed not abnormal and $\mathcal{Y}(p) = 0$ otherwise.

These systems present two challenges to model-based diagnosis:

- The system may operate continuously over a long period of time with very high speed while executing multiple plans in parallel such that retaining full details of behavior of all past executions is impractical, and
- The system may experience intermittent faults, which occur with very low frequency, but yet at high enough frequency to be unacceptable.

To efficiently update the probabilities an approach is needed, which derives the correct posterior probabilities with minimal space and time. Recording all past data and updating all possible hypothetical diagnoses is impractical, yet no valuable information should be thrown away. The presented approach addresses both challenges by estimating the probabilities based on maintaining a limited set of counters instead of the entire history. The counter convention is adopted from (Abreu, Zoeteweij & van Gemund 2006), which associates two counters with each executed action a_i:

- c_{f,a_i}: number of failed plan executions where a_i was use.
- c_{s,a_i}: number of succeeded plan executions where a_i was used.

Such counters can be leveraged to estimate the failure probabilities of system actions. These estimates are critical to detect and isolate intermittent faults during continuous operation (sequential diagnosis) and to stimulate future operation to avoid faulty components (prognosis). The presented approach is applicable to systems with faults single or multiple in numbers and persistent or intermittent in their appearance. Figure 5.3 illustrates all combinations.

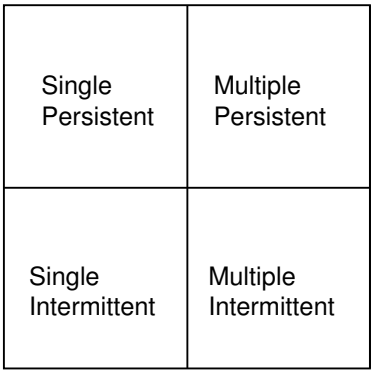


FIGURE 5.3 Fault combinations considered by the proposed approach.

The following simplifying assumptions are made as they apply for most planning systems. These systems have some striking differences from the commonly explored digital circuits analyzed in most of the model-based diagnosis literature (Section 3.2.1):

- Errors can not be masked or cancelled out, e.g. a damaged sheet can not be repaired by the machine. However, there are systems where faults may be masked, e.g. digital systems where internal faulty signals can be masked to produce correct outputs. The outlined approach still applies for such systems, but requires more reasoning to determine whether a faulty output is masked. (See (de Kleer 2007*a*).)
- Fault probabilities are stationary. The presented approach can be easily extended to accommodate slowly drifting probabilities by introducing discounting over time.
- Observations do not affect machine behavior. This assumption is made in most approaches to diagnosis.

- All faults are distinguishable. This is simply for exposition: as in digital circuits, indistinguishable faults are collapsed.

These assumptions hold in a broad range of systems. The only input the presented approach requires is a sequence of plan and execution-observation pairs where the plan is expressed as a set of actions. All observable planning agents provide this information. Over the next sections a framework is presented that enables the estimation of failure probabilities for systems with faults single or multiple in number and persistent or intermittent in their appearance.

5.2 Single Persistent Fault

This section considers the estimation of single persistent fault probabilities. This case follows from GDE (de Kleer & Williams 1987). Let $Pr(a_i)$ be the probability action a_i is faulted. Let p be a plan that produces the observation $\mathcal{Y}(p)$. The standard notation is used, with uppercase symbols denoting random variables and lowercase symbols denoting a particular realization: The sequential Bayesian filter (Berger 1995) is:

$$Pr_t(A|Y,P) = \alpha Pr(Y|A,P)Pr_{t-1}(A). \tag{5.1}$$

Where α is chosen so that the posterior probabilities sum to 1 (presuming the knowledge that there is a fault is given).

$Pr(Y|A,P)$ is 1 in situations where $c_{f,A}$ are incremented, otherwise it is 0. Namely, if the action is not element of a failing plan it is exonerated by the single fault assumption, and if the action is used in a successful plan it is exonerated because every fault is assumed to be observed. Figure 5.4 illustrates the possibilities.

o / u	Fail	Success
Used	1	0
Not Used	0	1

FIGURE 5.4 Summary of the observation likelihood in the single fault persistent case. Note that when diagnoses can have multiple faults, the test for whether a diagnosis is used generalizes to whether any of its actions are used in the current plan.

Assume that at $t = 0$ all actions fail with prior probability $Pr_0 = 10^{-10}$. Consider the sequence of pairs: ($p_1 =< a_A, a_B, a_C, a_D, a_E, a_F >$, $\mathcal{Y}(p_1) = 1$), ($p_2 =< a_A, a_B, a_C >$, $\mathcal{Y}(p_2) = 0$), ($p_3 =< a_E, a_F >$, $\mathcal{Y}(p_3) = 0$). After observing the execution of p_1 it can

be concluded that one of the six actions must be faulted. As the priors are all equal, each action must be faulted with probability $\frac{1}{6}$. Since all faults are assumed to be persistent and that all faults manifest, a successful plan execution exonerates all the actions of the plan. Thus observing plan p_2 indicates that a_A ,a_B and a_C are all working correctly. Finally, observing the execution of p_3 exonerates action a_E and a_F. Therefore, a_D is faulted with probability 1 (see Table 5.1).

TABLE 5.1 The resulting posterior probabilities $Pr(A = a|Y, P = p)$ over one sequence of plans assuming a single persistent fault.

t	$a = a_A$	$a = a_B$	$a = a_C$	$a = a_D$	$a = a_E$	$a = a_F$
0	10^{-10}	10^{-10}	10^{-10}	10^{-10}	10^{-10}	10^{-10}
1	$\frac{1}{6}$	$\frac{1}{6}$	$\frac{1}{6}$	$\frac{1}{6}$	$\frac{1}{6}$	$\frac{1}{6}$
2	0	0	0	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$
3	0	0	0	1	0	0

5.3 Single Intermittent Fault

This section considers the estimation of single intermittent fault probabilities. This case extends the model for intermittent faults presented in (de Kleer 2007*a*), which was informed by (Koren & Kohavi 1977). In the case of intermittent faults, $Pr(Y|A, P)$ is still 0 in case $c_{s,a}$ and 1 in case $c_{f,m}$. Otherwise, $Pr(Y|M, P)$ needs to be estimated using the counters. The probability that action a produces an incorrect output if faulted is calculated as follows:

$$\frac{c_{f,a}}{c_{f,a} + c_{s,a}}. \tag{5.2}$$

(The denominator can never be 0 as will be described later in Section 5.3.1.)

Let $Pr_0(A)$ be the prior probability that a is faulted. Given a particular observation Y, Bayes rule gives:

$$Pr_1(A|Y, P) = \alpha Pr(Y|A, P)Pr_0(A). \tag{5.3}$$

The observation likelyhood $Pr(Y|A, P)$ is estimated from the counters. If the observation is a failure ($\mathcal{Y}(p) = 1$) and a is used in plan p, then:

$$Pr(Y = 1|A = a, P = p) = \frac{c_{f,a}}{c_{f,a} + c_{s,a}}, \tag{5.4}$$

and if it is a success ($\mathcal{Y}(p) = 0$) and a is used in plan p, then:

$$Pr(Y = 0|A = a, P = p) = \frac{c_{s,a}}{c_{f,a} + c_{s,a}}, \tag{5.5}$$

otherwise if it is a success ($\mathcal{Y}(p) = 0$), a cannot affect Y as a is not in p, then:

$$Pr(Y = 0|A = a, P = p) = 1, \tag{5.6}$$

otherwise,

$$Pr(Y = 1|A = a, P = p) = 0, \tag{5.7}$$

(captures the single fault assumption). Figure 5.5 summarizes the all four possibilities.

i \ o	Fail	Success
Used	$\frac{m_{11}}{m_{11}+m_{10}}$	$\frac{m_{10}}{m_{11}+m_{10}}$
Not Used	0	1

FIGURE 5.5 Summary of the observation likelihood in the single fault intermittent case.

After many iterations of Bayes rule, intuitively,

$$Pr_t(A|\mathbf{Y}, \mathbf{P}) = \alpha Pr(good)^g Pr(bad)^b Pr_0(A), \tag{5.8}$$

where there are g observations of a being used in a plan execution with good behavior and b observations of a being used in a plan execution bad behavior. Formally:

$$Pr_t(A|\mathbf{Y}, \mathbf{U}) = \begin{cases} 0 & \text{if } \exists P \in \mathbf{P} \text{s.t.} \mathcal{Y}(p) = 1 \wedge A \notin P \\ \alpha w Pr_0(A) & \text{otherwise} \end{cases} \tag{5.9}$$

where,

$$w = [\frac{c_{s,a}}{c_{f,a} + c_{s,a}}]^{c_{s,a}} [\frac{c_{f,a}}{c_{f,a} + c_{s,a}}]^{c_{f,a}}. \tag{5.10}$$

Consider again the example of three itineraries: ($p_1 =< a_A, a_B, a_C, a_D, a_E, a_F >, \mathcal{Y}(p_1) = 1$), ($p_2 =< a_A, a_B, a_C >, \mathcal{Y}(p_2) = 0$), ($p_3 =< a_E, a_F >, \mathcal{Y}(p_3) = 0$). The probabilities are updated as follows: After observing p_1 all $c_{f,a}$ counters are 1 and the $c_{s,a}$ 0, therefore w's are 1. After observing p_2 the counters $(c_{s,a}, c_{f,a})$ for $\{a_A, a_B, a_C\}$ are all $(1, 1)$ and the counters for $\{a_D, a_E, a_F\}$ are all $(0, 1)$. Therefore, $w = \frac{1}{4}$ for $\{a_A, a_B, a_C\}$ and 1 for the rest. Next p_3 is observed. The counters for $\{a_A, a_B, a_C\}$ are all $(1, 1)$. The counters for a_D are $(0, 1)$ and the counters for $\{a_E, a_F\}$ are: $(1, 1)$. Now suppose plan $p1$ is repeatly executed for seven time. Table 5.2 illustrates how the posterior probabilities evolve.

Working with the same six actions, consider a slightly more realistic example. Assume that the prior probabilities of intermittent failures are equal for all the actions. Consider the

TABLE 5.2 The resulting posterior probabilities $Pr(A = a|Y, P)$ over a sequence of plans assuming a single intermittent fault.

t	$a = a_A$	$a = a_B$	$a = a_C$	$a = a_D$	$a = a_E$	$a = a_F$
0	10^{-10}	10^{-10}	10^{-10}	10^{-10}	10^{-10}	10^{-10}
1	$\frac{1}{6}$ .17	$\frac{1}{6}$ .17	$\frac{1}{6}$ .17	$\frac{1}{6}$ .17	$\frac{1}{6}$ .17	$\frac{1}{6}$ .17
2	$\frac{1}{15}$ .07	$\frac{1}{15}$ .07	$\frac{1}{15}$ .07	$\frac{4}{15}$ .27	$\frac{4}{15}$ .27	$\frac{4}{15}$ .27
3	$\frac{1}{9}$ .11	$\frac{1}{9}$ .11	$\frac{1}{9}$ .11	$\frac{4}{9}$ .44	$\frac{1}{9}$ .11	$\frac{1}{9}$ .11
4	$\frac{16}{107}$ .15	$\frac{16}{107}$ .15	$\frac{16}{107}$ .15	$\frac{27}{107}$ .25	$\frac{16}{107}$ .15	$\frac{16}{107}$ .15
10	.16	.16	.16	.18	.16	.16

case in which action a_D is intermittently faulted and behaves abnormal one out of every 1001 times (assume the example starts with execution 1001). Suppose that the system, e.g. a printer, repeatedly executes the plans: $p_1 =< a_A, a_B, a_E, a_F >$, $p_2 =< a_C, a_B, a_E, a_D >$, $p_3 =< a_A, a_B, a_C >$, $p_4 =< a_F, a_E, a_D >$. After seeing 2000 plan executions the counts for $\{a_A, a_F, a_C, a_D\}$ are $c_{s,a} = 1000, c_{f,a} = 0$ and counts for $\{a_B, a_E\}$ are $c_{s,a} = 1500, c_{f,a} = 0$. Suppose D behaves abnormal during the executing plan p_2. By the single fault assumption, actions a_A and a_F are exonerated and their posterior probability of failure is now 0. The w for actions a_B and a_E are now:

$$[\frac{1500}{1501}]^{1500} \frac{1}{1501} = .000245. \tag{5.11}$$

The term is higher for a_C and a_D as fewer samples of good behavior have been observed:

$$[\frac{1000}{1001}]^{1000} \frac{1}{1001} = .000368. \tag{5.12}$$

Normalizing, the posterior probability for a_B, and a_E failing are: 0.2 and for a_C, a_D are: 0.3. Suppose no errors are seen in the next 2000 itineraries. Then, a_D behaves abnormal in $p3$. By the single fault assumption, actions a_B and a_C are now exonerated. The values for w for a_D and a_E are now:

$$[\frac{2000}{2002}]^{2000} [\frac{2}{2002}]^2 = 1.352 \times 10^{-7}, \tag{5.13}$$

$$[\frac{3000}{3002}]^{3000} [\frac{2}{3002}]^2 = .601 \times 10^{-7}. \tag{5.14}$$

Normalizing $Pr(A = a_D|Y) = 0.7, Pr(A = a_E|Y) = 0.3$.

TABLE 5.3 The resulting posterior probabilities $Pr(A = a|Y, P)$ over a more complex sequence of itineraries assuming a single intermittent fault.

t	$a = a_A$	$a = a_B$	$a = a_C$	$a = a_D$	$a = a_E$	$a = a_F$
2001	0	.2	.3	.3	.2	0
4002	0	0	0	.7	.3	0
6003	0	0	0	.77	.23	0
8004	0	0	0	.83	.17	0
16008	0	0	0	.96	.04	0

In practice faults never occur with such regularity as in Table 5.3. Instead, every sequence of plans will yield different posterior probabilities.

As can be seen in this example, the restriction to single faults is a very powerful force for exoneration. All the actions not exonerated will have the same $c_{f,a}$ count. This results from the fact that under the single fault assumption, only actions that been used in every failing run remain suspect. Hence they have the same $c_{f,a}$. In the example, $c_{f,a} = 1$ in equations 5.11 and 5.12. After more observations, $c_{f,a} = 2$ in equations equations 5.13 and 5.14.

5.3.1 Incorporating prior counts

So far it was presumed that nothing is known about the counts prior to making observations. If counts are initially 0, then the denominator of equation 5.10 will be 0. One possible approach to avoid this is Laplace's adjustment: make all initial counts 1, which is equivalent to assuming an uniform prior over $Pr(a)$. Another approach which is utilized in this approach is to ensure that equation 5.10 is never evaluated until an observation is made. The current observation is always included in counts, thus the denominator of equation 5.10 will never be 0 whenever it is utilized. Both approaches converge to the same in the limit as the number of observations grows to infinity.

One important detail left out of the examples is that if an action has operated perfectly for very large counts it takes too many failing samples before its posterior probability rises sufficiently to be treated as a leading candidate diagnosis. Therefore, for it is advised to apply a small exponential weighting factor λ at every increment such that counts 100,000 in the past will have less weight then new samples, e.g. $\lambda = 0.99999$.

5.4 Multiple Persistent Faults

This section considers the estimation of multiple persistent fault probabilities. Over the last sections actions have been considered to estimate the probabilities. Instead of actions, this

section considers diagnosis hypotheses D. The number of possible diagnoses is exponential in the number of actions, however, Chapter 4 introduces a Tiered-partitioning Diagnosis Framework, which considers only a small subset of hypothesis at any given time. For illustration lets consider for the moment the general case.

Analogous to the single persistent fault case:

$$Pr_t(D|Y,P) = \alpha Pr(Y|D,P)Pr_{t-1}(D). \tag{5.15}$$

To determine the prior probability of a diagnosis $Pr_0(D)$ it is presumed that actions fail independently. Let $good(D)$ be all the good actions of D and $bad(D)$ all the bad action of D, than $Pr_0(D)$ is:

$$Pr_0(D) = \prod_{g \in good(D)} Pr_0(g) \prod_{b \in bad(D)} (1 - Pr_0(b)). \tag{5.16}$$

It remains to determine $Pr(Y|D,P)$. If all the actions used in a plan are a subset of the good actions of a diagnosis d, then $Pr(Y = 1|D = d, P) = 0$ and $Pr(Y = 0|D = d, U) = 1$. In every remaining case (i.e., if any of the used actions are bad in d), then $Pr(Y = 0|D = d, U) = 0$ and $Pr(Y = 1|D = d, U) = 1$. Figure 5.6 summarizes these results.

d \ o	Fail	Success
$bad(d) \cap U \neq \emptyset$	1	0
$bad(d) \cap U = \emptyset$	0	1

FIGURE 5.6 Summary of the observation likelihood in the multiple persistent case for an observation y of plan p.

For diagnostic purposes, the posterior probability that a particular action is faulted is computed by:

$$Pr(a|y_1, \ldots, y_t) = \sum_{d \ s.t. a \in bad(d)} Pr(d|y_1, \ldots, y_t). \tag{5.17}$$

5.5 Multiple Intermittent Faults

This section considers the estimation of multiple persistent fault probabilities. Bayes rule can be applied as before:

$$Pr_t(D|Y,P) = \alpha Pr(Y|D,P)Pr_{t-1}(D). \tag{5.18}$$

Let $Pr_b(a)$ be the probability that an action a produces an incorrect output when faulted. Let us first assume that $Pr_b(a)$ is given. In this case $P_t(D|O,P)$ is given by:

$$Pr_t(D|\mathbf{Y},\mathbf{P}) = \alpha w Pr_0(A) \tag{5.19}$$

$$w = \begin{cases} 1 - \prod_{a \in bad(D) \cap P}(1 - Pr_b(a)) & \text{If } \mathcal{Y}(P) = 1 \\ \prod_{a \in bad(D) \cap P}(1 - Pr_b(a)) & \text{If } \mathcal{Y}(P) = 0 \end{cases} \tag{5.20}$$

As in the previous analyses, the posterior probabilities of the diagnoses are obtained by repeatedly applying Bayes rule. Before introducing a more direct method consider the result of applying Bayes rule repeatedly. Iterating Bayes rule results in:

$$\begin{aligned} w = & \prod_{P \text{ s.t. } \mathcal{Y}(P)=1} [1 - \prod_{a \in bad(D) \cap P} (1 - Pr_b(a))] \\ & \times \prod_{P \text{ s.t. } \mathcal{Y}(P)=0} \prod_{a \in bad(D) \cap P} (1 - Pr_b(a)) \end{aligned} \tag{5.21}$$

Consider both terms separately. The second term, success, can be computed simply by maintaining the counter (as in the single fault case) $c_{s,a}$ for each action:

$$\prod_{P \text{ s.t. } \mathcal{Y}(P)=1} \prod_{a \in bad(d) \cap P} (1 - Pr_b(a)) = (1 - Pr_b(a))^{c_{s,a}}. \tag{5.22}$$

To compute the first term a single counter $c_{f,s}$ can be associated with each set of actions s utilized in a failing plan i. Since the approach is not dependent on the order of actions within a plan, counters can be kept per action set instead of for each plan to reduce the number of counters. For example consider two plans $p_1 =< a_A, a_B, a_C >$ and $p_2 =< a_C, a_B, a_A >$ by the same counter $c_{f,s}$ for action set $s =$ {A,B,C}. Let S be the set of all such sets, which have failed at least once. The first term is then:

$$\begin{aligned} &\textstyle\prod_{P \text{ s.t. } \mathcal{Y}(P)=1} [1 - \prod_{a \in bad(D) \cap P}(1 - Pr_b(a))] = \\ &\textstyle\prod_{s \in S} [1 - \prod_{a \in bad(D) \cap s}(1 - Pr_b(a))]^{c_{f,s}} \end{aligned} \tag{5.23}$$

Notice that it is not required to store the action sets of successful itineraries (by far the dominant case).

5.5.1 Learning the Intermittency Rate

As in the single fault case, the intermittency parameters of action failure q_i can be learned. In practice, it could be a single scalar (assuming that all the actions have the same intermittency parameter) or a vector (allowing actions to have different intermittency rates). This section presents a general methodology for the estimation of the intermittency rate q.

The goal of learning is to estimate the value of q to best match the observation Y. To achieve this goal, q is treated as a deterministic unknown parameter, and formulate the learning problem as a maximum-likelihood estimation problem:

$$\hat{q}_{ML} = arg\max_{q} Pr_q(Y). \tag{5.24}$$

Here Y is the observation history, i.e., the plans and their corresponding execution observations. To evaluate $Pr_q(Y)$ the following can be used:

$$Pr_q(Y) = \sum_{D} Pr_q(Y|D)Pr(D) \tag{5.25}$$

where $Pr(D)$ is the prior probability, initially all equal for all hypotheses. The observation likelihood $Pr_q(Y|D)$ is the $Pr(Y|D,P)$ given by (5.20) in the previous section; the plan P is known, hence it is removed to simplify notation. Given any intermittency parameter q (which equivalently specifies all the actions mis-operation probability $Pr_b(a)$), the observation likelihood $Pr_q(Y)$ can be evaluated by the marginalization (5.25). The optimal estimate is then obtained by search over the space for maximal $Pr_q(Y)$.

Example. Assume all faulty actions have the same intermittency parameter, i.e., $Pr_b(a) = q$ for all action a. In this case, given any itinerary, the probability of observing a success or a failure is:

$$Pr_q(Y|D,P) = \begin{cases} 1-(1-q)^{n(D,P)} & \text{if Fail} \\ (1-q)^{n(D,P)} & \text{if Success} \end{cases} \tag{5.26}$$

where the exponent $n(D,P) \stackrel{def}{=} |bad(D) \cap P|$ denotes the number of bad actions in the hypothesis D that are involved in the plan P. For any given D and P, $n(D,P)$ is easy to evaluate. This enables us to express $Pr_q(Y)$ as simple polynomial function of q. This can be used to search for the optimal $q \in [0,1]$.

Figure 5.7 shows the learning for a simple system consisting of five actions, among which two actions have faults with an intermittency rate of 0.2. Learning is done based on 100, 200, and 500 randomly simulated trials; the results are shown as the green, blue, and red curves respectively. The curves plot the computed observation likelihood $\ln Pr_q(Y)$ as a function of q. The maximum likelihood estimates are marked with circles. With more trials shows, the estimated q is closer to the underlying true value. For example, with 500 trials, $\hat{q}_{ML} = 0.19$ (ground truth is 0.2). As more trials are incorporated, the likelihood $\ln Pr_q(Y)$ has a more prominent optimal q estimate. This is expected.

This algorithm computes $Pr(D)$ and q simultaneously and converges rapidly. The generalization to the situation where all q's are different requires a multi-dimensional optimization.

5.6 Conclusions

This chapter lays out a framework for continuously diagnosing any combination of persistent and intermittent faults. Furthermore, an extension has been introduced to simultaneously learn

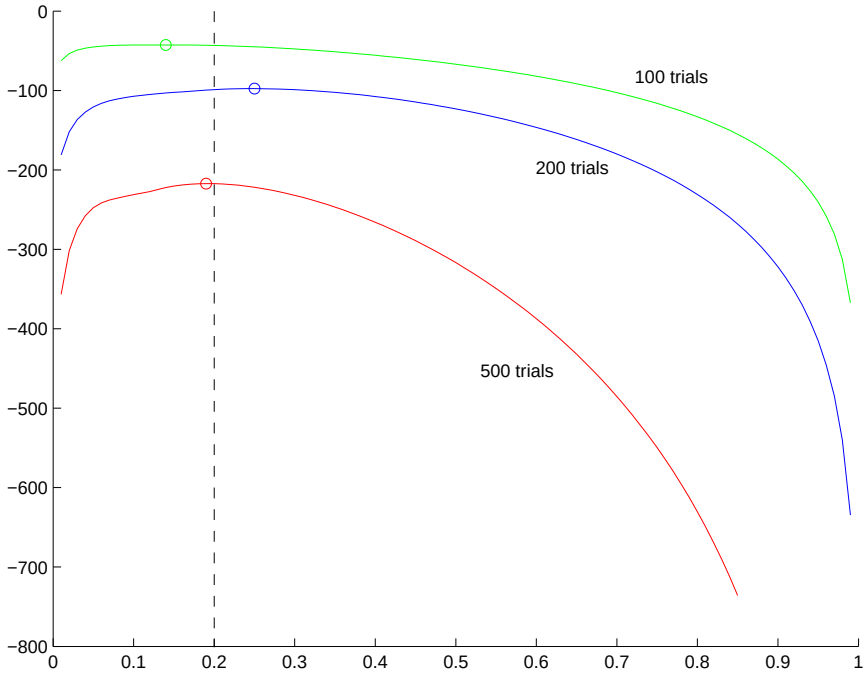


FIGURE 5.7 Learning of intermittency q. ($\ln Pr_q(Y)$ vs. q)

the intermittency rate q while compute the posterior probabilities. This approach demonstrates that it is possible to update the probabilities and learn the intermittency rate without recording all past itineraries. The key idea of the proposed framework is based on maintaining a limited set of counters instead of the entire itinerary history without losing valuable diagnosis information. Table 5.4 shows the granularity of the counters needed to store the entire diagnostic history. Note: the number of actions $|A|$ is much smaller than the number of action sets $|S|$ and the number of action sets $|S|$ is much smaller than the number of plans $|P|$. This extension

TABLE 5.4 Table shows the granularity of counters needed to store the entire diagnostic history.

	c_s	c_f
single fault	per action	per action
multiple fault	per action	per action set

to model-based diagnosis enables on-line diagnosis to modular planning agents, e.g. reprographic equipment. More importantly, it extends model-based diagnosis to the real challenges, such as efficient diagnosis of intermittent multiple faults, faced in diagnosing manufacturing plants, packaging equipment, laboratory test equipment, etc.

CHAPTER 6

Diagnosis with incomplete Models: Diagnosing Hidden Interaction Faults

This chapter extends model-based diagnosis (MBD) (de Kleer & Williams 1987, Reiter 1987) to systems with hidden interaction faults. An interaction fault is present if an interaction among a set of components leads to an observable failure, even though each individual component individually meets the specifications. A naive approach to address interaction faults is to simply account for all possible interaction faults in the system model. However, the naive approach presumes that all possible faults, both component and interaction faults, are known and addressed in the model. This assumption is violated by most real world systems, such as shorts in circuits (Davis 1984) or unmodeled connections (de Kleer 2007*b*). That leads to incomplete system models, hence possibly hidden interaction faults. The problem of hidden interactions has been known for a long time (Davis 1984), but until now no general solution has been proposed. Instead of pushing for complete models (Preist & Welham 1990) or relying on additional structural information (Davis 1984, Bottcher 1995, de Kleer 2007*b*) the proposed framework approaches the challenge differently. System models are allowed to be incomplete and a general, domain independent extension is introduced to model-based diagnosis to account for resulting hidden interaction faults. This extends model-based diagnosis to systems with incomplete models, in particular to models with incomplete structural information. In the chapter, the proposed diagnosis framework is demonstrated on a logic circuit with a hidden interaction fault, before it is outlined how this framework can be used in the context of action driven systems.

6.1 Introduction

Model-based diagnosis assumes that all necessary information, regarding all possible failure causes, is available in the system model. In our experience, this generally accepted assumption does not hold in practice. In reality, systems fail for all kind of reasons, some of which designers might not be able to predict at the time the system model is built. This leads to incomplete models and to possible hidden interaction faults. For example, during the landing maneuver of the Mars Polar Lander an interaction between a touch sensor and the deployment of one of the Lander's legs most likely caused the mission to fail (Young, Arnold, Brackey, Carr, Dwoyer, Fogleman, Jacobson, Kottler, Lyman & Maguire 2000). The deployment of the leg caused the touch sensor to produce a noise spike which was incorrectly classified as an in-

dication of touch down. As a consequence, the lander shut off its thrust about 40 meters above the touch-down surface. This is a classic example of a failure caused by a hidden interaction. If the engineers could have predicted this interaction, the failure could have been avoided. The classification algorithm could have requested either additional information from the altitude sensor onboard the lander or a time persistent signal from the touch sensor. Building adequate models for increasingly complex systems, especially for embedded systems, is very difficult; building complete models is practically impossible. In practice most models are incomplete; especially when all possible interactions are not known at the time the system is built.

Unlike a behavior model, which might describe the behavior only partially, e.g. weak fault model (describes only nominal behavior), the structural model is usually assumed to be complete (Davis 1984, Preist & Welham 1990, Bottcher 1995, de Kleer 2007*b*). An incomplete system topology, e.g. a model that doesn't capture all connections, causes standard diagnosis frameworks to result in an irresolvable contradiction.

Instead of pushing for complete models, the challenge is approached differently. Models are allowed to be incomplete and a diagnosis framework is introduced that works with incomplete models, extending model-based diagnosis to systems with hidden interactions. The framework accounts for interaction faults without explicitly modeling them. The resulting approach enables diagnosis for systems with multiple, interaction faults.

The chapter is organized as follows: The first section reviews related work. A logic circuit, *SMALLY*, is introduced which serves as our example system to illustrate the limitations of standard model-based diagnosis (Section 3.2.1). Then, interaction faults are formally defined and a general extension to model-based diagnosis is introduced to account for hidden interaction faults. The last section outlines how this framework can be applied to action driven systems, such as planning agents.

6.2 Related Work

The general mechanisms of inferring health states from observations have a long history in artificial intelligence and engineering including logic based frameworks (Reiter 1992), continuous non-linear systems (Rauch 1995), xerographic systems (Zhong & Li 2000), and hybrid logical probabilistic diagnosis (Poole 1991).

The process of diagnosis can be viewed as the interaction between observations and predictions. Observations capture the actual system behavior, whereas predictions are deduced from the system model. Model-based diagnosis presumes a system failure to be present if predictions and observations differ from each other.

Model-based approaches (de Kleer & Williams 1987, Reiter 1987) predict component interaction only where these are explicitly provided in the system description. The problem of faults caused by hidden interactions has been known since (Davis 1984). In (Davis 1984) bridge faults between adjacent components are introduces, but the suggested solution requires explicit knowledge about which unintended connections potentially result from adjacent components. In (Preist & Welham 1990) a solution, similar to the naive approach, is proposed which explicitly models all possible unintended interactions. The argument that a complete model is preferable over an incomplete model is true, but note that a complete model might

not always be available. In (Bottcher 1995) the work of (Davis 1984) is generalized by introducing a notion of neighbors that requires information about spatial proximity among components. A literature review suggests that there exists no other diagnosis framework that accounts for hidden interaction faults by a general, domain independent extension without relying on additional domain dependent knowledge. All approaches listed in the related work section, (Davis 1984, Preist & Welham 1990, Bottcher 1995, de Kleer 2007*b*) assume that additional knowledge regarding potential hidden interaction is available. This chapter introduces an approach to diagnosing hidden interactions that does not rely on any kind of additional information such as knowledge about potential unintended connections or spatial proximity among components.

6.3 Limitations of Model-based Diagnosis

Model-based diagnosis, as reviewed in Section 3.2.1, has a long history in artificial intelligence. This section outlines limitations of model-based diagnosis in the context of hidden interaction faults. Consider the logic circuit, *SMALLY*, illustrated in Figure 3.4. Assume observation obs_1 is collected, which involves both components, A and C, for example by observing a, b, c, and f,

$$obs_1 \quad = \quad [a \equiv 1 \wedge b \equiv 1 \wedge c \equiv 1] \rightarrow f \equiv 1. \tag{6.1}$$

Given observation obs_1 and the system description SD, it can evaluated if the predicted behavior is consistent with what was observed. In the presented example the predicted behavior is not consistent with the actual observation obs_1. The system description SD together with $\neg AB^*$ imply that $a \equiv b \equiv c \equiv 1$ that $d \equiv 1$, $e \equiv 1$, and $f \equiv 0$. The predicted value for f is therefore 0, but the actually observed value is 1. Based on system description SD and observation obs_1 a conflict can be inferred. The resulting conflict in our example is

$$SD \cup \{obs_1\} \vdash AB(A) \vee AB(B) \vee AB(C). \tag{6.2}$$

The diagnosis task is to find health assignments that make SD and OBS consistent. The list in Equation 6.3 shows all valid diagnoses based on observation obs_1 ordered by cardinality.

$$\begin{aligned}
&\textit{single fault diagnoses:}\\
&\Delta_1 = \{A\}, \qquad \Delta_2 = \{B\}, \qquad \Delta_3 = \{C\},\\
&\textit{double fault diagnoses:}\\
&\Delta_4 = \{A,B\}, \qquad \Delta_5 = \{A,C\}, \quad \Delta_6 = \{B,C\},\\
&\textit{triple fault diagnoses:}\\
&\Delta_7 = \{A,B,C\}
\end{aligned} \tag{6.3}$$

The set of diagnoses can be reduced to the set of minimal cardinality diagnoses, defined in Definition 14. The list in Equation 6.4 shows the set of minimal cardinality diagnoses. The minimal cardinality among all diagnoses is 1 yet it can not be concluded that there is only one failure in the system. The only conclusion that can be drawn is that there is at least one failure in the system.

$$\begin{aligned} &\textit{single fault diagnoses:} \\ &\Delta_1 = \{A\}, \quad \Delta_2 = \{B\}, \quad \Delta_3 = \{C\} \end{aligned} \tag{6.4}$$

Assume another observation obs_2 is collected:

$$obs_2 = [a \equiv 1 \wedge b \equiv 1] \rightarrow d \equiv 1. \tag{6.5}$$

Based on the two observations collected, obs_1 Equation 6.1 and obs_2 Equation 6.5, it can be deduce that a fault in component A individually can not explain the discrepancy. Recall, non-intermittent faults are assumed. This reduces the set of minimal cardinality diagnoses to the list:

$$\begin{aligned} &\textit{single fault diagnoses:} \\ &\Delta_2 = \{B\}, \quad \Delta_3 = \{C\}, \end{aligned} \tag{6.6}$$

To illustrate the limitations of standard diagnosis frameworks, assume that another two observations obs_3 and obs_4 are collected, shown in Equation 6.8.

$$\begin{aligned} obs_3 &= [d \equiv 1 \wedge c \equiv 1] \rightarrow e \equiv 1 \\ obs_4 &= [e \equiv 1] \rightarrow f \equiv 0 \end{aligned} \tag{6.7}$$

Based on the observations obs_1, obs_2, and obs_3, it can be concluded that neither component A individually nor component B individually can explain the discrepancy. Once the reasoning is expanded to include all available observations, obs_1, obs_2, obs_3, and obs_4, the diagnosis framework results with an irresolvable contradiction. But why is that? Lets take a closer look at the observations, obs_1, obs_2, obs_3, and obs_4. The observations can be re-written as shown in Equation 6.9.

$$\begin{aligned} obs_2 &= [a \equiv 1 \wedge b \equiv 1] \rightarrow d \equiv 1 \\ obs_3 &= [d \equiv 1 \wedge c \equiv 1] \rightarrow e \equiv 1 \\ [obs_2 \wedge obs_3] &\rightarrow [a \equiv 1 \wedge b \equiv 1 \wedge c \equiv 1] \rightarrow e \equiv 1 \\ obs_4 &= [e \equiv 1] \rightarrow f \equiv 0 \\ [obs_2 \wedge obs_3 \wedge obs_4] &\rightarrow [a \equiv 1 \wedge b \equiv 1 \wedge c \equiv 1] \rightarrow f \equiv 0 \\ obs_1 &= [a \equiv 1 \wedge b \equiv 1 \wedge c \equiv 1] \rightarrow f \equiv 1 \end{aligned} \tag{6.8}$$

As a result, it can now be seen that observation obs_1 is in an irresolvable contradiction to the observations obs_2, obs_3, and obs_4. This inconsistency is independent of the chosen health assignment. As there exists no health assignment that makes the observations consistent, it can

be inferred that there exists no diagnosis for the system. Generally, the existence of a diagnosis is defined as:

Definition 18. *A diagnosis exists for* $(SD, COMPS, OBS)$ *if and only if* $SD \cup OBS$ *is consistent.*

Standard model-based diagnosis will terminate with an irresolvable contradiction, if no diagnosis can be found. Formally, from Definition 18 it follows that there exists no diagnosis for $(SD, COMPS, OBS)$ if and only if $SD \cup OBS$ is inconsistent. Hence, in our example, no diagnosis can be found, and standard model-based diagnosis terminates with an error.

The limitations are caused by the assumption that an accurate and complete model is available. In a real world scenario this is rather impractical. Building models is a time demanding, expensive process. Therefore, most system models are limited to enable detection or isolation of a small pre-defined set of failures. Typically, a system model captures only the knowledge required to diagnosis this pre-defined set of failures and abstracts all other information away.

For example, the automobile industry adopted on-board diagnosis for cars, but only targeted to specific subsystems and failure modes. Some cars have the capability to diagnosis if a head light bulb is out, but fail if the connecting cable is broken. A broken cable occurs so infrequently, that most diagnosis designer neglect that a cable might break and abstract it away (de Kleer 2007*b*). If the cable does break, the initial diagnosis might suggest that one of the two connected components is faulted. Once the two components are individually tested without noticeable abnormality the diagnosis framework either incorrectly concludes an intermittent fault in one of the two components (if the framework is aware of this fault type) or terminates with an irresolvable error. The irresolvable error results from the unawareness of the connection. The connection is hidden to the diagnosis framework and all other components are exonerated as fault candidates. The diagnoser results with an empty list of diagnosis candidates, yet has observed a discrepancy. This results in an irresolvable contradiction.

Another reason for incomplete system models is due to model recycling, the act of reusing an already existing model. Building models is an expensive and time demanding task, which makes model recycling attractive. Typically, there are two kinds of sources for reusable models: Either there exists a similar system, similar enough to adapted its model or there exists a model for the target system which was originally built for a different task, e.g. system based on a planning (Kuhn et al. 2008*b*) or scheduling (Muscettola et al. 1998) model.

The set of failures desired to be diagnosable as well as the intended repairs influence the scope and abstraction level of the resulting system model. For example, a vendor that only performs repair by exchanging entire subsystems might neglect fault isolation on the component level as it is not necessary for the repair. This leads to abstract system models targeted towards a specific diagnosis task. In our car example, diagnosis is targeted to find the most common failures (e.g. broken light bulb), but results with an irresolvable contradiction if a component outside of the model scope causes the abnormality, e.g. a broken cable. Such models violate the no-function-in-structure principle.

6.4 Model-based Diagnosis with Interaction Faults

This section proposes a diagnosis framework that is able to diagnose component faults as well as hidden interaction faults. The approach builds on the assumption, that there is no additional knowledge besides what is already captured by the model. Hence all available knowledge is already built into the model, yet the model might still be incomplete. This leads to a new fault type: faults caused by hidden interactions, coined interaction faults. An interaction fault is present, if an interaction among a set of components leads to an observable failure, even though each individual component individually meets the specification. Hidden interactions can lead to interaction faults. Hidden interaction faults can be characterized into the following two groups of hidden interactions:

- A *hidden component interaction* is present if a set of known components interact through a hidden component. The component is hidden in the sense that it does not appear in the model. A common example for hidden components are connections in circuits (de Kleer 2007*b*). Most system models abstract connections away. Typically, a fault in a connection initially results in the belief that one of the two connected components is faulted. Once the two components are individually tested without noticeable abnormality the diagnosis framework either incorrectly concludes an intermittent fault (if the framework is aware of this fault type) or terminates with an irresolvable contradiction.

- A *hidden behavior interaction* is present if and only if the interaction between a set of components leads to unpredicted behavior. Consider a food processing line for candy bars. There are multiple components wrapping and boxing candy bars. It may be that component A leaves a tiny rip which is of no consequence for the consumer, but boxing component B has a small protrusion such that the rip sometimes catches and destroys the candy bar. Such faults are called hidden behavior interaction faults: A and B are perfectly operational individually but will not work correctly if A and B operate together. Hidden behavior faults also occur in circuits: Two gates A and B may not work well together as the accumulated delay leads to a failure. Testing both components individually might convey that both are late, yet within specification. (Some might call this bad design, but most complex systems have design errors.)

The following section introduces a diagnosis framework for systems containing interaction faults. First, such systems are defined and an extension to standard model-based diagnosis is discussed such that the diagnosis framework accounts for both individual component faults as well as hidden interaction faults. The extension is generally applicable without additional system knowledge. Definition 19 defines a model-based diagnosis system with hidden interactions:

Definition 19. *A model-based diagnosis system with hidden interactions is represented as a quadruple*
$(SD+, COMPS, OBS+, SCOPE)$ *where:*

- $SD+$*, extended system description, is a set of first-order sentences,*

- $COMPS$, *components, is a set of constants,*
- $OBS+$, *extended observations, is a set of first-order sentences,*
- $SCOPE$, *scope of an observation, is a function mapping an observation onto a subset of* $COMPS$.

The extended system description $SD+$ extends the standard system description SD, defined in Definition 5, to account for potential hidden interactions. Similar to before, the $SD+$ contains knowledge about the behavior and the structure of the system, but admits the possibility of a hidden interaction. Related approaches (Davis 1984, Bottcher 1995, de Kleer 2007*b*) have suggested to extend the SD by explicitly modeling potential hidden interactions by using additional knowledge about the system. The suggested extension is domain independent, hence any system is extended with the same extension. This enables the approach to be widely applicable, even if potential hidden interactions are not known at design time. Before the extension is introduced, the definition of AB-literals needs to be modified. The modified literals are denoted as AB^i-literals:

Definition 20. *Let* $Pow(COMPS)$ *be the set of all subsets of* $COMPS$ *such that*

$$Pow(COMPS) = \{pc \mid pc \subseteq COMPS\}\,, \tag{6.9}$$

let $pc \in Pow(COMPS)$ *indicate an interaction among all components in* pc *and let all* $AB^i(x) = AB(x)$. *An* AB^i*-literal indicates the health of* $pc \in Pow(COMPS)$ *and can be either* $AB(pc)$ *or* $\neg AB(pc)$, *where* $AB(pc)$ *represents that* pc *is ABnormal (faulted) and* $\neg AB(pc)$ *indicates that* pc *is not ABnormal, thus behaving normal.*

Based on the standard system description SD, the extended system description $SD+$ is simply constructed by first adding the model extension ME shown in Equation 6.10 and secondly by replacing all AB-literals with the corresponding single component AB^i-literals.

$$ME = \bigcup_{pc \in Pow(COMPS)} AB^i(pc) \rightarrow \left[\bigwedge_{pc' \subset pc} \neg AB^i(pc') \right] \tag{6.10}$$

The extension semantically adds two aspects to a system description. First, it introduces AB^i-literals for all pc's with higher cardinality and second it introduces the relation among individual AB^i-literals. AB^i-literals for higher cardinality pc's account for unmodeled interactions which might occur between components $c \in pc$. This guides the diagnosis framework to detect and isolate abnormalities even if they are caused by hidden interactions. In the case some pc is diagnosed to be abnormal, sentence 6.10 enforces that all subsumed pc''s are diagnosed to be not abnormal. The second aspect is important as it only makes sense to hypothesize about an interaction fault if all hypotheses of subsumed individual component faults as well as interaction faults are exonerated.

The extended system description for our example *SMALLY* can be formalized as shown in Equation 6.11. The hidden interaction is indicated as a dashed connection in Figure 3.4.

$$
\begin{aligned}
SD &= CL \cup ST \cup ME \text{ where} \\
CL &= \{And(x) \rightarrow \\
&\quad [\neg AB(\{x\}) \rightarrow [in(x,1) \wedge in(x,2) \equiv out(x)]] \\
&\quad Inv(x) \rightarrow [\neg AB(\{x\}) \rightarrow [in(x,1) \equiv \neg out(x)]]\} \\
ST &= \{And(A) \wedge And(B) \wedge Inv(C), \\
&\quad a \equiv in(A,1) \wedge b \equiv in(A,2) \wedge out(A) \equiv d, \\
&\quad d \equiv in(B,1) \wedge c \equiv in(B,2) \wedge out(B) \equiv e, \\
&\quad e \equiv in(C,1) \wedge out(C) \equiv f\} \\
ME &= \{AB^i(\{A,B\}) \rightarrow [\neg AB(\{A\}) \wedge \neg AB(\{B\})], \\
&\quad AB^i(\{A,C\}) \rightarrow [\neg AB(\{A\}) \wedge \neg AB(\{C\})], \\
&\quad AB^i(\{B,C\}) \rightarrow [\neg AB(\{B\}) \wedge \neg AB(\{C\})], \\
&\quad AB^i(\{A,B,C\}) \rightarrow \\
&\quad [\neg AB(\{A\}) \wedge \neg AB(\{B\}) \wedge \neg AB(\{C\}) \wedge \\
&\quad \neg AB(\{A,B\}) \wedge \neg AB(\{A,C\}) \wedge \neg AB(\{B,C\})]\}
\end{aligned}
\tag{6.11}
$$

The extended $SD+$ does not define any relations between possible observations and interaction faults. A good technician can infer interaction faults from observations. Consider a technician tests two components individually and observes no abnormality, but if both components are tested together the observations indicate an abnormality. The technician would draw the conclusion that there might exist a hidden interaction. To enable a diagnosis framework to perform the same kind of inference, two things have to be defined: First, the scope of an observation has to be defined, basically what is being tested together. Second, the scope has to be incorporated into the observations, to indicate which observation is relevant to which interaction faults. Given an observation, the concept of an observation scope defines the set of components that has potentially impacted this observation. Once the function $SCOPE$ is defined, a set of observations OBS can be extended to a set of extended observations $OBS+$ according to:

$$
OBS+ = \left\{ [obs] \vee \bigvee_{pc \subseteq Scope(obs)} AB^i(pc) \mid |pc| > 1, obs \in OBS \right\} \tag{6.12}
$$

Generally, the function $SCOPE$ can be extracted from the system description without relying on additional information. An observation is a set of measurement points. The system structure combined with the component behavior provides information in order to determine, which set of components has potential impact on, which set of measurement points. The scope of an observation can be extracted by backward reasoning. In the example, the scope of an ob-

servation is informally defined as the components that had potentially impacted the resulting measurement point. For example an observation measuring at point a and e scopes over component A and B, an observation measuring at point a, b, and f scopes over the components A, B, C and an observation measuring only a and b scopes over no component as the signal neither travels from a to b nor from b to a. The scope for the observations in our example are illustrated in Listing 6.14.

$$\begin{aligned} SCOPE(obs_1) &= \{A, B, C\} \\ SCOPE(obs_2) &= \{A\} \\ SCOPE(obs_3) &= \{B\} \\ SCOPE(obs_4) &= \{C\} \end{aligned} \tag{6.13}$$

Given the observation scopes the extended observation can be constructed based the extension rule shown in Equation 6.12. The resulting extended observations are shown in Listing 6.15.

$$\begin{aligned} obs_1^i &= [obs_1] \vee AB^i(\{A, B\}) \vee AB^i(\{A, C\}) \vee \\ &\quad AB^i(\{B, C\}) \vee AB^i(\{A, B, C\}) \\ obs_2^i &= obs_2 \\ obs_3^i &= obs_3 \\ obs_4^i &= obs_4 \end{aligned} \tag{6.14}$$

Until now the system definition has been extended and AB^i-literals have been introduced with the intent to diagnose hidden interaction faults. Definition 11 defines a diagnosis as set of components assigned to be abnormal such that the resulting assignment over all components makes the system description consistent with the observations. In this definition an assignment is limited to determine which individual component is considered to be abnormal or not abnormal. In order to account not only for individual components, but also for hidden interactions, the assignment is expanded to be over all AB^i-literals. Individual components are captured by AB^i-literals with cardinality 1 and hidden interactions are addressed by AB^i-literals with higher cardinality. A diagnosis is an assignment of abnormal or not abnormal to all elements of $Pow(COMPS)$ describing one possible health state of the system. Formally, a diagnosis for systems with hidden interactions, called an interaction diagnosis, is defined in Definition 22.

Definition 21. *Given two sets C_{AB}, $C_{\neg AB} \subseteq Pow(COMPS)$, $D^i(C_{AB}, C_{\neg AB})$ is defined to be the conjunction:*

$$\left[\bigwedge_{pc \in C_{AB}} AB_i(pc)\right] \wedge \left[\bigwedge_{pc \in C_{\neg AB}} \neg AB_i(pc)\right] \tag{6.15}$$

where $AB^i(x)$ corresponds to the AB^i-literal of x.

Definition 22. *An interaction diagnosis Δ^i for $(SD+, COMPS, OBS+, SCOPE)$ is a sub-*

set of $Pow(COMPS)$, such that the following set of sentences is satisfiable

$$SD \cup OBS \cup \left\{D^i(\Delta, Pow(COMPS) - \Delta)\right\} \tag{6.16}$$

The Listing 6.17 shows all valid interaction diagnoses for $SMALLY$ given that only observation obs_1^i was observed.

$$\begin{aligned}
&\textit{single fault interaction diagnoses:}\\
&\Delta_1^i = \{\{A\}\}, \quad \Delta_2^i = \{\{B\}\},\\
&\Delta_3^i = \{\{C\}\}, \quad \Delta_4^i = \{\{A,B\}\},\\
&\Delta_5^i = \{\{A,C\}\}, \quad \Delta_6^i = \{\{B,C\}\},\\
&\Delta_7^i = \{\{A,B,C\}\}\\
&\textit{double fault interaction diagnoses:}\\
&\Delta_8^i = \{\{A\},\{B\}\}, \quad \Delta_9^i = \{\{A\},\{C\}\},\\
&\Delta_{10}^i = \{\{B\},\{C\}\}, \quad \Delta_{11}^i = \{\{A\},\{B,C\}\},\\
&\Delta_{12}^i = \{\{A,B\},\{C\}\}, \quad \Delta_{13}^i = \{\{B\},\{A,C\}\}\\
&\textit{triple fault interaction diagnoses:}\\
&\Delta_{14}^i = \{\{A\},\{B\},\{C\}\}
\end{aligned} \tag{6.17}$$

Similar to Definition 14, the set of interaction diagnoses can be reduced by adapting the concept of minimal cardinality interaction diagnoses, as illustrated in Definition 23.

Definition 23. *An interaction diagnosis Δ_x^i for $(SD+, COMPS, OBS+, SCOPE)$ is minimal in cardinality if and only if there exists no other interaction diagnosis Δ_y^i such that $|\Delta_y^i| < |\Delta_x^i|$.*

The minimal cardinality interaction diagnoses resulting from observation obs_1^i are all single fault diagnoses in Listing 23. Further the set can be reduced by an even more strict definition of minimal, coined a minimal cardinality, minimal interaction diagnosis.

Definition 24. *A minimal cardinality diagnosis Δ_x^i for $(SD+, COMPS, OBS+, SCOPE)$ is also a minimal cardinality, minimal interaction diagnosis if and only if there exists no other diagnosis Δ_y^i such that an element in $|\Delta_y^i|$ is a strict subset of any element in $|\Delta_x^i|$.*

The resulting set of minimal cardinality, minimal interaction diagnoses, given that observation obs^i was observed, is illustrated in Listing 6.18.

$$\begin{aligned}
&\textit{minimal cardinality, minimal interaction diagnoses:}\\
&\Delta_1^i = \{\{A\}\}, \quad \Delta_2^i = \{\{B\}\}, \quad \Delta_3^i = \{\{C\}\}
\end{aligned} \tag{6.18}$$

Listing 6.18 shows the set of minimal cardinality, minimal interaction diagnoses assuming only observation obs_1^i is available. Consider observation obs_2^i is included such that the resulting set of interaction diagnoses has to be consistent with both observations, obs_1^i and obs_2^i. It can be deduced that a fault in component A individually can not explain the discrepancy. Therefore, the resulting set of minimal cardinality, minimal interaction diagnoses reduces to the once shown in Listing 6.19

$$\begin{aligned} &\textit{minimal cardinality, minimal interaction diagnoses:} \\ &\Delta_2^i = \ \{\{B\}\}, \ \ \Delta_3^i = \ \{\{C\}\} \end{aligned} \tag{6.19}$$

Consider now that observations obs_3^i is included. From all three observations, it can be deduced that a single fault in component A as well as a single fault in component B can not explain the discrepancy. The only explanation for the discrepancy is that there exists either a single fault in component C, an interaction fault or a multiple fault. Restricting the set of diagnoses to the set of minimal cardinality interaction diagnoses implies that multiple fault diagnoses are only be considered if all single faults are exonerated. This leaves only the hypotheses of a single fault in component C or single interaction faults as valid diagnoses. The remaining set of diagnoses is shown in Listing 6.20, in more detail.

$$\begin{aligned} &\textit{minimal cardinality interaction diagnoses:} \\ &\Delta_3^i = \ \{\{C\}\}, \qquad \Delta_4^i = \ \{\{A,B\}\}, \\ &\Delta_5^i = \ \{\{A,C\}\}, \qquad \Delta_6^i = \ \{\{B,C\}\}, \\ &\Delta_7^i = \ \{\{A,B,C\}\} \end{aligned} \tag{6.20}$$

The diagnoses in Listing 6.20 are all minimal cardinality interaction diagnoses according to Definition 23, yet not all of them are also minimal cardinality, minimal interaction diagnosis. According to Definition 23 and the fact that diagnosis Δ_3^i is a valid diagnosis it can concluded that diagnoses Δ_5^i, Δ_6^i, and Δ_7^i are not considered to be minimal cardinality, minimal interaction diagnoses. All three contain at least one element x, such that $\{C\} \subset x$. The resulting set of minimal cardinality, minimal interaction diagnoses is shown in Listing 6.21.

$$\begin{aligned} &\textit{minimal cardinality, minimal interaction diagnoses:} \\ &\Delta_3^i = \ \{\{C\}\}, \ \ \Delta_4^i = \ \{\{A,B\}\} \end{aligned} \tag{6.21}$$

Considering all four observations $obs_1^i, obs_2^i, obs_3^i, obs_4^i$, it can be deduced that a single fault in component C can not explain the observations either. Diagnosis Δ_3^i is not longer a valid diagnosis. At this point the standard model-based diagnosis framework terminates with an irresolvable contradiction. The proposed framework, however, generates the set of minimal cardinality, minimal interaction diagnosis shown in Listing 6.22.

minimal cardinality, minimal interaction diagnoses:

$$\Delta_4^i = \{\{A,B\}\}, \quad \Delta_5^i = \{\{A,C\}\}, \quad \Delta_6^i = \{\{B,C\}\} \tag{6.22}$$

Our framework he does not result with an irresolvable contradiction if and only if at least one of the AB^i-literals shown in Listing 6.23 is assigned to $abnormal$. By assigning one of the interaction AB^i-literals to $abnormal$ observation obs_1^i evaluates independently of the assignment to all other AB^i-literals without conflict.

$$AB^i(\{A,B\}) \vee AB^i(\{A,C\}) \vee AB^i(\{B,C\}) \vee AB^i(\{A,B,C\}) \tag{6.23}$$

Definition 25. *A system is diagnosed to contain multiple faults if and only if a minimal cardinality diagnosis Δ^i contains more than one element.*

Definition 26. *A system is diagnosed to contain an interaction fault if and only if a minimal cardinality, minimal interaction diagnosis Δ^i contains an element $x \in \Delta$ with more than one component.*

6.5 Conclusions

This chapter has proposed a fundamentally new approach to address the very real issue that most system models are incomplete. Ensuring complete models is practically impossible. Through introducing interaction literals most kinds of unintended interactions can be accommodated within the model-based diagnosis framework. One of the main motivations behind this work arose from developing diagnostic algorithms for Xerographic equipment. Interaction faults are surprisingly common and are difficult for technicians to diagnose. They are also difficult to self-diagnose (more and more equipment includes self-diagnosis).

This chapter has lays out a fundamental approach to hidden interaction faults. As with all model-based frameworks, it is computationally explosive if directly implemented as described in the definitions of this chapter. An efficient implementation is outline in Chapter 4, called Tiered-Partitioned Inference, which introduces both $AB(x)$ and $AB^i(x)$ literals only when needed, i.e., extends ME only when needed. A direct translation of Equation 6.10 to all interaction faults of cardinality n leads to potentially $|COMPS|^n$ clauses.

CHAPTER 7

Target-Value-Search

This chapter defines a new class of combinatorial search problems, called Target-Value Search Problem, in which the objective is to find the path or set of paths between two given vertices in a graph whose length/value is as close as possible to some *target-value*. Unfortunately, traditional heuristic search methods developed for shortest path search cannot be directly applied due to the lack of optimal substructures in target-value problems. Target-value problems are a generalization of shortest path problems. Target-value Search Problems can be decomposed into simpler searches using an interval-valued heuristic. Given the decomposition the problem can then be solved with efficient A^* searches. Further it is shown that switching from a single-interval to a multi-interval heuristic further improves performance. Target-value problems arise in a variety of domains including planning for activity recommendation and the integration of automated planning and diagnosis. This chapter illustrate the problem and theoretical findings experimentally.

7.1 Introduction

This Chapter addresses the problem of finding a path that achieves the goal, while coming as close as possible to a target length. These types of problem arise in a variety of domains. In consumer recommendation domains people often have approximate targets (Winter 2002) that they would like to get as close to as possible. For instance, one might want to hike in a park from some parking lot to some mountain cabin in about 3 hours. Within the set of hikes that reach the cabin, hikes much shorter than 3 hours would be too easy and hikes much longer than 3 hours would be too tiring.

Target-value search is not limited to problems involving distance. In diagnosis, a technician would like to select a valid action sequence that is most informative about why the system is failing, see Chapter 3. Of course, if the sequence is invalid, it will not execute at all. Within the set of valid sequences, the most informative sequence will be the one whose probability of failing comes as close as possible to some target(-value) failure probability mass, e.g. $T = 0.5$ in the persistent failure case. A sequence that succeeds too often, or fails too often, reveals less information, see Section 3.5.1.

The chapter is organized as follows: In the next section the Target-value Search Problem is formally defined and a decomposition of the Target-value Search Problem is introduced such that solutions can be found by leveraging A^* search (Hart et al. 1968). The key idea of the presented approach is a novel method of calculating an interval-valued heuristic. Finally, the last

section evaluates the presented heuristic search on a set of (synthetic and real-world) problems from the diagnosis domain designed to explore the performance of this new techniques.

7.2 Target-value Search Problem

Let $G = (\mathcal{V}_G, \mathcal{E}_G)$ be a finite, directed acyclic graph (DAG), with a set of vertices $\mathcal{V}_G$ and a set of positive valued edges $\mathcal{E}_G$. Let $p = \langle v_0, ..., v_n \rangle$ be a path where $v_i \in \mathcal{V}_G$ and neighboring vertices are connected by edges $(v_i, v_{i+1}) \in \mathcal{E}_G$. Let the path value, $g(p)$, be the sum of the edge values in p.

Definition 27. *Let $p_{v_0 \to v_g}$ be a path from vertex v_0 to v_g and $\mathcal{P}_{v_0 \to v_g}$ be the set of all paths from v_0 to v_g. A target-value path $p^T_{v_0 \to v_g}$ w.r.t. v_0, v_g and target-value T is a path from v_0 to v_g with minimal deviation between its path value and the specified target-value T:*

$$p^T_{v_0 \to v_g} \in \operatorname{argmin}_{p \in \mathcal{P}_{v_0 \to v_g}} \mid g(p) - T \mid. \tag{7.1}$$

7.3 Conventions

For clarity and brevity, the discussion is limited to *connection graphs*. A *connection graph* $C^G_{v_0 \to v_g}$ is defined of two vertices $v_0, v_g \in G$ as the sub-graph of G, containing v_0, v_g and each vertex that is both a successors of v_0 and a predecessor of v_g as well as all corresponding edges ($\in \mathcal{E}_G$) between them. Note that $C^G_{v_0 \to v_g}$ can be extracted from G (as shown in Figure 7.1). This can be done in linear time and space ($O(|\mathcal{V}_G| + |\mathcal{E}_G|)$) by constructing the *successor graph* of v_0 and the *predecessor graph* of v_g using breadth-first sweep and taking their intersection.

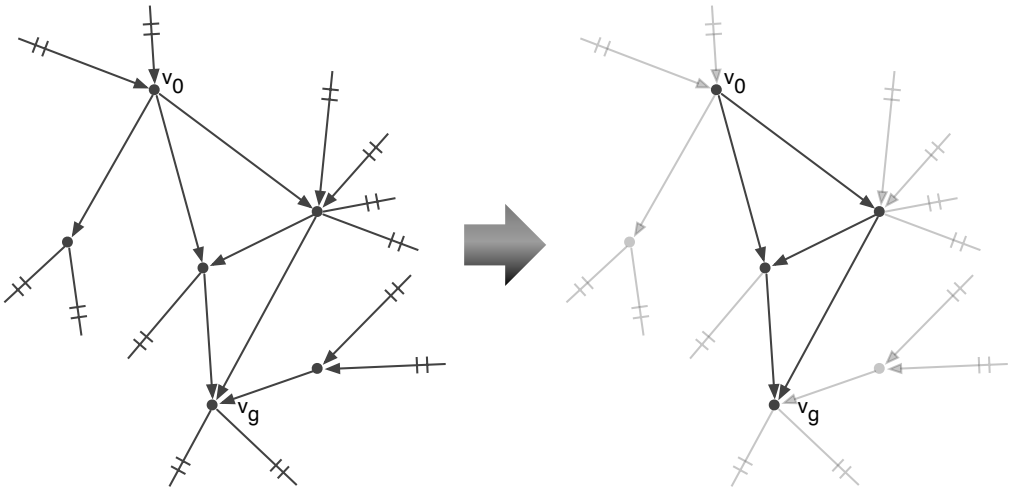


FIGURE 7.1 Extracting $C^G_{v_0 \to v_g}$ from some graph G

In the remainder of the chapter, the discussion refers to the connection graph $C^G_{v_0 \to v_g}$ as simply C, to the set of edges $\mathcal{E}_C$ as $\mathcal{E}$, and to the set of edges $\mathcal{V}_C$ as $\mathcal{V}$.

7.4 Challenges of the Target-Value Search Problem

Target-value Search Problem is challenging because it does not exhibit *optimal substructure*. A problem is said to have optimal substructure if an optimal solution can be constructed efficiently from optimal solutions of its sub-problems. This property is a prerequisite of dynamic programming. For example, the shortest path problem has optimal substructure. Any sub-path of a shortest path is itself a shortest path with respect to its first and last vertices. For example, in Figure 7.2 the shortest path from v_0 to v_g will use the shortest path from v_1 to v_g independent of the path selection from v_0 to v_1. The shortest path from v_1 to v_g will then be combined with whatever the shortest path between v_0 and v_1 is.

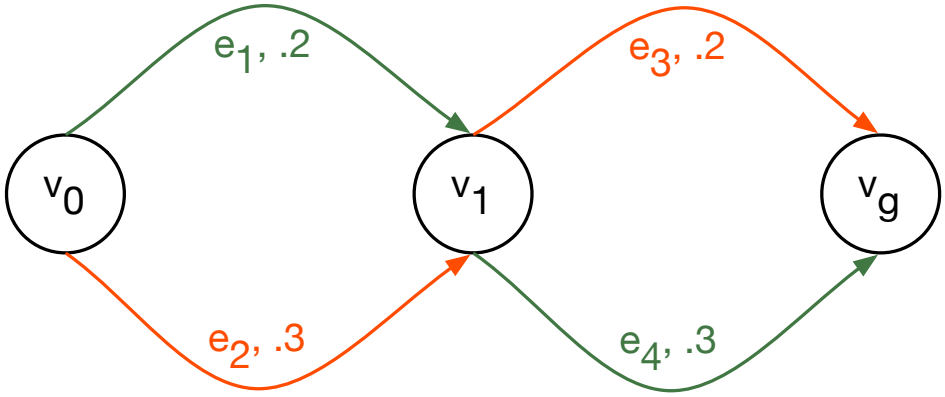


FIGURE 7.2 Example Target-Value Search Problem

The Target-value Search Problem lacks optimal substructure. A sub-path of a target-value path with respect to some target-value T is not necessarily a target-value path between its first and last vertices with respect to the same T. For example, let the target-value be $T = 5$, let the graph be the one shown in Figure 7.2, and let the target-value path problem from v_0 to v_g be decomposed into two sub-problems: (1) from v_0 to v_1 and (2) from v_1 to v_g. Given the target-value T, the solutions of the sub problems can not be construed as a solution of the original problem from v_1 to v_g.

7.5 Naive Target-value Search Algorithm

In this section a naive target-value search algorithm is introduced and used to illustrate the decomposition, which most Target-value Search Algorithms are based on. The naive algorithm is constructed from Equation 7.1 by decomposing the argmin over complete paths from v_0 to v_g into a recursive set of argmin calculations over sub-paths from v' to v_g where v' is an immediate successor of v_0. After traveling from v_0 to v' some value has been added to the prefix. To be more specific, the value, which has been added, is the edge value of edge (v_0, v'). Therefore the target-value can be adjusted by this amount to get the remaining target-value for the remaining search or respectively to search the suffix of the path. Formally,

Definition 28. *The set of target-value paths $\mathcal{P}^T_{v_0 \to v_g}$ can be recursively defined as:*

$$\begin{aligned} \mathcal{P}^T_{v_0 \to v_g} &= \begin{cases} \{\langle v_g \rangle\} & \textit{if} \quad v_0 = v_g \\ \emptyset & \textit{if} \quad notConnected(v_0, v_g) \\ v_0 \circ \phi(T, v_0) & \textit{otherwise} \end{cases} \\ \phi(T, v_0) &= \underset{p' \in \bigcup_{(v_0, v') \in E} \mathcal{P}^{T'}_{v' \to v_g}}{\operatorname{argmin}} |g(p') - T'| \end{aligned} \tag{7.2}$$

with $T' = T - g(\langle v_0, v' \rangle)$ being the remaining target-value. The operator $\circ$ maps a vertex v and a set of paths $\mathcal{P}$ to sets of paths $\mathcal{P}' = \{\langle v, v_1, \ldots, v_n \rangle | \langle v_1, \ldots, v_n \rangle \in \mathcal{P}; (v, v_1) \in \mathcal{E}; v, v_i \in \mathcal{V}\}$.

As shown in the Equation 7.2, the set of target-value paths is one of:

- the singleton $\{\langle v_g \rangle\}$ iff $v_0 = v_g$ (Note that only DAG's are considered and therefore if $v_0 = v_g$ there exists no other path between them),
- the empty set $\emptyset$ iff v_0 and v_g are not connected in G (since v_0 and v_g are not connected in G, there will be no path between v_0 and v_g in C),
- the set of paths resulting from concatenating to v_0 the "best" (w.r.t. the remaining target-value T') completions in C.

It remains to be shown that the naive algorithm terminates. Let the cardinality of a path $||p||$ be the number of vertices in its sequence minus one (i.e., the number of edges in the path). Since C is a finite DAG,

1. for any partial path from v_0 to v_n, there are a finite number of successors v'_n —- certainly less than $|\mathcal{V}|$.
2. all paths in the graph are finite —- certainly $||p|| < |\mathcal{V}|$
3. since the branching is finite and the depth is finite, the recursion will be finite. The number of paths $<< |\mathcal{V}|^{|\mathcal{V}|}$.

The naive algorithm clearly shows the structure decomposition and how a solution can be computed. Unfortunately, due to the tree recursive nature of this algorithm, the computation can lead to an exponential blow-up, since the number of paths from v_0 to v_g in C can be exponential in the number of vertices in C. However, using a heuristic search approach can (potentially) avoid the enumeration of most paths.

7.6 Using Heuristic Search

In general, best-first search explores first the prefixes that appear to be most likely to lead towards the goal. A special case of best-first search is A^*-search (Hart et al. 1968), which takes the distance already traveled into account. A straightforward approach to solve the target-value path problem is to search the prefix space with A^*-search using Equation 7.3:

$$f(pre) = \mid g(pre) + h(pre.lastV) - T \mid \tag{7.3}$$

where $g(pre)$ is the actual path value of the prefix pre and $h(pre.lastV)$ is an admissible lower bound heuristic for the path value from pre's last vertex to the goal vertex. Prefixes ending in the same vertex and having equal $g(pre)$ values can be considered duplicates and can be used to prune the search tree. Note, that the evaluation function $f(pre)$ will not be a lower bound on the target-value deviation due to the non-linearity introduced by the absolute value operator. See Figure 7.3 for an example.

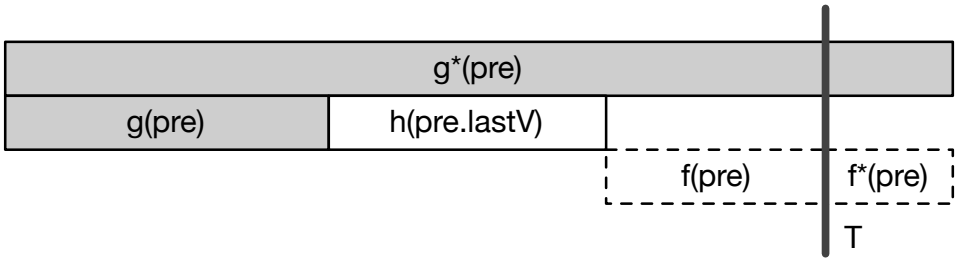


FIGURE 7.3 Heuristic that underestimates suffix values will not lead to an admissible target-value search heuristic: here $f(pre) > f^*(pre)$ while $g(pre) + h(pre.lastV) < g^*(pre)$

Therefore, the search can not optimally be terminated as soon as a goal vertex is to be expanded. In order to guarantee optimality, the currently *best* target-value path from v_0 to v_g w.r.t. the target-value has to be maintained and all prefixes in C for which Equation 7.4 holds have been explored.

$$g(pre) + h(pre.lastV) < T + f(best) \tag{7.4}$$

After all prefixes in C for which Equation 7.4 holds have been explored, the search can be terminated with the currently best target-value path as the result path. Note that for big target-values this can still lead to an exponential blow-up, since the number of prefixes in C, for which Equation 7.4 holds, can be exponential in $|V|$.

However, using $h(pre.lastV)$ as the lower bound heuristic lets us prune prefixes for which Equation 7.5 holds.

$$g(pre) + h(pre.lastV) > T + f(best) \tag{7.5}$$

To further limit the search space a non-underestimating upper bound heuristic can be introduced, denoted $h^+(pre.lastV)$, in addition to the lower bound heuristic. This upper bound heuristic can be used to prune all prefixes in the search tree for which Equation 7.6 holds.

$$g(pre) + h^+(pre.lastV) < T - f(best) \tag{7.6}$$

The next section proposes an algorithm that uses both lower and upper bound heuristics and introduces an evaluation function $f(pre)$ that does not overestimate the target-value deviation.

7.7 Heuristic Target-Value Search

The proposed algorithm uses lower bounds and upper bounds to limit the search space, and introduces an admissible (non-overestimate) evaluation function $f(pre)$ for the target-value deviation.

For each vertex v in C, let $h^-(v)$ be a non-overestimate lower bound heuristic and $h^+(v)$ be a non-underestimate upper bound heuristic on the path value from v to the goal vertex. Given a vertex v with its corresponding lower and upper bound heuristics, a range can be defined, denoted $r(v)$, that bounds the path value of all possible suffixes from v to the goal vertex as shown in Equation 7.7.

$$r(v) = [h^-(v), h^+(v)] \tag{7.7}$$

The lower bound of the interval is denoted as $r(v).l$ and the upper bound as $r(v).u$.

Furthermore, the path family of some prefix pre, denoted fam^{pre}, is the set of all paths from v_0 to v_g sharing the common prefix pre. Given some path family fam^{pre}, a path value interval (similar to the range) can be constructed by combining $g(pre)$ with either $r(pre.lastV).l$ or $r(pre.lastV).u$ as shown in Equation 7.8.

$$\begin{aligned} i(pre) \;\; = \;\; & [\; g(pre) + r(pre.lastV).l, \\ & \quad g(pre) + r(pre.lastV).u \;] \end{aligned} \tag{7.8}$$

The path value interval $i(pre)$ bounds the path value for all paths in fam^{pre}. The lower bound of the interval is denoted as $i(pre).l$ and the upper bound as $i(pre).u$. The interval $i(pre)$ is used to construct an admissible (non-overestimate) evaluation function $f(pre)$ for the target-value deviation:

$$f(pre) = \begin{cases} 0 & i(pre).l \leq T \leq i(pre).u \\ i(pre).l - T & i(pre).l > T \\ T - i(pre).u & i(pre).u < T \end{cases} \tag{7.9}$$

If the target-value falls in the path value interval of some prefix pre, there is potentially a path in fam^{pre} that exactly achieves the target-value. In this case $f(pre)$ returns zero, as it must be a true non-overestimate of the target-value deviation. For prefixes where the target-value does not fall within the interval the following fact can exploited . Since it is assumed that the lower bound $i(pre).l$ and upper bound $i(pre).u$ are true bounds, it is known that there exists no path in fam^{pre} outside those bounds. If the target-value is below the lower bound of the interval, the smallest possible deviation is $i(pre).l - T$. If the target-value is above the upper bound, the smallest deviation is $T - i(pre).u$. The evaluation function $f(pre)$ in Equation 7.9 is therefore an admissible lower bound on the target-value deviation.

The path value intervals can also be used to calculate a non-underestimate upper bound of

target-value deviation, as shown in Equation 7.10.

$$\begin{aligned} pruneBound(pre) &= \max(\mid T - i(pre).l \mid, \\ &\qquad \mid T - i(pre).u \mid) \end{aligned} \tag{7.10}$$

This upper bound function can be used to maintain a global best upper bound on target-value deviation $pruneBound(best)$. This global best upper bound can be leveraged to prune prefixes pre where

$$f(pre) > pruneBound(best). \tag{7.11}$$

The evaluation function $f(pre)$ together with $pruneBound(pre)$ enables search to use A^*. The search basically results in a shortest path search when T is below the interval and longest path search when T is above the interval. The challenge occurs when the target-value falls within the interval. This case occurs often near the beginning of the search, when there are many possible path completions for each prefix, making the interval associated with the last vertex of the prefix very wide. Many vertices will therefore be assigned the same heuristic value, namely zero. While admissible, this effectively forces us into a blind search. The blind search is not explicit, but results from the behavior of the A^* algorithm. The prefixes that evaluate to zero will move to the front of A^*'s priority queue and will be expanded first. A prefix that evaluates to zero ($f(pre) = 0$) is denoted as a zero-prefix and the space of all zero-prefixes as the blind-spot. Search in the blind-spot exposes two main difficulties:

- Search is not guided in the blind-spot and therefore will perform as poorly as the naive algorithm introduced earlier.

- The priority queue might grow exponentially during the blind search, due to a combination of the tree structure of the naive algorithm and our inability to prune any zero-prefix (it potentially leads to a path that exactly achieves the target-value).

Target-value search can therefore still lead to an exponential blow-up in time and space, since the number of prefixes in the blind-spot can be exponential in $|V|$. However, once the blind-spot is exhaustively searched, the target-value starts to fall outside the path value intervals. Informative (non-zero) heuristic bounds can then be assigned, and informed search will again dominate.

The next sections introduce two techniques to improve the performance of the Heuristic Target-Value Search: *Max-Value-First Search* and *Multi-interval Heuristic*. *Max-Value-First Search* is a best-first search technique, which uses an secondary objective function of break ties of the primary objective function. This results in a significant size reduction of the priority queue and leads to great performance improvements. *Multi-interval Heuristic Search* leverages multiple intervals to represent the heuristic value ranges, which leads to a blind spot reduction. *Multi-interval Heuristic Search* is only briefly discussion and as it is described in more detail in (Schmidt et al. 2009).

7.8 Max-Value-First Search for Blind-Spot Search

The previous section illustrated that the size of the priority queue can grow exponentially during the blind search. This is due to a combination of an inability to prune zero-prefixes and the uninformed search in the blind-spot. Since search is not guided in the blind-spot, search in the blind spot could result in breadth-first search. The challenge using breadth-first search manifests in the fact that the priority queue can grow exponentially within the blind-spot. That can lead to an exponential blow-up in space complexity.

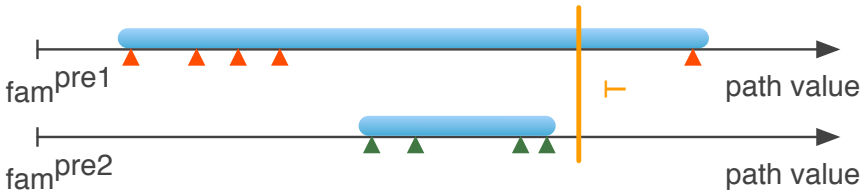
Instead of unguided search, *max-value-first search* can be used within the blind-spot. *Max-value-first search* uses a secondary objective function to guid the search in the blind spot. In more detail, if the primary objective function f(pre) evaluates a prefix pre to zero (blind spot), Max-value-first uses a secondary objective function, which expands the prefix with the greatest path value $g(pre)$ first. *Max-value-first search* is different from depth-first search, which chooses randomly among the prefixes with the maximum edge number (depth). Therefore, a search approach, which leverages depth-first search within the blind spot can be interpreted as a serach which uses depth of a prefix as secondary objective. In comparison, *Max-value-first search* guides the search directly to the frontier of the blind-spot where a zero-prefix can be expanded with successors which are not zero-prefixes. Pruning can then be applied to those successors. This is also interesting from an anytime algorithm perspective. Using max-value-first search enables strong anytime performance, because max-value-first search guides search in average faster out of the blind-spot (in average faster than depth-first search). Given a fixed number of expansions, the number of non-zero prefixes found is greater or equal than with depth-first search or breadth-first search. Therefore if an anytime algorithm is desired, the most promising non-zero-prefix from A^*'s priority queue can be periodically selected and expanded towards the goal. Note that non-zero-prefixes can be completed readily using informed search, e.g. A^*. Without the anytime requirement, the search will still expand all zero-prefixes, but now in a different order. This leads to significant size reduction of the priority list and additionally allows pruning earlier in the process. A^* can be used as the primary search algorithm and *max-value-first search* can be implemented within the blind-spot by using the following tie-breaking rule: Given two prefixes have equal $f(pre)$-values ties can be broken by preferring the prefix with greater $g(pre)$-value. Since all prefixes in the blind-spot have the same $f(pre)$-value, namely zero, the secondary objective (tie-breaking rule) dominates within the blind-spot. That reduces the space required by the priority queue, because the blind spot is explored in a depth-first fashion but with increased pruning strength while searching the blind-spot.

Assume that the suffix range bounds are perfect (true for the multi-interval heuristic proposed in the next section), then target-value search using *max-value-first search*, enforced by the tie-breaking rule, requires at worst $O(bd + |V| * r_{max})$ in space where b is the maximum branching factor in the blind-spot, d is the maximum depth of the blind-spot search tree and $|V| * r_{max}$ accounts for the stored ranges for all vertices (r_{max} is the maximum allowed number of ranges per vertex). As illustrated in the experiment section this enables target-value search to scale to much bigger problems. The next section briefly introduces *Multi-interval Heuristic Target-value Search*, which leads to size reductions the blind-spot.

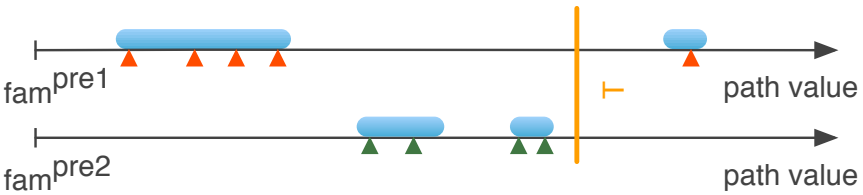
7.9 Multi-interval Heuristic Target-value Search

As known from the previous two sections, the search performance is directly correlated to the size of the blind-spot, which is directly correlated to the space covered by path value intervals. Therefore, if less space is covered then the blind-spot is smaller, which leads to increased search performance. This section introduces the idea of a multi-interval heuristic, which reduces the size of the blind-spot. The key idea of multi-interval heuristic target-value search is to characterize the path value space associated with a path family by using a set of intervals instead of just one interval. The set of intervals associated with a path family is denoted as $I(pre)$.

For example, consider two path families fam^{pre_1} and fam^{pre_2}, which are characterized either with a single interval (Figure 7.4(a)) and or with two intervals (Figure 7.4(b)).



(a) single path value interval



(b) two path value intervals

FIGURE 7.4 Single vs. multiple path value intervals.

The triangles indicate existing path values, and the blue bars the intervals. Given the target-value T, in the single interval case prefix pre_1 is a zero-prefix ($f(pre_1) = 0$) and therefore an element of the blind-spot. This forces target-value search to expand pre_1 before pre_2, even if prefix pre_2 leads ultimately to a better solution. In the case of two intervals per path family, the space covered by the intervals is smaller as by a single interval, shown in Figure 7.4. In general, in the worst case a multi-interval representation covers at most the same. In the case where two intervals are used, prefix pre_1 is no longer an element of the blind-spot. As a result, both prefixes, pre_1 and pre_2, evaluate to a non-zero heuristic value and therefore prefix pre_2 can directly be expanded, since it is closer to the target-value.

A non-overestimate lower bound evaluation function on the target-value deviation $f(pre)$ can now be constructed in much the same fashion as for a single interval. For the multiple interval case, the target-value can be compared to the closest upper interval bound and the closest lower interval bound among all intervals associated with a path family. This leads to the following lower bound evaluation function: (For clarity and brevity, but with some abuse of notation, a path value interval $i(pre)$ is referred to as i and a set of intervals $I(pre)$ as I.)

$$f(pre) = \begin{cases} 0 & \exists\, i \in I : i.l \leq T \leq i.u \\ \min(\min\limits_{i \in I} |i.l - T|, & \\ \quad \min\limits_{i \in I} |T - i.u|\ ) & \text{otherwise} \end{cases} \tag{7.12}$$

The multiple intervals can also be exploited for pruning. A non-underestimate upper bound heuristic on the deviation can be defined from the set of intervals associated with some path family in a similar manner.

$$pruneBound(pre) = \max(\min_{i \in I, i.l \leq T} T - i.l, \min_{i \in I, i.u \geq T} i.u - T\ ) \tag{7.13}$$

By converting the set of path value intervals to a non-overestimating lower bound evaluation of deviation from target-value, it is possible to search with A^*.

Multi-interval target-value search relies on efficient computation of the multi-interval heuristic assigned to each path family. Details on how a Multi-interval Heuristic can be computed can be found in (Schmidt et al. 2009).

7.10 Applications of Target-value Search

The Target-value Search Problem arises in a variety of domains ranging from activity recommendation to navigation to diagnosis. This section outlines two application domains in which Target-value Search Problems appear: Automated diagnosis of plan-driven Systems and consumer recommendation.

7.10.1 Automated diagnosis of plan-driven systems

Plan-driven control is a key technology for creating reliable and highly adaptable production systems. A practical example, of a planner driven system can be found in the modular redundant printing engine introduced in Chapter 2.4.1. The system is composed of 197 independent modules including feeders, marking engines, redirectors, finishers, etc. The modules work together to produce a variety of printed products. Given an user specified goal, such as a bound report with the following pages and spot colors, a model-based planner is used to synthesize a control program to provide real time coordination and control for the modules (Fromherz et al. 2003, Do et al. 2008*b*). However, real-world systems might evolve overtime and require reasoning about their underlying health. In printing systems, rare intermittent faults can be particularly difficult to diagnose. To reduce the cost of diagnosis, it is possible alter production plans such that more information can be gained from production plans, resulting in informative production plans.

A technique for creating informative production plans is called *Pervasive Diagnosis* see Section 3.2.2.4, and (Kuhn et al. 2008*b*). A central problem in pervasive diagnosis is to find production plans that fail with a probability mass optimal for information gain, see Section 3.5.1

and (Liu et al. 2008). If the failure probability is too low, the plan will likely never fail and if it is too high, it probably uses components already known to be broken. Pervasive diagnosis can be formulated as a target-value search, Section 3.5.2. Let the action set be the edge set $\mathcal{E}$, and the success probability of an action be the edge weight. Given independent action failures, the optimal probability of a plan succeeding or failing is equal to the "total probability mass" of a plan.

7.10.2 Consumer recommendation

A consumer recommendation of complex activities and products, such as route planning (Winter 2002) or vacation planning, can be formalized as a multi-step search problem consisting of a start state, a goal state, a set of transitions and some optimization criteria. Many types of activity planning have a target-value component.

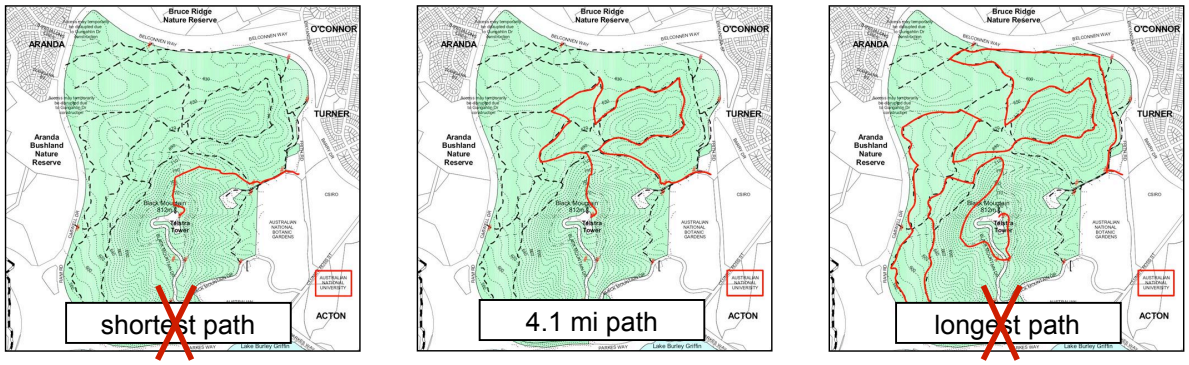


FIGURE 7.5 Applications for the Target-Value Path Problem: Hiking trail planning.

For example, a consumer may wish to find a route for a hiking trip (Figure 7.5) that takes a certain amount of time, burns a certain amount of calories, or as in illustrated in Figure 7.5 is as close as possible to 4 miles long. The more the value of the path deviates from the target-value, the less desirable the path is to the consumer. Specifically, the shortest and the longest path are not of interessed.

In practical domains, such as hiking, apparent cycles in the domain such as loops on a hiking trail map can be seen as not true cycles in state space, as the state of the hiker before and after hiking around the loop might be very different. In all practical examples studied as part of this work, it makes sense to unroll the problem (to some depth) and model it as a DAG.

7.11 Experiments

The experiments in this section show the performance of different approaches on synthetic and real-world graphs. Graphs for the real world applications were extracted as DAGs from a system model for the prototype printer shown in Figure 2.3. Graphs for the synthetic experiments were supplied by a graph generator which constructs DAGs like that in Figure 7.6. The

structure of these graphs is inspired by graphs taken from the connectivity graphs of a number of production systems. Each graph starts with a vertex v_0, branches out to a grid of interior vertices and then collapses back down to a goal vertex v_g. Within the grid, a vertex is always connected to its direct successor in the same row. Each vertex is also connected to each other vertex in the next column with a probability p. The grid is parameterized by the number of columns (width W) and the number of rows (height H).

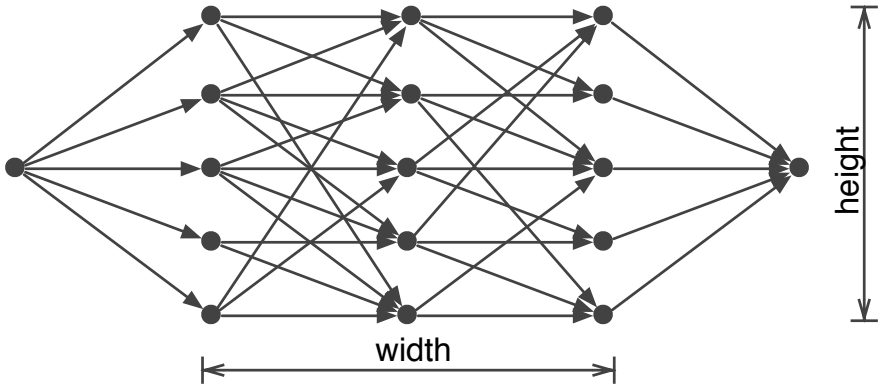


FIGURE 7.6 Schematic of graph parameterization

In the plots the y-axis shows either search time in milliseconds (search time includes the heuristic construction) or the maximum size of the search queue in number of vertices, and the x-axis shows k different normalized target-values, spanning between the shortest path (SP) and the longest path (LP) in each graph.

The first experiment uses the tightly integrated parallel printer (tipp) domain graph and a set of synthetic graphs (7x7, 8x8, and 9x9 grid, connection probability $p = 0.5$). The results are plotted for $k = 30$ different target-values each averaged over 200 runs. In the experiment four different algorithms are compared with different f functions on this problem:

- $|T - g|$: A^* with $f(pre) = |T - g(pre)|$.
- $|T - (g + h)|$: A^* with $f(pre) = |T - (g(pre) + h(pre.lastV))|$.
- $HTVS$: Heuristic Target-Value Search with Breadth-first blind-spot search.
- $HTVS{-}MVF$: Heuristic Target-Value Search with Max-Value-First blind-spot search.

Figure 7.7 and Table 7.1 give the search time results of the first experiment. It can be seen that both $|T - g|$ and $|T - (g - h)|$ perform considerably slower with increasing target-value. That is due to the non-admissible $f(pre)$ function and the resulting termination criterion. Therefore to terminate optimally the search has to exhaust the prefix space of all prefixes with a shorter then roughly the target value. An interesting observation is that $HTVS$ is able to outperform both $|T - g|$ and $|T - (g - h)|$ on smaller problems, but with growing problem size it starts to fall behind for target-values smaller than about 0.7. This is caused by the

enormous memory requirement due to the breadth-first blind-spot search. As shown in Figure 7.8 and Table 7.2, $HTVS$ requires orders of magnitude more memory space than all other approaches. In comparison, $HTVS - MVF$, which uses Max-Value-First blind-spot search, reduces the memory requirement by exploring the blind-spot in a more directed fashion. The reduced memory management enables $HTVS - MVF$ to dominate in search time as shown in Figure 7.7 (Table 7.1) and in the zoomed-in Figure 7.9 (Table 7.3).

In Figure 7.9 it is demonstrated how $HTVS - MVF$ is influenced by using the multi-interval heuristic. The experimental results show that doubling the number of intervals per path family reduces the search time roughly by half. This coincides with the observations of (Holte & Hernádvolgyi 1999) about memory-based heuristics.

In experimental results problems have been solved up to a graph size of 500x500-graph using $HTVS-MVF-120$, whereas the three other approaches could only solve problems up to 15x15-graph-size. That results from the memory efficient max-value-first blind-spot search and the informative multi-interval heuristic. More detailed experiments on the multi-interval heuristic can be found in (Schmidt et al. 2009).

7.12 Conclusions

This chapter has introduced a novel and widely applicable class of combinatorial search problems, which are called target-value search problems. Further it was shown that target-value search problems can be reduced to the well-known shortest path problem using an evaluation function that gives a lower bound on the deviation of path value and a specified target-value. While the target-value problem is not decomposable (no optimal substructures), it was possible to show that it can be computed from an interval-based heuristic on upper and lower bounds, which are decomposable. Finally a novel search algorithm, called max-value first search, has been described, which searches the blind-spot more efficiently resulting in better memory and time performance. An extensive literature search indicates that there exist no other algorithms for efficiently solving such problems. The contribution of this chapter makes it possible to solve target-value search problems such as planning consumer activity recommendations and the integration of planning and diagnosis.

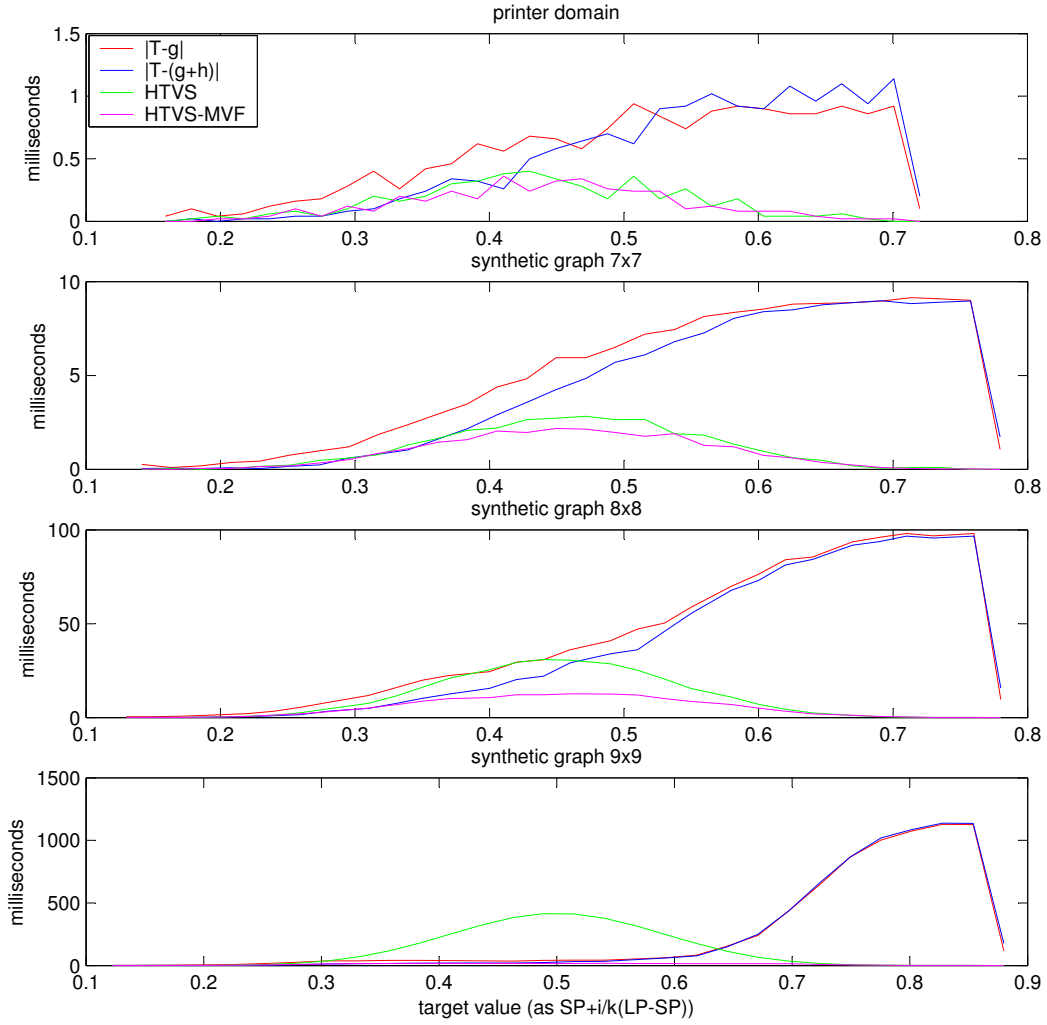


FIGURE 7.7 Search time in milliseconds as a function of the target-value. Target-value normalized as SP+i/k(LP-SP)

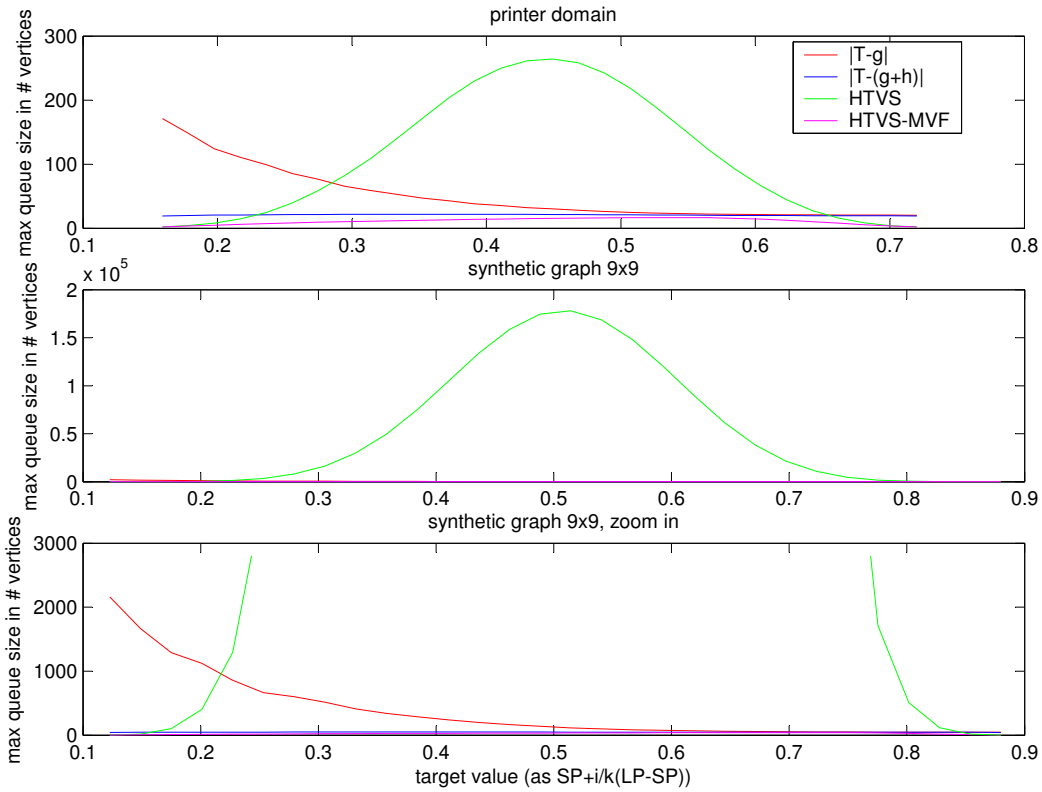


FIGURE 7.8 Maximum size of search queue as a function of the target-value. Target-value normalized as SP+i/k(LP-SP)

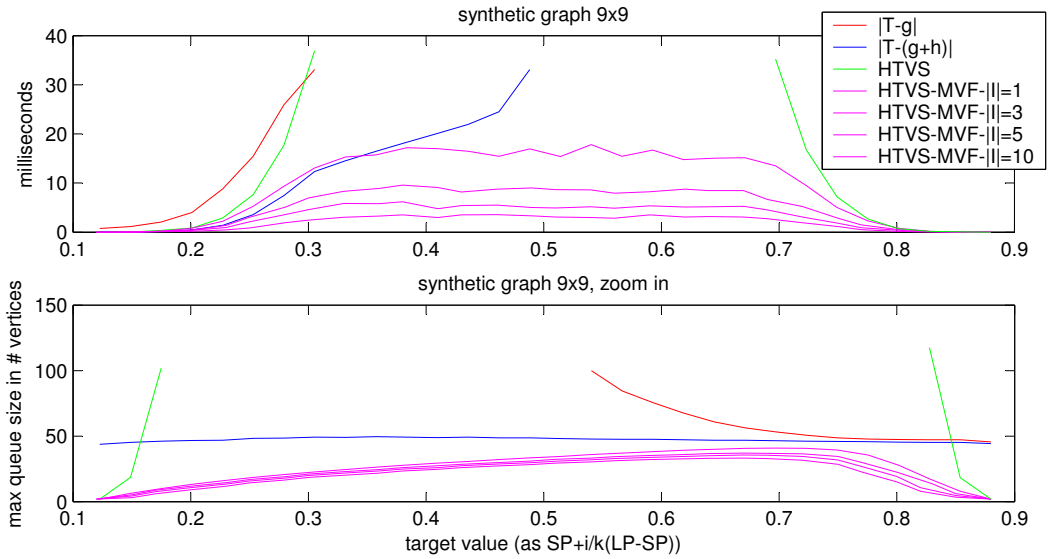


FIGURE 7.9 Zoom in for time and space as a function of the target-value. Target-value normalized as SP+i/k(LP-SP)

target value	0.20	0.30	0.40	0.51	0.59	0.69	0.80
printer domain							
$\|T-g\|$	0.06	0.28	0.46	0.66	0.94	0.90	0.92
$\|T-(g+h)\|$	0.02	0.08	0.34	0.58	0.62	0.90	1.10
$HTVS$	0.02	0.10	0.30	0.34	0.36	0.04	0.06
$HTVS-MVF$	0.02	0.12	0.24	0.32	0.24	0.08	0.02
synthetic graph 7x7							
$\|T-g\|$	0.36	1.20	3.48	5.94	7.44	8.84	9.14
$\|T-(g+h)\|$	0.08	0.60	2.16	4.84	6.80	8.76	8.82
$HTVS$	0.04	0.60	2.08	2.82	1.90	0.48	0.10
$HTVS-MVF$	0.04	0.52	1.58	2.14	1.90	0.36	0.04
synthetic graph 8x8							
$\|T-g\|$	1.24	8.04	22.46	36.12	50.36	85.42	98.00
$\|T-(g+h)\|$	0.18	3.32	12.52	29.12	45.82	84.16	96.64
$HTVS$	0.16	4.64	20.84	30.56	20.48	2.50	0.20
$HTVS-MVF$	0.24	3.08	10.08	12.58	10.14	1.92	0.30
synthetic graph 9x9							
$\|T-g\|$	4.02	36.68	39.90	42.12	61.14	644.34	1074.82
$\|T-(g+h)\|$	0.44	12.36	20.12	31.34	59.10	659.20	1085.32
$HTVS$	0.84	36.94	251.58	412.86	246.56	16.76	0.72
$HTVS-MVF$	0.78	13.08	17.00	15.40	16.70	9.48	0.80

TABLE 7.1 Search time in milliseconds as a function of the target-value. Table contains data illustrated in Figure 7.7. Unit: milliseconds; Target value as SP+i/k(LP-SP);

target value	0.20	0.30	0.40	0.51	0.59	0.69	0.80
printer domain							
$\|T-g\|$	110.70	65.60	43.10	30.30	24.90	21.30	20.80
$\|T-(g+h)\|$	20.80	21.70	21.90	21.50	20.90	20.00	19.70
$HTVS$	14.70	82.20	203.40	264.30	218.40	66.30	16.00
$HTVS-MVF$	6.00	10.30	13.20	15.60	16.40	14.40	7.90
synthetic graph 9x9							
$\|T-g\|$	1122.70	514.40	241.90	114.20	75.60	50.60	47.40
$\|T-(g+h)\|$	46.60	49.20	48.80	48.10	47.50	46.10	45.40
$HTVS$	405.30	16315.40	104170.60	178034.70	120541.70	10706.50	513.10
$HTVS-MVF$	13.10	22.70	29.40	34.40	38.10	40.70	27.80

TABLE 7.2 Maximum size of search queue as a function of the target-value. Table contains data illustrated in Figure 7.8. Unit: max queue size in # vertices; Target value as SP+i/k(LP-SP);

target value	0.20	0.30	0.40	0.51	0.59	0.69	0.80
milliseconds							
$\|T-g\|$	4.02	33.08	39.90	48.12	61.14	644.34	1074.82
$\|T-(g+h)\|$	0.44	12.36	20.12	38.14	59.10	659.20	1085.32
$HTVS$	0.84	36.94	251.58	412.86	246.56	16.76	0.72
$HTVS-MVF-\|I\|=1$	0.78	13.08	17.00	15.40	16.70	9.48	0.80
$HTVS-MVF-\|I\|=3$	0.42	6.98	9.10	8.66	8.24	5.22	0.58
$HTVS-MVF-\|I\|=5$	0.34	4.58	4.80	4.90	5.38	3.10	0.40
$HTVS-MVF-\|I\|=10$	0.18	2.46	3.00	3.06	3.50	1.92	0.18
max queue size in # vertices							
$\|T-g\|$	1122.70	514.40	241.90	114.20	75.60	50.60	47.40
$\|T-(g+h)\|$	46.60	49.20	48.80	48.10	47.50	46.10	45.40
$HTVS$	250.00	16315.40	104170.60	178034.70	120541.70	10706.50	513.10
$HTVS-MVF-\|I\|=1$	13.10	22.70	29.40	34.40	38.10	40.70	27.80
$HTVS-MVF-\|I\|=3$	11.70	21.00	27.30	32.20	35.10	36.40	22.50
$HTVS-MVF-\|I\|=5$	10.70	19.90	25.90	30.80	33.80	34.50	19.10
$HTVS-MVF-\|I\|=10$	9.00	18.40	24.60	29.30	32.10	31.60	15.00

TABLE 7.3 Zoom in for time and space as a function of the target-value. Table contains data illustrated in Figure 7.9. Graph: synthetic graph 9x9; Target value as SP+i/k(LP-SP);

CHAPTER 8

Conclusions

This chapter summarizes the work on Self-diagnosing Agents by outlining the overall approach and explaining how those concepts can be integrated into the widely used Learning Agent architecture. Finally, it concludes by revisiting the core contributions and stating some suggestions for future work.

A long standing vision of artificial intelligence has been to build fully autonomous systems that achieve desired goals over a long period of time without external intervention. This requires that autonomous systems know about their own capabilities (model), reason about their course of action (planning), and reflect on their actual behavior (diagnosis). In general, long-life autonomy can be seen as a combination of two methods: (1) diagnosis to determine the current condition of the system and (2) planning to optimize system operation for the diagnosed condition. A step towards long-life autonomy is to integrate automated diagnosis with regular operation.

This work introduced a new architecture, coined a *Self-diagnosing Agent*, which tightly integrates diagnosis into regular operation such that active information gathering, online diagnosis, and regular operation can occur at the same time. This novel integration leads to higher long-run performance than alternative integrations. The presented approach is realized by a novel diagnosis paradigm called *pervasive diagnosis*. The core idea of pervasive diagnosis is to trade off information gain objectives and performance objectives to generate operational plans that unveil diagnostic information. The conceptual framework of pervasive diagnosis is illustrated in Figure 8.1.

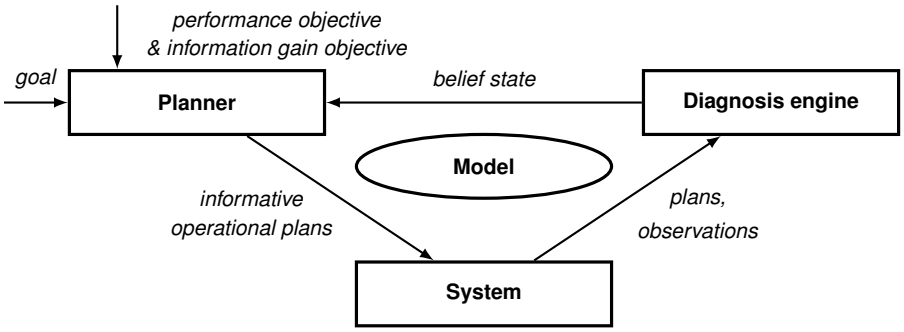


FIGURE 8.1 Pervasive Diagnosis: Integration of Operational Planning and Active Diagnosis.

Consider a system where operational goals can be achieved in multiple ways. The planner

exploits this flexibility during the plan generation process to determine informative operational plans. Ideally, executing such plans results in valid operation and informative observations. After gathering observations the diagnosis engine updates its beliefs online, updates the system model to represent the current capabilities, and forwards the beliefs to the planner. The planner then determines future plans based on the current beliefs, the performance objectives, and the information gain objectives. As a result systems that embody pervasive diagnosis benefit from high long-run performance as synergy effects may be leveraged by exploiting the overlap between operational plans and diagnostic plans.

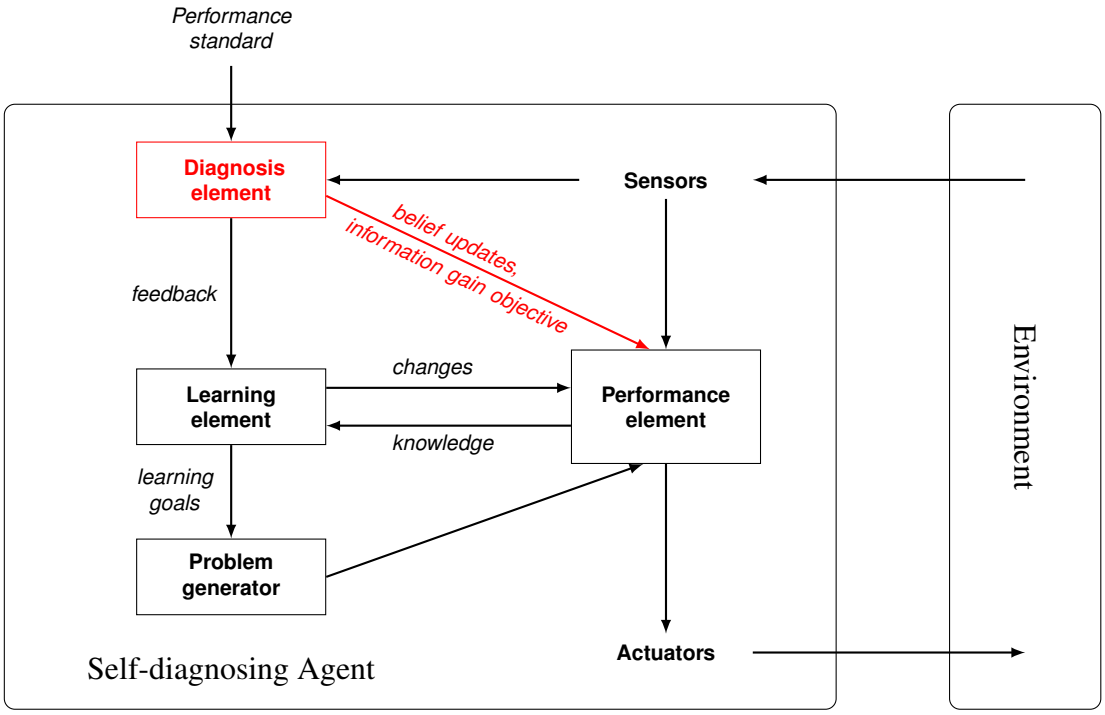


FIGURE 8.2 Self-diagnosing Agent

A Self-diagnosing Agent extends the widely used framework of a Learning Agent by integrating pervasive diagnosis. The goal of a Learning Agent is to increase its understanding of itself and the world by learning to perform better than without learning (see Section 2.5.6). In general, a Learning Agent realizes this goal by identifying incorrect or incomplete knowledge and improving its knowledge by learning better predictive models directly from observations. In comparison, a Self-diagnosing Agent extends those capabilities by the ability to perform diagnosis. The core differentiator between a Self-diagnosing Agent and a Learning Agent is that a Self-diagnosing Agent has the ability to perform root cause analysis to improve its understanding to guide learning. Due to the integration of diagnosis an agent can determine the root-causes of discrepancies between predicted and actual behavior before reasoning about how it learns. This analysis enables an agent to change its behavior with respect to the root-causes instead of continuously adapting to surface symptoms. This is particular important if the optimal reaction can only be inferred by reasoning over a set of symptoms. Suppose a sce-

nario in which a root-cause can only be observed non locally. In this case, an agent that relies on pure learning without root-cause analysis may conclude that a local adaptation is needed. Only an agent that performs in-depth reasoning over a set of observations may correctly conclude that the root-cause is non local and therefore can only be compensated by a non local adaptation.

The overall framework of a Self-diagnosing Agent is illustrated in Figure 8.2. The framework complies with the general Learning Agent framework, except that the critics element is replaced by a diagnosis element, which leads to the following four components: a performance element, a diagnosis element, a learning element, and a problem generator. The ability to perform diagnosis enables the diagnosis element to provide diagnosis belief updates and information gain objectives to the performance element. The performance element can then generate informative operational plans to efficiently perform active diagnosis during regular operation to increase the agent knowledge about its current state. The main contribution of this work is an overall framework, which tightly integrates regular operation and active diagnosis. In particular, an information criterion is defined that quantifies how informative plans are, a plan generation algorithm is designed to derive informative operational plans, and a diagnosis framework is introduced to efficiently perform online diagnosis for systems that plan. The overall framework is optimized for systems with faults multiple in number, intermittent in appearance, and potentially caused by hidden interactions.

Experimental results have shown that the theoretical benefits can be realized on real time applications. Throughout the discussion an application of this framework to a hyper-modular, multi-engine printer has been presented, which achieves up to eight percent higher long-run performance than alternative approaches. To date, the Self-diagnosis Agent Architecture has only been evaluated in the context of an experimental printer setting, but the results suggest that the same techniques generalize to a wide range of domains such as manufacturing, assembly, packing, logistic, transport, earth and space exploration, surgery, weaponry, and smart networks.

8.1 Future Work

In general, the contributions of this work are a step towards long-life autonomy. In its ambition, a long-life autonomous system achieves desired goals continuously over a long period of time without external guidance or intervention. This requires that autonomous systems know about their own capabilities (model), reason about their course of action (planning), and reflect on their actual behavior (diagnosis). Long-life autonomy can not simply be reached by robust and reliable operation. In a real-world scenario systems break, experience unforeseen events, or change overtime. Therefore it is especially true for long-life autonomous system that they need the ability to adapt to changes, to recover from unforeseen situations, and to interact with unknown environments.

As a result a long-life autonomous system can only operate successfully over a long period of time if it learns better predictive models, performs diagnosis to determine its current conditions, and leverages the gain knowledge from learning and diagnosis to optimize its behavior. As illustrated in Figure 8.2, the introduced Self-diagnosing Agent framework combines those

three processes by extending the widely used Learning Agent with the ability to diagnosis. As a result learning and diagnosis can occur during regular operation.

However, the Self-diagnosing Agent work does not address if and how the coexistence of learning and diagnosis can be exploited to leverage synergy effects to improve long-run performance. The intuition is that learning and diagnosis perform both information acquisition during regular operation to enrich their understanding about the system. However, the information acquisition during periods of learning and diagnosis have been considered separately and during distinct phases of analysis. Learning is concerned with building better predictive models. The same predictive models can then be used to diagnosis the system based on observations. Learning and diagnosis are both most efficient if the process of information acquisition is optimized based on their current state of knowledge. An open challenge is to bridge the gap between information acquisition for learning and diagnosis and to enable an uniform approach that considers simultaneously the extension of the predictive model and the determination of the current system state. The overall goal is to combine their information acquisition objectives to improve long-run performance. The Self-diagnosis Agent is a step towards this vision, but future work has to be done to investigate how an integration can lead to more efficient utilization of regular operation, online diagnosis, and online learning.

Bibliography

Abreu, R., Zoeteweij, P. & van Gemund, A. (2006), An evaluation of similarity coefficients for software fault localization, *in* 'In Proceedings of the 12th IEEE Pacific Rim Symposium on Dependable Computing (PRDC'06)', Riverside, CA, USA.

Ali, M. F., Veneris, A., Safarpour, S., Abadir, M., Drechsler, R. & Smith, A. (2004), Debugging sequential circuits using boolean satisfiability, *in* 'Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design', pp. 204–209.

Bellman, R. (1958), 'On a routing problem', *Quarterly of Applied Mathematics* **16**(1), 87–90.

Bellman, R. (1978), *An introduction to artificial intelligence : can computers think?*, Boyd & Fraser Publishing Company, San Francisco, CA, USA.

Berger, J. O. (1995), *Statistical Decision Theory and Bayesian Analysis*, Springer Verlay, New York.

Bottcher, C. (1995), No faults in structure?: how to diagnose hidden interactions, *in* 'IJCAI'95: Proceedings of the 14th international joint conference on Artificial intelligence', Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, pp. 1728–1734.

Bushnell, M. L. & Agrawal, V. D. (2000), *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits*, Kluwer Academic Publishers, Boston.

Cascio, F., Console, L., Guagliumi, M., Osella, M., Panati, A., Sottano, S. & Dupré, D. T. (1999), On-board diagnosis of automotive systems: From dynamic qualitative diagnosis to decision trees, *in* 'In Proc. Workshop on Qualitative Reasoning on Complex Systems and their Control at IJCAI99'.

Cervoni, R., Cesta, A. & Oddi, A. (1994), Managing dynamic temporal constraint networks, *in* 'Proceedings of AIPS-94', pp. 13–18.

Charniak, E. & McDermott, D. (1985), *Introduction to artificial intelligence*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Christensen, H., Batzinger, T., Bekris, K., Bohringer, K., Bordogna, J., Bradski, G., Brock, O., Burnstein, J., Fuhlbrigge, T., Eastman, R., Edsinger, A., Fuchs, E., Goldberg, K., Henderson, T., Joyner, W., Kavaraki, L., Kelly, C., Kelly, A., Kumar, V., Manocha, D., McCallum, A., Mosterman, P., Messina, E., Murphey, T., Peters, R. A., Shephard, S., Singh, S., Sweet, L., Trinkle, J., Tsai, J., Wells, J., Wurman, P., Yorio, T. & Zhang, M. (2009), A roadmap for us robotics: From internet to robotics, Technical report, Georgia

Institute of Technology, University of Southern California, Johns Hopkins University, University of Pennsylvania, University of California, Berkeley, Rensselaer Polytechnic Institute, University of Massachusetts, Amherst, University of Utah, Carnegie Mellon University, Tech Collaborative.

Cover, T. M. & Thomas, J. A. (1991), *Elements of Information Theory*, New York, NY: John Wiley and Sons, Inc.

DARPA (2004), 'Darpa grand challenge rule book.', http://www.darpa.mil/grandchallenge05/-Rules_8oct04.pdf. good description.

Davis, R. (1984), 'Diagnostic reasoning based on structure and behavior', *Artificial Intelligence* **24**(1), 347–410. qrps:also.

de Kleer, J. (2007*a*), Diagnosing intermittent faults, *in* '18th International Workshop on Principles of Diagnosis', Nashville, USA, pp. 45–51.

de Kleer, J. (2007*b*), Modeling when connections are the problem, *in* 'Proc 20th IJCAI', Hyderabad, India, pp. 311–317.

de Kleer, J. (2007*c*), Troubleshooting temporal behavior in "combinational" circuits, *in* '18th International Workshop on Principles of Diagnosis', Nashville, USA, pp. 52–58.

de Kleer, J., Kuhn, L., Liu, J., Price, B., Do, M. & Zhou, R. (2009), Continuously estimating persistent and intermittent failure probabilities., *in* '7th IFAC Symposium on Fault Detection, Supervision and Safety of Technical Processes'.

de Kleer, J., Mackworth, A. & Reiter, R. (1992), 'Characterizing diagnoses and systems', *Artificial Intelligence* **56**(2-3), 197–222.

de Kleer, J. & Williams, B. C. (1987), 'Diagnosing multiple faults', *Artificial Intelligence* **32**(32), 97–130.

Dearden, R. & Clancy, D. (2002), Particle filters for real-time fault detection in planetary rovers, *in* 'Proceedings of the Thirteenth International Workshop on Principles of Diagnosis', pp. 1–6.

Dechter, R., Meiri, I. & Pearl, J. (1991), 'Temporal constraint networks', *Artificial Intelligence* **49**, 61–95.

Dijkstra, E. (1959), 'A note on two problems in connexion with graphs', *Numerische Mathematik* **1**, 269–271.

Do, M. B. & Ruml, W. (2006*a*), Lessons learned in applying domain-independent planning to high-speed manufacturing, *in* 'Proceedings of ICAPS-06', pp. 370–373.

Do, M. B. & Ruml, W. (2006*b*), Lessons learned in applying domain-independent planning to high-speed manufacturing, *in* 'ICAPS', pp. 370–373.

Do, M. B., Ruml, W. & Zhou, R. (2008*a*), Planning for modular printers: Beyond productivity, *in* 'ICAPS', pp. 68–75.

Do, M., Ruml, W. & Zhou, R. (2008*b*), On-line planning and scheduling: An application to controlling modular printers, *in* 'Proceedings of the 23rd AAAI Conference on Artificial Intelligence (AAAI-08)'.

Fikes, R. & Nilsson, N. (1971), 'Strips: A new approach to the application of theorem proving to problem solving', *Artificial Intelligence* **2**, 189–208.

Fromherz, M. P. (2007), 'Planning and scheduling reconfigurable systems with regular and diagnostic jobs', US Patent 7233405.

Fromherz, M. P., Bobrow, D. G. & de Kleer, J. (2003), 'Model-based computing for design and control of reconfigurable systems', *The AI Magazine* **24**(4), 120–130.

Ghallab, M. & Laruelle, H. (1994), Representation and control in IxTeT, a temporal planner, *in* 'Proceedings of AIPS-94', pp. 61–67.

Ghallab, M., Nationale, E., Aeronautiques, C., Isi, C. K., Penberthy, S., Smith, D. E., Sun, Y. & Weld, D. (1998), Pddl - the planning domain definition language, Technical report.

Green, N., Garrett, H. & Alan Hoffman, A. (2006), 'Anomaly trends for long-life robotic spacecraft', *Journal of Spacecraft and Rockets* **43**(1), 218–224.

Hart, P., Nilsson, N. & Raphael, B. (1968), 'A formal basis for the heuristic determination of minimum cost paths', *IEEE Transactions on Systems Science and Cybernetics (SSC)* **4**(2), 100–107.

Haugeland, J. (1985), *Artificial intelligence: the very idea*, Massachusetts Institute of Technology, Cambridge, MA, USA.

Hayes-Roth, F. (1985), Rule-based systems, Technical Report 9, New York, NY, USA.

Hoffmann, G. M., Waslander, S. L. & Tomlin, C. J. (2006), Mutual information methods with particle filters for mobile sensor network control, *in* 'Proceedings of the 45th IEEE Conference on Decision and Control (CDC)', San Diego, California, USA.

Holte, R. C. & Hernádvolgyi, I. T. (1999), A space-time tradeoff for memory-based heuristics, *in* 'Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99', AAAI Press, pp. 704–709.

Kim, Y.-W., Rizzoni, G. & Utkin, V. (1998), 'Automotive engine diagnosis and control via nonlinear estimation', *IEEE Control Systems* pp. 84–98.

Koren, I. & Kohavi, Z. (1977), 'Diagnosis of intermittent faults in combinational networks.', *IEEE Trans. Computers* **26**(11), 1154–1158.

Korf, R. (1995), 'Optimal number partitioning', Technical report, also available at ftp://ftp.cs.ucla.edu/tech-report/1995-reports/950062.ps.Z.

Kuhn, L. & de Kleer, J. (2008), An integrated approach to qualitative model-based diagnosis, *in* 'Qualitative Reasoning Workshop (QR 2008)', Boulder, Colorado, USA.

Kuhn, L. & de Kleer, J. (2010), Diagnosis with incomplete models: Diagnosing hidden interaction faults, *in* 'Proceedings of the 2010 AAAI Spring Symposium on Embedded Reasoning'.

Kuhn, L., de Kleer, J. & Liu, J. (2009), Online model-based diagnosis for multiple, intermittent and interaction faults., *in* 'Proceedings of Prognostics and Health Management (PHM2009)'.

Kuhn, L., Price, B., de Kleer, J., Do, M. & Zhou, R. (2008*a*), Heuristic search for target-value path problem, *in* 'Proceedings of the 23rd AAAI Conference on Artificial Intelligence (AAAI-08)'.

Kuhn, L., Price, B., de Kleer, J., Do, M. & Zhou, R. (2008*b*), Pervasive diagnosis: Integration of active diagnosis into production plans, *in* 'Proceedings of the National Conference on Artificial Intelligence (AAAI08)', Chicago, Illinois, USA, pp. 132–139.

Kuhn, L., Price, B., Do, M., Liu, J., Zhou, R., Schmidt, T. & de Kleer, J. (2010), Pervasive daignosis: Integration of planning and diangosis., *in* 'IEEE International Conference on Systems, Man, and Cybernetics'.

Kurzweil, R. (1990), *The age of intelligent machines*, MIT Press, Cambridge, MA, USA.

Liu, J., de Kleer, J., Kuhn, L., Price, B. & Zhou, R. (2008), A unified information criterion for evaluating probe and test selection, *in* 'Proceedings of Prognostics and Health Management (PHM2008)'.

Liu, J., Kuhn, L. & de Kleer, J. (2009), Computationally efficient tiered inference for multiple fault diagnosis., *in* 'Proceedings of Prognostics and Health Management (PHM2009)'.

Liu, J., Reich, J. E. & Zhao, F. (2003), 'Collaborative in-network processing for target tracking', *EURASIP, Journal on Applied Signal Processing* **2003**(4), 378–391.

Marder-Eppstein, E., Berger, E., Foote, T., Gerkey, B. & Konolige, K. (2010), The office marathon: Robust navigation in an indoor office environment, *in* 'International Conference on Robotics and Automation'.

Mauss, J., May, V. & Tatar, M. (2000), Towards model-based engineering: Failure analysis with mds, *in* 'Workshop on Knowledge-Based Systems for Model-Based Engineering, European Conference on AI, ECAI-2000'.

McCarthy, J., Minsky, M., Rochester, N. & Shannon, C. (2006), 'A proposal for the dartmouth summer research project on artificial intelligence, august 31, 1955', *AI Magazine* **27**(4).

McDermott, D. (2000), 'The 1998 ai planning systems competition', *AI Magazine* **21**, 35–55.

McGann, C., Berger, E., Bohren, J., Chitta, S., Gerkey, B., Glaser, S., Marthi, B., Meeussen, W., Pratkanis, T., Marder-Eppstein, E. & Wise, M. (2009), Model-based, hierarchical control of a mobile manipulation platform, *in* 'ICAPS Workshop on Planning and Plan Execution for Real-World Systems', Thessaloniki, Greece.

Muscettola, N., Nayak, P. P., Pell, B. & Williams, B. C. (1998), 'Remote agent: To boldly go where no AI system has gone before', *Artificial Intelligence* **103**(1-2), 5–47.

Natarajan, B. K. (1986), The complexity of fine motion planning, Technical report, Ithaca, NY, USA.

Nau, D., Ghallab, M. & Traverso, P. (2004), *Automated Planning: Theory and Practice*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Nilsson, N. J. (1984), Shakey the robot, Technical Report 323, AI Center, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025.

Nilsson, N. J. (1998), *Artificial Intelligence: A New Synthesis*, Morgan Kaufmann, San Francisco, CA.

Poole, D. (1991), Representing diagnostic knowledge for probabilistic horn abduction, *in* 'International Joint Conference on Artificial Intelligence (IJCAI91)', pp. 1129–1135.

Poole, D., Mackworth, A. & Goebel, R. (1998), *Computational Intelligence: A Logical Approach*, Oxford University Press.

Pravan, G. (2001), Hierarchical model-based diagnosis, *in* 'Proc. International Workshop on Principles of Diagnosis (DX)'.

Preist, C. & Welham, B. (1990), Modelling bridge faults for diagnosis in electronic circuits, *in* 'Proceedings of the First International Workshop on Principles of Diagnosis', Stanford.

Provan, G. & Chen, Y.-L. (1999), Model-based diagnosis and control reconfiguration for discrete event systems: An integrated approach, *in* 'Proceedings of the thirty-eighth Conference on Decision and Control', Phoenix, Arizona, pp. 1762–1768.

Rauch, H. E. (1995), 'Autonomous control reconfiguration', *IEEE Control Systems Magazine* **15**(6), 37–48.

Reiter, R. (1987), 'A theory of diagnosis from first principles', *Artificial Intelligence* **32**(1), 57–96.

Reiter, R. (1992), A theory of diagnosis from first principles, *in* 'Readings in Model-Based Diagnosis', pp. 29–48.

Rich, E. & Knight, K. (1990), *Artificial Intelligence*, McGraw-Hill Higher Education.

Ruml, W., Do, M. B. & Fromherz, M. (2005), On-line planning and scheduling for high-speed manufacturing, *in* 'Proc. of ICAPS-05', pp. 30–39.

Russell, S. J. & Norvig, P. (2009), *Artificial Intelligence: A Modern Approach*, 3rd edn, Prentice Hall.

Russell, S. & Wefald, E. (1991), *Do the Right Thing: Studies in Limited Rationality*, MIT Press.

Schmidt, T., Kuhn, L., Zhou, R., de Kleer, J. & Price, B. (2009), A depth-first approach to target-value search., *in* 'International Symposium on Combinatorial Search (SoCS 2009)'.

Shannon, C. E. (1950), 'Programming a computer for playing chess', *Philos. Mag. (Ser. 7)* **41**, 256–275.

Siddiqi, S. & Huang, J. (2007), Hierarchical diagnosis of multiple faults, *in* 'Proceedings of IJCAI'.

Smith, D. E. & Weld, D. S. (1999), Temporal planning with mutual exclusion reasoning, *in* 'Proc. of IJCAI-99', pp. 326–333.

Smith, D., Frank, J. & Cushing, W. (2008), The anml language, *in* 'Proceedings of ICAPS'.

Srinivas, S. (1994), A probabilistic approach to hierarchical model-based diagnosis, *in* 'Proc. Conference on Uncertainty in AI (UAI)', pp. 538–545.

Thiebuax, S., Cordier, M., Jehl, O. & Krivine, J. (1996), Supply restoration in power distribution systems – a case study in integrating model-based diagnosis and repair planning, *in* 'Prof.8th International Workshop on Principles of Diagnosis (DX)'.

Thrun, S., Montemerlo, M., Dahlkamp, H., Stavens, D., Aron, A., Diebel, J., Fong, P., Gale, J., Halpenny, M., Hoffmann, G., Lau, K., Oakley, C., Palatucci, M., Pratt, V., Stang, P., Strohband, S., Dupont, C., Jendrossek, L.-E., Koelen, C., Markey, C., Rummel, C., van Niekerk, J., Jensen, E., Alessandrini, P., Bradski, G., Davies, B., Ettinger, S., Kaehler, A., Nefian, A. & Mahoney, P. (2006), 'Stanley: The robot that won the darpa grand challenge', *Journal of Field Robotics* **23**(1), 661–692.

Weber, J. & Wotawa, F. (2008), Dependent failures in consistency-based diagnosis, *in* '18th European Conference on Artificial Intelligence (ECAI 2008)', Patras, Greece, pp. 801 – 802.

Williams, B. C. & Nayak, P. P. (1996), A model-based approach to reactive self-configuring systems, *in* 'Proceedings of the National Conference on Artificial Intelligence (AAAI96)', pp. 971–978.

Winston, P. H. (1992), *Artificial intelligence (3rd ed.)*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Winter, S. (2002), Route specifications with a linaer dual graph, *in* 'Symposium on Geospatial Theory'.

Young, T., Arnold, J., Brackey, T., Carr, M., Dwoyer, D., Fogleman, R., Jacobson, R., Kottler, H., Lyman, P. & Maguire, J. (2000), Mars program independent assessment team report, Technical report, Report to the NASA Administrator and to Congress.

Zhong, C. & Li, P. (2000), Bayesian belief network modeling and diagnosis of xerographic systems, *in* 'Proceedings of the ASME Symposium on Controls and Imaging - IMECE', Orlando, Florida.