TECHNISCHE UNIVERSITÄT MÜNCHEN

Institut für Informatik
Lehrstuhl IX, Emmy-Noether Nachwuchsgruppe
"Constraintbasierte Modelle und Algorithmen für Diagnose und Planung"

Model-based Plan Assessment for Autonomous Technical Systems

Paul Maier

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.

Die Dissertation wurde am 23.05.2012 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 22.10.2012 angenommen.

Abstract

Next-generation technical systems, such as household robots and intelligent factories, autonomously plan and schedule actions to achieve high levels of robustness and flexibility. However, because planning and scheduling is computationally hard, it must typically be done off-line using simplified system models, and is unaware of on-line observations and possible component faults and contingencies. Especially in uncertain environments, it therefore becomes important to predict the remaining success probabilities to reach the goals of a partially executed plan, based on behavior models and current observations about the system's (possibly faulty) behavior. This plan assessment problem combines diagnosis and probabilistic reasoning aspects. For example, faulty behavior in a factory plant may jeopardize certain products; as a reaction, the plant could locally re-plan manufacturing steps for these products, avoiding faulty components. However, to make such decisions, it needs a component that diagnoses potential faults and predicts which of the planned products are, as a result, unlikely to be finished.

As its major contributions, this work defines and analyzes this problem and develops two computational approaches to solve it. Both presented methods share the following characteristics: 1) they are adapted to rich, hierarchical model representations used in engineering domains, and 2) they exploit efficient, off-the-shelf implementations of generic solution algorithms. Since the plan assessment problem straddles probabilistic reasoning (predicting goal probabilities) and model-based diagnosis (finding failure causes), one approach focuses on techniques from probabilistic reasoning, and the other on techniques from model-based diagnosis.

We developed and implemented a plan assessment library capable of inferring the required information using different solvers that may be flexibly chosen. We used it to compare the two approaches using the state-of-the-art probabilistic inference tool ACE for the probabilistic reasoning approach, and the modified constraint optimizer TOULBAR2 for the model-based diagnosis approach. The experimental results on our example scenarios are promising: both approaches perform equally well on this problem in terms of memory and CPU usage, indicating that tools from both areas are viable choices. Overall, we argue that this work provides an important component for decision support that is needed for creating reliable autonomous systems.

Acknowledgements

That the journey of a PhD student towards its degree is possible at all, and fun and exciting and an overall satisfying experience is really because of the many great people that support said student on its way. That student is me and I feel overwhelmingly grateful to all these people. This piece of text won't do them justice. I will try anyway.

Martin Sachenbacher is my adviser, and his efforts in supporting me are practically uncountable. Making possible fascinating research visits or encouraging me to present my work on conferences (in fact very early on) are only two of many more examples. He taught me, motivated me and occasionally pushed me in the right directions whenever I was stuck. I clearly remember the many times I was unsure about my work before a meeting with him, and came out of the meeting re-motivated. Thank you, Martin, for your patience and tireless support.

I want to take this opportunity and thank my other two assessors, Prof. Javier Esparza and Prof. Michael Beetz. Your suggestions have greatly improved this thesis.

Then there's the office gang, Andreas Artmeier, Andreea Stegaru, Julian Haselmayr, Michael Geppert, Daniel Quinger (people from our group), Michael Esser, Alessandro Fraracci and Tesfaye Regassa. Thank you guys for countless discussions, your support and generally for making our office a really nice place to be. Without you, I would've been less motivated to come to the office every day. Special thanks go to Andreas, Julian and Michael Esser, who probably caught the biggest chunk of my ramblings, crazy ideas and urges to discuss some fine points of the topics I was working on. Furthermore, I would like to thank our students and student assistants for their great work, especially Gregor Wylezich and Jian Wang.

I was extremely lucky to actually have two mentors during the final stages of my thesis. Alexandra Kirsch is a research group leader at our chair, and she agreed to be my TUM graduate school mentor. Alex, your patient listening, motivation, your insights and, not to the least part, your calming words were incredibly helpful during these last months. Thank you.

I also would like to thank the people in Alex' group, Christina Lichtenthäler, Michael Karg and Thibault Kruse. Not only have you been constantly open to debating the weirdest things, you have in general been a very great company. Special thanks to Michael for his images of PR2.

During my research I was lucky to work together with some great people from our chair, especially Thomas Rühr, Dominik Jain and Lukas Kuhn from Michael Beetz' Intelligent Autonomous Systems (IAS) group. Thank you guys, working with you has been an invaluable experience.

Thomas, Lukas, Martin and me developed the idea for plan assessment in winter 2008/2009. The long discussions I later had with Lukas when I visited him at Palo Alto Research Center (PARC), about plan assessment and what it should be all about, greatly helped me in clarifying my ideas and my conceptions. Thank you Lukas, that was a great time.

At this point I also would like to thank all the other people at PARC who made my stay a very pleasant one. In particular, I would like to thank Johan de Kleer, who gave me the opportunity of coming to PARC and presenting my work there.

Dominik greatly helped me understand the fascinating world of probabilistic reasoning. He constantly questioned my approaches and methods (sometimes quite bluntly), and thus forced me to constantly rethink and refine. It ultimately led to some especially nice co-authored work. Thank you, Dominik, I greatly enjoyed working with you.

I would like to thank Natarajan Shankar of the Computer Science Laboratory (CSL) at Stanford Research Institute International, who together with Martin gave me the opportunity to visit his group at CSL. Also, I feel deeply grateful to all the people I met at CSL, especially Bruno Dutertre, John Rushby, Ashish Gehani, Ashish Tiwari, Hassan Saïdi, Sam Owre, Minyoung Kim, Lori Truitt and Wilfried Steiner. Not only did I gain a lot of insight in very useful techniques such as satisfiability modulo theory, model checking and theorem proving, but the many discussions and presentations allowed me to put together pieces from math, logic and computer science in a whole new way.

I was supported in numerous ways by many people from the TUM departments of engineering and control. In particular, I would like to thank Prof. Vogel-Heuser for letting me present my work in her group, and Jens Folmer, Daniel Schütz, Martin Merz and Christoph Legat for helpful discussions and presentations. Special thanks go to Martin Merz, for helping me making my first steps in Matlab Simulink, and Christoph Legat, for providing insights into manufacturing as done by a well-known big company. Then I would like to thank the people of the Cognitive Mashineshop project, in particular Christoph Ertelt, Martin Ostgathe, Thomas Gmeiner, Dejan Pangercic and again, Thomas Rühr, for helping me understand the intricacies of the cognitive factory. Thomas, you helped me a lot in understanding the cognitive factory, planning and scheduling, thank you. Finally, I would like to thank Prof. Olaf Stursberg and his group, especially Matthias Althoff and Tina Paschedag.

I also would like to thank Mark Chavira and Adnan Darwiche for providing version 3.0 of their software Ace.

Of course I would like to thank all my friends, who greatly supported me throughout this whole project. I bet I have not always been easy to deal with during these times. An extra big thank you to those who tried to wrap their heads around the brick called "thesis" I threw at them. Your corrections and suggestions made a big difference, thank you guys.

Finally, I would like to thank my family, Kathrin, Bernd, Thomas and Johannes Maier, who have always been a haven of relaxation for me, and my wife Kerstin, the love of my life, who bore with me through every single up and down on my journey.

Contents

1. Introduction

Across many domains of technical systems, research and industry develop forms of autonomy to achieve flexible and adaptive, yet robust and reliable system behavior. Satellites, for example, must operate for long periods in harsh environments without the possibility of human intervention [132, 80]. Driverless cars are being developed that require autonomous navigation [84, 5]. Finally, automated manufacturing needs to keep up with quickly changing market demands by autonomously producing large varieties of products at comparably small quantities [31, 9]. In particular, the cluster of excellence Cognition for Technical Systems (CoTeSys) [15] explores autonomy in form of human-like "cognitive capabilities", which are being applied within the *cognitive factory* [166].

A technical system acts with a certain degree of *autonomy* if it takes decisions without human intervention. These decisions require information about the system's state and potential outcomes of its current operation. To understand what we mean by that, imagine the following example of an autonomous manufacturing plant. The plant consists of machining and assembly stations and is supposed to produce small lots of individual products. This is known as *mass-customization*. In contrast to mass-production of few specific products, the large variety of products doesn't allow to guarantee successful execution for each production plan beforehand, for example during the design of the system. Therefore, instead of being pre-programmed, the plant dynamically plans the production on demand. A plan details, for each product, when to process parts at which station. The goals of such a plan are to finish each product.

Scenarios such as these often require planning that quickly becomes too computationally complex to be done during runtime. For that reason planning is done offline, for example during the night. Later, the plans are executed online, for example during the day. Autonomous behavior is now limited to dynamically coping with contingencies during execution of the plans. In such cases the system autonomously chooses an appropriate reaction. However, this requires information about the *uncertain outcome of the plant's behavior*.

A contingency might be a broken machining cutter. An appropriate reaction could be a plan modification for a jeopardized product, inexpensive enough to be computed

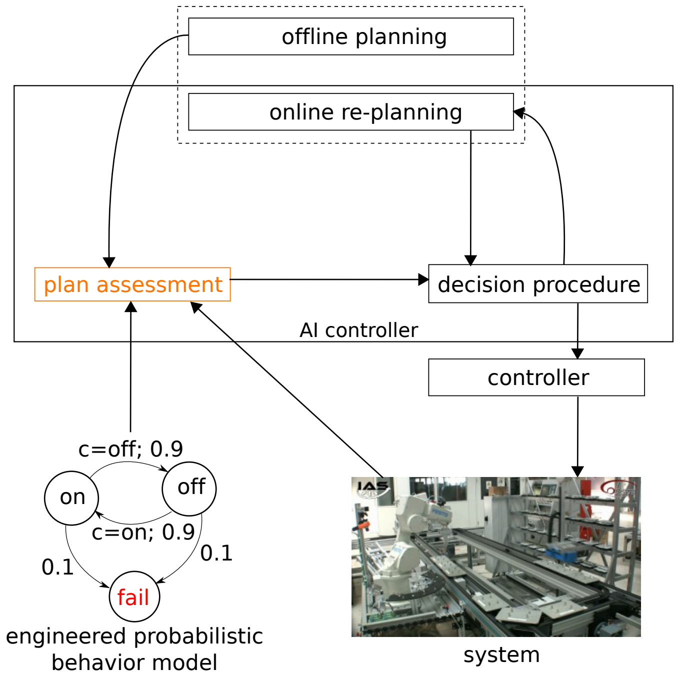


Figure 1.1.: Architecture of a system, for example a factory plant, that executes offline generated plans. Occasionally, the system may have to react autonomously to unforeseen events. This work is concerned with the development of a plan assessment component that provides essential information for these autonomous decisions.

online. To make the right decision, the plant must answer the following question: Which products now have low probability of being finished and what are potential causes for these jeopardized products?

In this thesis, we address the computational problem of how to automatically answer this question using three information sources: The plan, available sensor data and a model of the system specifying its behavior. We call this *plan assessment*.

We develop approaches for plan assessment that regard the unique context of engineering tool-chains and that build on generic algorithms that are the result of decades of artificial intelligence research. The topic of this thesis is therefore to compute the required answers using off-the-shelf implementations of the said generic algorithms and a system model encoded with a modern, expressive formal language.

Plan assessment depends on existing techniques necessary for autonomous behavior, most important planning [64, 125, 13, 17, 139, 163]. Figure 1.1 illustrates these dependencies with a potential architecture for the described example system.

1.1. Model-Based Reasoning for Autonomous Behavior

To achieve the kind of autonomous behavior illustrated above, researchers study problems and develop methods which are often subsumed as *model-based reasoning* [68, 163, 159, 77, 11, 86]. In particular, *model-based diagnosis* [77, 146, 128, 163, 103] addresses problems of identifying causes of faulty behavior in technical systems.

Autonomous behavior is often characterized as being able to handle unknown, unexpected or not-before-seen events. “Unknown” of course does not mean that a system has to deal with situations it has no knowledge about whatsoever. Rather, it means to avoid the naive approach of explicitly encoding sensor-action mappings for all relevant events, because typically there is a huge number of such events. Instead, we wish to exploit what we know in general about any relevant situation the system may encounter and what we can observe about the system’s current situation. In model-based reasoning, this is achieved by compactly representing the general knowledge in a *model* and by combining it with available observations, typically gathered from sensors. The model may capture knowledge about the system and/or the environment. However, in this work, we focus on cases where only a system model is needed.

Model-based diagnosis tries to identify faulty components using a model of system components and “unexpected” behavior. In its classical form, the model is a set of logical formulas describing the static connections and interactions of components working normally. Observations are encoded logically as well. They are “unexpected” if they are inconsistent with the model formulas, that is the normal behavior of the system. Faulty components can now be identified by regaining consistency: we try to find the smallest subset of components that, when assumed to be malfunctioning, restore consistency. One can think of it as the faulty components “explaining” the inconsistency.

Current diagnosis approaches improve on this method by taking into account failure probabilities of components [146] and by modeling the evolution of the system over time [108, 128]. The former reduces the number of component subsets to consider, the latter allows to identify faults even if their symptoms appear only later on [108]. The latter works with potential system behaviors instead of subsets of components.

Model-based diagnosis takes a more global view in asking for faulty components that explain unexpected observations. For plan assessment, we are also interested in the probability of a plan successfully achieving some goal. For this we have to take a more local view, asking for the influence of external events and internal behavior on one specific part of the system. This corresponds to a general problem studied in *probabilistic reasoning*

[21, 135], namely to compute the marginal probability of a single event [135, Chapter 4.5, p. 223].

Following this line of reasoning, we see plan assessment as an extension of model-based diagnosis that combines it with elements from probabilistic reasoning. The problem is thus relevant for both areas. Since the areas overlap, often taking different views on similar problems, both offer promising building blocks for solution approaches.

1.2. Problem Statement: Plan Assessment

This work considers scenarios of autonomous behavior in a certain class of technical systems that give rise to a unique computational problem: the *plan assessment problem*. What is the nature of these scenarios?

On the one hand, researchers identify some form of autonomous behavior to be essential for future technical systems. Even for rigid systems such as factory plants autonomy can become necessary: In simple cost-benefit terms, it will likely cost less to augment a factory plant with autonomous behavior (and deal with the ensuing complexity) than trying to remove all uncertainties, all possibilities for failures and contingencies beforehand. On the other hand, in our view it is valid to assume that technical systems of this kind, such as cars, printers, satellites or factory plants, although required to be flexible, will still mostly face rigid environments.

We thus investigate a class of technical systems that will, on the one hand, be rigidly designed with mostly deterministic behavior, while on the other hand a certain degree of autonomy allows these systems to efficiently deal with unexpected contingencies. A typical scenario then confronts us with enough uncertainty such that we have to use dynamic planning, yet the mostly deterministic behavior of our rigid system allows to generate long term plans during an offline phase. This leads to settings were a system will be operated according to pre-planned steps most of the time, while occasional contingencies will be handled by a decision procedure with appropriate reactions. Sensor signals will provide information about the current state of the system. We cannot expect this information to be exhaustive, since in many cases this would require a prohibitively large number of sensors.

From these considerations we derive three properties that characterize the scenarios that we consider:

1. A technical system shall be operated that is rigidly designed yet complex enough that some uncertainties remain.

2. The system executes pre-planned operation steps and receives observations during execution, for example through sensor measurements. Occasionally the system must flexibly react to contingencies, such as failing components. It has to automatically choose among appropriate reactions while execution is still underway.

3. Execution of the operation steps should achieve a set of explicitly defined and represented goals.

These three properties correspond to three formal elements:

1. A system model M with a large deterministic and relatively small probabilistic part, which captures potential behaviors of the system to be controlled. We will call the potential behaviors the *trajectories* of a system.

2. A plan $\mathcal{P}$ that has been executed up to current time t and observations $\mathbf{o}^{0:t}$ given for the past time points $0, \ldots, t$.

3. A set of goals $\{G_1, \ldots, G_i, \ldots, G_n\}$. We will denote this set with the abbreviated notation $\{G_i\}$.

Now we can give a first definition of the plan assessment problem:

> Compute, based on a system model M, a plan $\mathcal{P}$ and observations $\mathbf{o}^{0:t}$ the most probable diagnosis of this system as well as the probability, for each i, that the goal G_i will be achieved.

In this work we develop solution approaches to this problem that revolve around generating potential trajectories of system behavior along with their probabilities. The probability of a single trajectory represents how probable it is that this trajectory models the real behavior. The most probable trajectory is considered the most probable diagnosis, since it contains information about which faults have most probably occurred (if any). The probability of reaching a goal is computed by summing over probabilities of goal-achieving trajectories and normalizing by the sum over goal-achieving and goal-violating trajectories.

The latter requires, for exact computation, *all* behaviors with non-zero probability. This is not an easy task: Consider a comparably simple model of a manufacturing plant with 2 stations and 3 products, where the 2 stations are modeled with (non-deterministic) finite state machines with 5 states and 12 transitions each, and the products with 2 states and 3 transitions each. The product of these 5 state machines has $2 \times 2 \times 2 \times 5 \times 5 = 200$ states and $3 \times 3 \times 3 \times 12 \times 12 = 3888$ transitions, which yields $\frac{3888}{200} \approx 19$ possible transitions per state. This in turn yields 19^N possible behaviors for N time steps, which means that after only 5 time steps we have to deal with millions of potential behaviors, many of them with zero probability. But even the number of non-zero probabilities might be too large to handle. Therefore we also propose an approach that approximates the success probabilities for goals $\{G_i\}$ based on a reduced set of trajectories.

We make some further important assumptions throughout this work:

- We assume discrete time steps in our models. This is a very common assumption in model-based reasoning to allow the use of symbolic reasoning methods.
- We use time synchronous modeling and thus restrict ourselves to synchronized system components.
- We assume pre-processing steps to be in place that convert continuous sensor signals into abstract, discrete observations.

1.3. Two Application Examples for Plan Assessment

We have already briefly mentioned the example of an autonomous manufacturing plant. These sorts of plants are being developed to realize automated production with high product variability and at the same time only small lot sizes, maybe even one-of-a-kind production [31, 9, 107, 63]. A particular instance is the cognitive factory [166] within the cluster of excellence CoTeSys. In this work we consider an example where this factory produces two different products, toy mazes and toy robot arms. The mazes may additionally vary. The products are processed at machining stations to cut, for example, a labyrinth groove into an alloy block for the maze, and at an assembly station. During cutting, the cutter may break and then subsequently lead to flawed products. This is only detected later on indirectly when sensors at the assembly station give abnormal signals. Given the sensor information, we would like to know how each product is affected and what the most probable faults are. Formulated as a task, we want to compute the products remaining success probabilities and ideally identify the broken cutter as fault.

While we think that plan assessment is more interesting for scenarios such as the one described above, it is a general problem not confined to autonomous manufacturing. Within CoTeSys, another area of research focuses on autonomous household robots. Robotic assistants are being developed that shall support humans in their everyday household activities [14], for example setting the table. The task in such a scenario is to bring plates, cups, cutlery etc. to the table and place them there, such that the table is set for each person sharing the meal. Problems might occur with robot components, for instance its arms, or with complex visual recognition capabilities. This can lead to a failed grasping of a plate, for example, which then drops to the floor. Given available sensor information we want to deduce how probable it is that the other items, like the cups, can still be placed on the table. And again, we also would like to know the cause for the problems.

1.4. Expressive Models and Off-the-Shelf Tools as Ingredients for Solution Approaches

Computational solution approaches to plan assessment will require two essential ingredients: a way to represent models of the considered system and algorithms to compute most probable diagnoses and success probabilities. We develop approaches that try to regard the toolset that engineers use for the design and development of those systems. The reason is simple: First of, these tools already address many problems when it comes to modeling systems. And second, approaches that are adapted to the engineering environment are more likely to be accepted.

State-of-the-art tools for the development of complex technical systems rely on expressive description languages. In particular, these languages allow to represent system behavior in a hierarchical fashion. Matlab® Stateflow®[1] is an example for an industry standard language. It is based on Statecharts [78]. It is used to program embedded controllers for complex systems using hierarchical finite state machines, which can be compiled to embedded control code. With next-generation languages such as Modelica [56] researchers try to expand the possibilities of how to use the resulting models. For example, models may be reused for reasoning tasks during the *operation* of the system [163, 69, 106], such as estimating its hidden state from partial observations. The vision is to have, on the one hand, system independent embedded algorithms for tasks such as control, and on the other hand, system specific models as input to these algorithms. A model, in our case, is an explicit formal description of a system's structure, its capabilities, its behavior and/or its potential faults. Specifically, the authors of [163] developed a formalism called probabilistic hierarchical constraint automata (PHCA) that allows to describe *uncertain* system behavior with hidden states and probabilistic transitions between them, while also allowing hierarchical structures. This work uses PHCA to describe system models that encode a system's behavior as well as goal-achieving and goal-violating states. Figure 1.2 shows an example of such a PHCA.

[1]`http://www.mathworks.com/products/stateflow/`

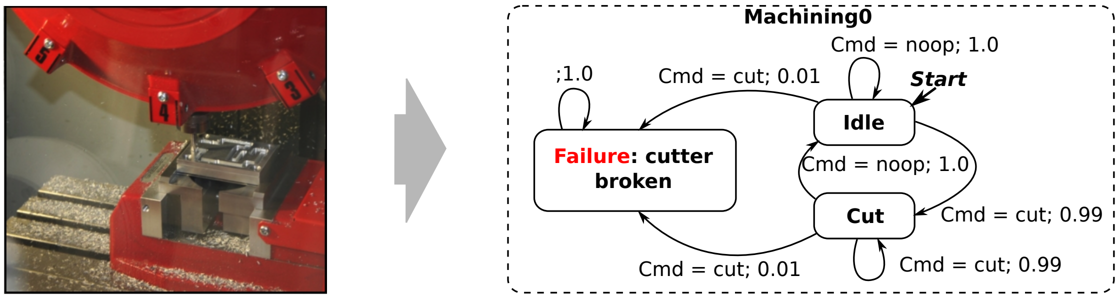


Figure 1.2.: A machining station (left) and a model for its behavior in form of a probabilistic hierarchical constraint automaton (right).

There are many fast reasoning algorithms for specific tasks such as estimating the hidden model state of a system from observations. However, algorithms are constantly being improved, novel methods being developed. Incorporating an improvement manually in one's own algorithm can become tedious. A different approach is to use off-the-shelf implementations of generic algorithms, translating the problem description at hand (based on system models) to the interface language of this implementation. We believe this to be a better approach, since the additional separation of concerns between the specific problem domain and generic algorithms allows us to exploit novel developments in the general domain quickly. This resembles the situation in software engineering, where code optimization is mostly done in a general fashion by the compiler rather than by the programmer for a specific application. Typically, the communities behind these generic algorithms are quite large and therefore make quicker advances. Finally, domain specific knowledge could still be introduced, if necessary, by developing specific heuristics for these algorithms.

In our choice of generic algorithms we focus on the field of artificial intelligence. Since we pose plan assessment as a problem that arises in supporting autonomous decisions, it is an artificial intelligence problem. More specifically, as we argued, we see plan assessment at the intersection of model-based diagnosis [77] (computing most probable diagnoses) and probabilistic reasoning [21, 135] (computing success probabilities).

Generic problem solving for model-based diagnosis is often done using constraint processing [51], in particular constraint *optimization* [149]. The latter uses networks of local objective functions, the constraints, to describe a problem, such as finding most probable diagnoses for a system. We develop an approach that uses constraint optimization as algorithmic backend.

In probabilistic reasoning, methods often involve Bayesian network models, which represent a system as a set of variables and their conditional probability distributions.

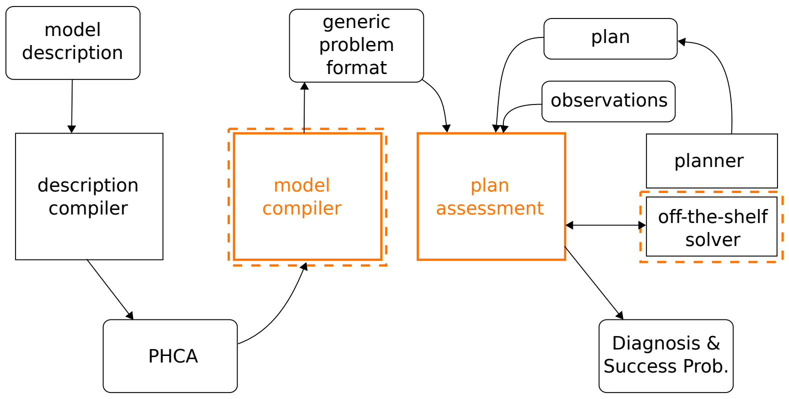


Figure 1.3.: Simplified schema of our prototypical tool chain implemented for this work. Highlighted boxes indicate own implementations, dashed lines modified external tools.

During the last decade, powerful extensions have been developed that combine Bayesian networks with first-order logic. Our second approach exploits such a framework, called Bayesian logic networks [95]. This leverages a wide range of generic probabilistic algorithms. To test this approach, we have chosen an algorithm that preprocesses the probabilistic model such that the complexity of subsequent probabilistic inference is greatly reduced [46]. It accepts Bayesian networks as input, which are naturally included in the more general Bayesian logic networks.

For this work we developed a prototypical tool chain to test our approaches. Figure 1.3 shows its structure, linking key components (boxes) and data (rounded boxes). An initial model description is compiled into a PHCA model, which in turn is then translated to a generic problem description that the solving backend of our choice understands. The plan assessment component then uses this description together with the plan and available observations to generate the desired diagnoses and success probabilities. The implementations done for this work are highlighted, the dashed line indicates an existing tool modified for our purposes. This diagram is simplified, later sections will explain our implementations in greater detail.

1.5. Contributions

This work develops a model-based reasoning capability that provides information for an AI decision procedure in order to support autonomous decisions. It combines the more global view of model-based diagnosis, which asks questions such as “which components have to be assumed faulty to explain the observations?”, with a more local view often found in probabilistic reasoning, which asks "Given the current observations, how probable is it that a goal state is still reached by executing the planned actions?".

The two key contributions of this thesis are the following:

- It introduces and formalizes plan assessment as a model-based reasoning problem for probabilistic hierarchical constraint automata (PHCA) as an extension to model-based diagnosis. It demarcates it against the state of the art and shows relationships to related problems such as state estimation.
- It develops two different solution approaches: One based on model-based diagnosis methods and one based on probabilistic reasoning. Both leverage existing off-the-shelf tools by translating PHCA into generic problem descriptions that these tools accept as input. Part of the latter is a novel translation from PHCA models to Bayesian logic network models, a generalization of Bayesian networks. We test this translation in practice and provide theoretical correctness results.

The thesis makes a number of additional contributions:

- It evaluates and compares the two solution approaches. To obtain the necessary results, we developed a tool chain for automatic model translation to low-level languages used by off-the-shelf solvers for combinatorial optimization and probabilistic reasoning. The chain integrates existing tools with our own implementations to realize prototypes of the presented solution approaches.
- It develops three extensions of the presented approaches.
  1. The model-based diagnosis approach is extended with a receding horizon approach based on filtering techniques. It is motivated by the fact that analyzing the complete time horizon at once quickly becomes intractable when increasing the number of time steps.
  2. The probabilistic reasoning approach is extended with an approach to compute stochastic error bounds when using approximate sampling techniques. Sampling is a class of very general and widely used approximation techniques. However, implementations typically use hard iteration bounds based on user experience to stop the calculation. The stochastic error bound can provide an additional criterion based on the quality of the solution.
  3. The model-based diagnosis approach to plan assessment is advanced towards hybrid discrete/continuous models. Often, system behavior is easier to model using a mixture of discrete elements (for example states) and continuous

elements (such as differential equations). The thesis introduces a hybrid extension of PHCA called Hybrid PHCA (HyPHCA) and shows how these can be conservatively abstracted to purely discrete models. It extends the mentioned tool chain with an implementation of this translation and formulates a theoretical correctness property for it. The translation is used for experiments that demonstrate the feasibility of this approach.

1.6. Structure of the Thesis

The remaining text of this thesis is structured as follows. Chapter 2 details the described examples for plan assessment. Chapter 3 provides the technical background: It explains how to model systems, how to formulate generic problems for these models and the generic algorithms that we use. Also, it introduces PHCA and Bayesian logic networks.

Chapters 4 and 5 constitute the two major contributions of this work. Chapter 4 formally introduces the plan assessment problem based on the PHCA formalism, analyzes its role in autonomous manufacturing and puts it in context of related problems from related fields such as artificial intelligence, probabilistic reasoning and control theory. Chapter 5 develops the two mentioned model-based algorithmic approaches to the plan assessment problem. Results of evaluations and a comparison of these approaches are presented in chapter 6. Chapter 7 introduces extensions of our plan assessment approaches: the receding horizon method for the model-based diagnosis approach, the approach to compute stochastic error bounds for the probabilistic reasoning approach and the extension of plan assessment towards hybrid discrete/continuous models.

Chapter 8 presents related work. Most of the literature related to this thesis is treated in context of the most relevant parts of the text. The literature presented in chapter 8, however, is better treated in a separate chapter. Finally, chapter 9 concludes this work by summarizing its contributions and results as well as discussing open ends and opportunities for future work.

2. Application Examples

2.1. Assuring Plan Success in Manufacturing and Assembly

In this work we consider as our main example a somewhat idealized manufacturing plant that can automatically schedule operations for products based on abstract product descriptions. The latter is known as *design-to-fabrication* [57], a major intelligent capability of the so-called *cognitive factory* [9]. The cognitive factory is a demonstrator within the cluster of excellence Cognition for Technical Systems (CoTeSys) [15], but also a lead concept for research projects in this cluster that develop such capabilities. The overall idea is to augment technical systems like factories with capabilities akin to human cognition, such as reasoning over potential outcomes of planned operation steps. CoTeSys has an actual test-bed meant for research evaluation and demonstration of concepts developed for the cognitive factory. We derive our realistic examples and models from this test-bed.

2.1.1. Autonomous Behavior and the State of the Art in Factory Automation

Before we describe our main example in detail, we cast a look at the role of autonomous behavior in the state of the art in factory automation. For a long time research efforts have been made to develop more flexible and adaptable manufacturing systems, as an answer to the steadily increasing market dynamics and in order to stay competitive. A reoccurring theme is that autonomous or semi-autonomous behavior is proposed as a means to achieve the required adaptivity and flexibility.

In the mid and late nineties of the 20th century many concepts emerged. A prominent example is *reconfigurable manufacturing systems* [104]. It focuses on dynamically adapting capacity and functionality of a plant to market needs, with the goal to produce exactly as much as needed and exactly what is needed. Another set of approaches are known as *agent-based manufacturing* [130, 165]. These approaches represent the characteristic elements of production, such as products, parts or machines, as agents that act and interact autonomously to achieve (global and local) goals. According to [130] agent-based design principles are especially suited because of their distributed nature, which

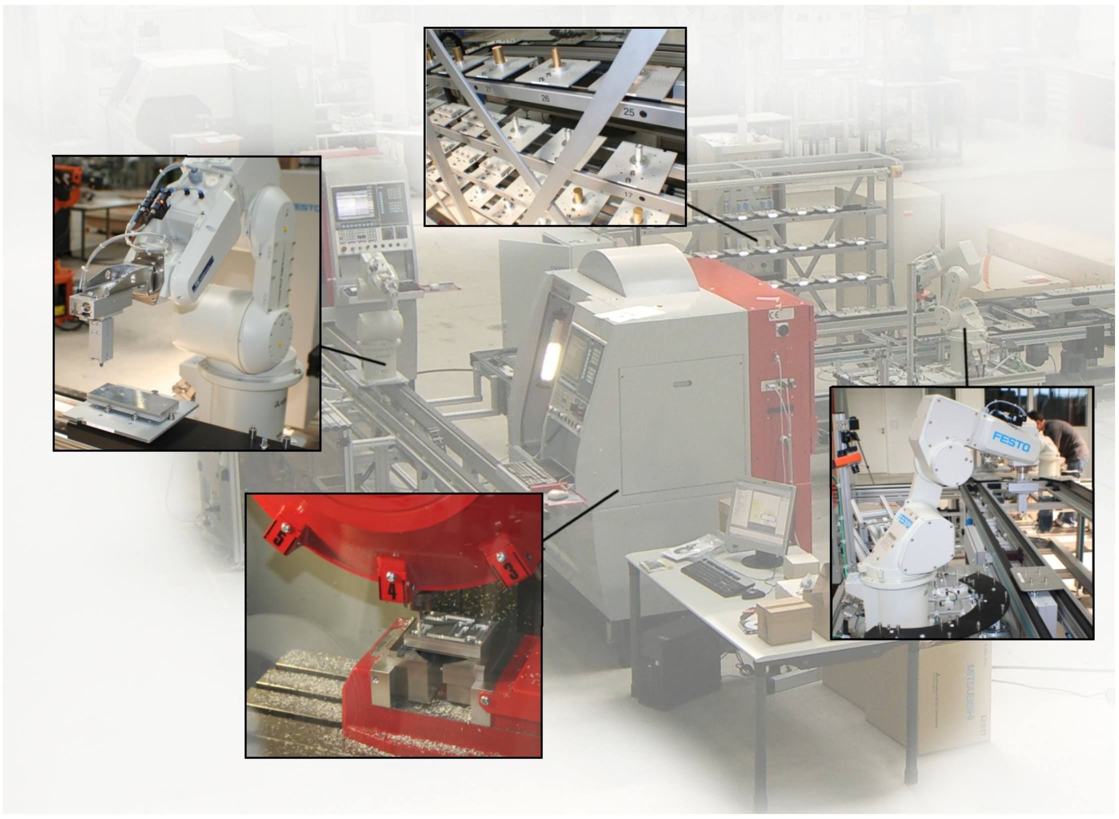


Figure 2.1.: The test-bed for the cognitive factory, an iCIM3000-based Festo flexible manufacturing system, which is part of the TUM excellence cluster Cognition for Technical Systems (CoTeSys). © Prof. Shea, TUM.

nicely corresponds with the distributed design of factory plants, and the agents' intrinsic autonomy, which allows to adapt to different situations. In contrast to reconfigurable manufacturing systems, these approaches also consider adaptation to unpredictable fault conditions.

Two prominent agent-based approaches are *biological manufacturing* [155] and *holonic manufacturing* [32]. Biological manufacturing tries to implement adaptive behavior by mimicking the biological process to grow "goods" with interacting agents representing product parts and different factory stations. Holonic manufacturing specifically addresses the problem that distributed agents are hard to control on a global level. It tries to reconcile distributed control with the hierarchical, centralized control in classical automation.

A more recent approach to distributed manufacturing was developed within the EU project PABADIS'PROMISE [107, 63]. In particular, the problems of robust planning for new product variations and robust plan execution in the face of plant faults are being addressed. Other works within this project deal with flexible production [154] and scheduling for production [30].

Today, currently running national and international research programs keep working towards adaptive, fault tolerant automation with the help of autonomous behavior, aiming for automated mass-customization and one-of-a-kind manufacturing [58, 59, 40]. In Germany, the "excellence initiative" is of special interest. Part of this initiative are so-called "clusters of excellence", topic-driven collaborations across institutional and domain boundaries. One of them is "Integrative Production Technology for High-Wage Countries", located in Aachen [31]. It essentially focusses on the question how to make it cheaper to build systems that produce customized products with required quantities. Thereby, it is in line with the goals of previous research efforts. However, it focusses less on autonomous behavior as a means to achieve this goal. We already mentioned the second cluster that is concerned with this goal, namely CoTeSys. In contrast to the former cluster, the study of capabilities that allow autonomous behavior are at the core of CoTeSys. Addressing technical systems in general, the goal is to create "cognitive systems" that will "know what they are doing" [34]. The idea is to learn from humans and animals how to implement "cognitive capabilities" and the ability for "cognitive control". Key technologies are again planning, but also learning techniques. "Cognitive capabilities" shall achieve the level of flexibility, reliability and efficiency necessary for automated production of even small numbers of sophisticated, individual products. Demonstrators are being developed to investigate and test such capabilities. One of them is the already mentioned cognitive factory [9].

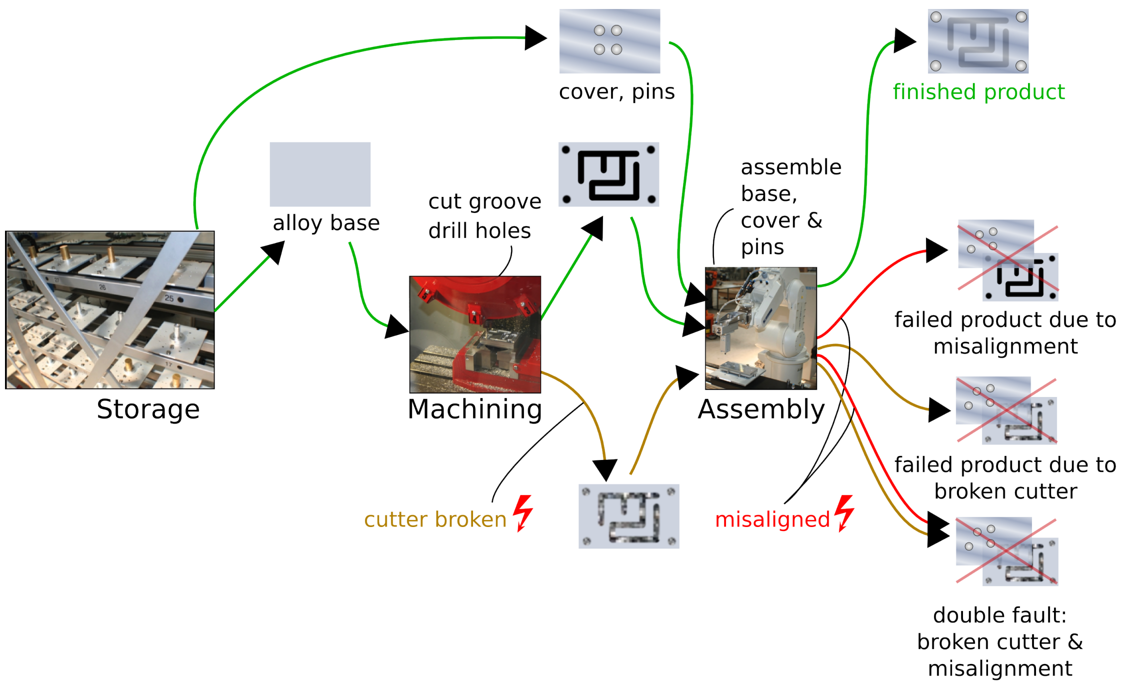


Figure 2.2.: Illustration of the maze production process with possible faults of a broken cutter and a misaligned gripper of the assembly station.

Automated planning, the classical ingredient for autonomous behavior, is studied intensely in the mentioned works. However, autonomy also means to adapt and change plans when necessary. To be able to decide whether and how to do that requires information. This leads to the problem of automatically retrieving this information from already generated plans and current information about the plant. We have seen that this problem and approaches to solve it computationally are the topic of this thesis. In this, the thesis follows and builds on research that leverages artificial intelligence for autonomous behavior in technical systems, specifically planning [163, 106, 57, 10, 156], model-based diagnosis [77, 163, 145, 106, 10, 60] and probabilistic reasoning [135, 72, 96, 91].

We do not address the question of distributed computing in this work. However, the computational approaches in this work could possibly be embedded in a component that serves as support for autonomous decisions of agents in an agent-based architecture, for example.

2.1.2. Manufacturing and Assembly Scenarios in a Cognitive Factory

The cognitive factory test-bed is an iCIM3000-based Festo flexible manufacturing system and consists of conveyor transports, storage, machining and assembly. It serves as the basis for our hypothetical example scenarios in which manufacturing schedules for two different products, toy mazes (figure 2.3) and toy robot arms (figure 2.4), are automatically synthesized. The "CIM" in iCIM stands for computer-integrated manufacturing. The iCIM3000 package from Festo[1] is primarily meant for education.

A maze consists of an alloy base plate, a small metal ball and an acrylic glass cover fixed by metal pins. It is manufactured by first cutting the labyrinth groove into the maze base-plate and drilling the fixation holes, putting the ball into the labyrinth, putting the glass cover onto the base plate and finally pushing the pins in place to fixate it. Figure 2.2 illustrates this process.

The robot product is still in a conceptual stage. It consists of alloy brackets and servos and is manufactured by first machining the brackets and then assembling brackets and servos. The maze is a real-world demonstration product, which the actual test-bed can produce.

In our scenarios the factory plant is controlled by a component that exhibits AI capabilities such as automatic planning and scheduling. We name that component "AI controller", although it isn't necessarily a controller which is embedded in the machine and has to fulfill hard real-time requirements like a programmable logic controller (PLC). Although the long-term goal is to bring AI methods even to "low-level" PLCs, currently the more likely scenario is, due to the computationally complex tasks, that an AI controller would be a piece of software running on standard industrial PC hardware.

A quick note about planning and scheduling at this point. Planning means to find a sequence of actions to reach given goals, whereas scheduling means to assign resources to actions. The two problems are connected, and thus often considered at the same time. In this work, we denote the whole process by "planning" and simply take scheduling as a part of that. Consequently, we consider a schedule as a special plan. A plan, in turn, is simply a sequence of abstract elements, which take a concrete form depending on the application.

Depending on incoming sensor data, the AI controller has to choose appropriate actions, for example:

1. Proceed with normal operation (maybe despite potential contingencies).
2. Ask the scheduling sub-component for a local re-scheduling, e.g.
    a) dispatching a product from a faulty machining station to a different station with similar capabilities.

[1]`http://www.festo-didactic.com/de-de/lernsysteme/feldbustechnik/icim-3000-die-komplette-mit-potenzial.htm` (09.2011)

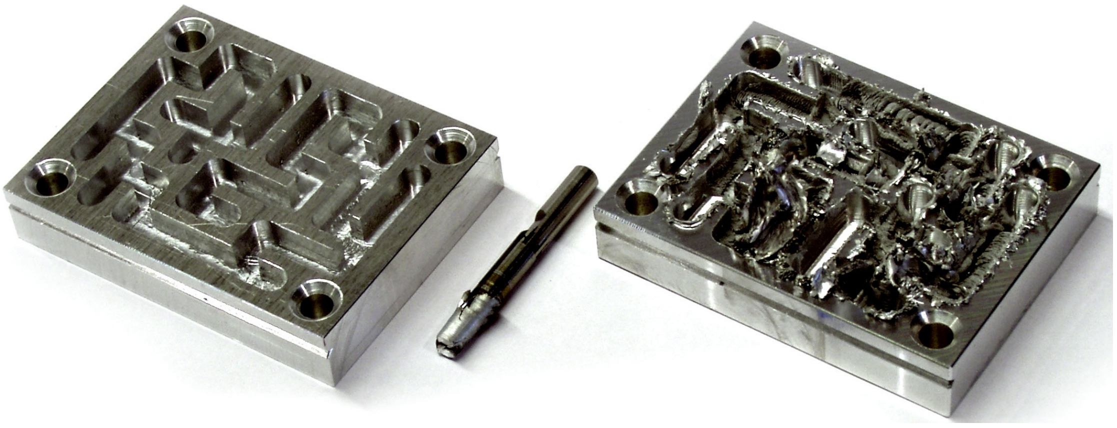


Figure 2.3.: Alloy base plate of the maze product. A regular cut maze groove (left) and a maze groove being cut with a broken cutter (right). © Prof. Shea, TUM.

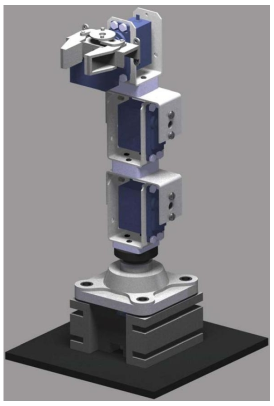


Figure 2.4.: The (virtual) robot arm product. © Prof. Shea, TUM.

   b) decide to have the automated assembly step done manually later on.

3. Initiate information gathering actions.
4. Call a human technician to resolve complex contingencies it cannot deal with.
5. Emergency stop the entire plant.

2.1.3. An Example Plan Assessment Scenario

Consider the following scenario where two mazes and one robot arm are being manufactured. Their schedule is shown in figure 2.5. The mazes are termed Maze0 and Maze1, the robot arm Robot0. Three stations are available to work the products, two machining stations termed Machining0 and Machining1 and an assembly station termed Assembly. We assume that mazes can be individualized with respect to the groove being cut into the base plate, ranging from simple to more complex. In our scenario, Maze1 has a more complex groove and thus requires more machining (both machining stations are equally fast).

In our example plant, assembly stations have a force sensor that allows to measure the force being applied while pushing pins into the holes of maze base plates. At time $t_{\text{alarm}} = 3$ the force sensor triggers an alarm, indicating that too much force was applied. What is most likely to have happened and what should be done?

On its route through the factory a product might become flawed as a result of being worked by faulty stations. Machining stations are suspicious candidates because their cutter might break during operation. A blunt or broken cutter severely damages maze products (see figure 2.3). We assume that machining stations not only cut grooves but also drill the holes for the pins. Therefore, broken cutters might also damage these holes. The damage, however, can only be detected later on: If an assembly station tries to push pins into damaged holes, too much force is applied and an alarm is triggered.

Sometimes sensor signals can be ambiguous, for example if they have unintended additional physical causes. In our example scenario, a misalignment of the gripper holding the pin and the base plate's hole might also lead to the force alarm being triggered. For the pin to be pushed in place a very accurate fit of pin and hole is necessary. That fit is easily lost if, e.g., the assembly station's calibration degrades over time or due to external events. In this scenario we assume that this calibration/misalignment fault leads to flawed mazes as well as flawed robot arms when being worked by the assembly. However, the alarm is only triggered during the assembly of mazes, since only they need pins being pushed in place. The general information given above (faulty station flaw products, machining cutters might break, assembly station calibration might be degraded) is captured in a model of the factory plant. Our models focus on the machining and assembly stations with their potential faults, leaving out storage and transport systems, which we assume to have deterministic behavior. Together with the automatically synthesized schedule and the observations the model provides the knowledge from which the AI controller can derive conclusions. It computes the following information:

- Robot0 and Maze0 are predicted to fail (according to the available knowledge), i.e. their probability of success is 0.

- Maze1 has a probability of .83 to succeed.

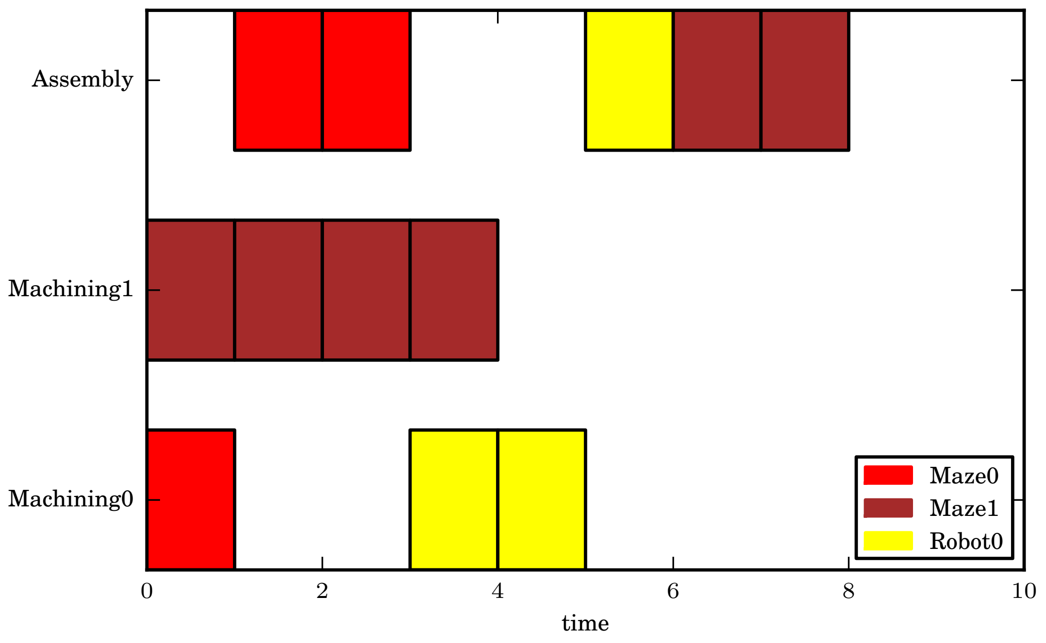


Figure 2.5.: Example schedule of three products being manufactured on two machining and one assembly station.

- The *most probable behavior* of the plant shows the station Machining0 becoming faulty within the first three time steps.

This information leads the AI controller to the following conclusions and according actions:

1. There's not much it can do about Maze0 and Maze1. From the expected finishing times in the schedule the controller can conclude that Maze0 is already broken. It has to ignore it in this and future decisions. Maze1 has a good chance to be produced successfully, it doesn't warrant any changes, let alone stopping the entire plant. **Decision:** Proceed with normal operation.

2. Robot0 is at risk of total loss. However, the controller knows from the schedule it's not yet finished, so it might be worth to try and ask for re-scheduling Robot0, avoiding the machining station assumed to be faulty. **Decision:** Ask scheduler for a changed schedule that avoids the faulty machining station for Robot0.

The AI controller combines the individual decisions, which in this case means that it first asks the scheduler for the re-scheduling and then proceeds with normal operation.

In general two pieces of information are computed, success probabilities and the most probable behavior. Success probabilities with regard to goals (finished products) can be compared against pre-defined thresholds to aid decision making. The most probable behavior, which allows to identify faulty components, can guide specific actions such as re-scheduling.

The example illustrates how the plan assessment problem of computing goal-related success probabilities and diagnoses in form of most-probably faulty components arises, and how solving it aids in automated decision making. Of special interest is the decision to actively gather more information (for example as described in [106]). It requires a trade-off typical in AI, namely between exploration and exploitation. If, for example, many success probabilities are around .5 this indicates that available sensor data only allows decisions close to chance level. In that case, more information is needed for good decisions.

2.2. Assuring Plan Success in Household Robots

Research within CoTeSys in autonomous systems develops robotic household assistants that support humans in their everyday household activities, especially in the kitchen. A demonstrator for household scenarios is being developed, called the assistive kitchen [14]. In the following, we look at an example from this domain involving plan assessment as a means to support a household robot's decisions. We think that plan assessment is more useful for domains such as manufacturing, where systems only have to face uncertainty in their own behavior and act within rather deterministic environments (for example shop floors). However, this didactic example shows that plan assessment is not confined to these domains.

Household chores involve many tasks that benefit from a constant monitoring of how events could affect the task's goals. For example, when setting the table, a malfunctioning arm might lead to a plate being dropped. This might influence subsequent items that are planned to be handled with that same arm. Our example illustrates such a scenario.

The goal in a table setting task is to place a number of items, namely plates, cutlery, cups etc. on the table (see figure 2.6). Each item should be placed in a distinct area on the table, for example a plate in front of a seat. This overall goal can be decomposed in separate goals for each item: that it must be placed in its distinct area.

In this example, the table is equipped with RFID (radio frequency identification) antennae, that allows it to detect RFID tagged items within distinct areas. The household robot is a mobile robot with two arms, each having a gripper as end effector. As part of the CoTeSys demonstration scenario "cognitive household", two such robots are currently in use, called TUM-James and TUM-Rosie [16] (figure 2.6 shows photographs of them).

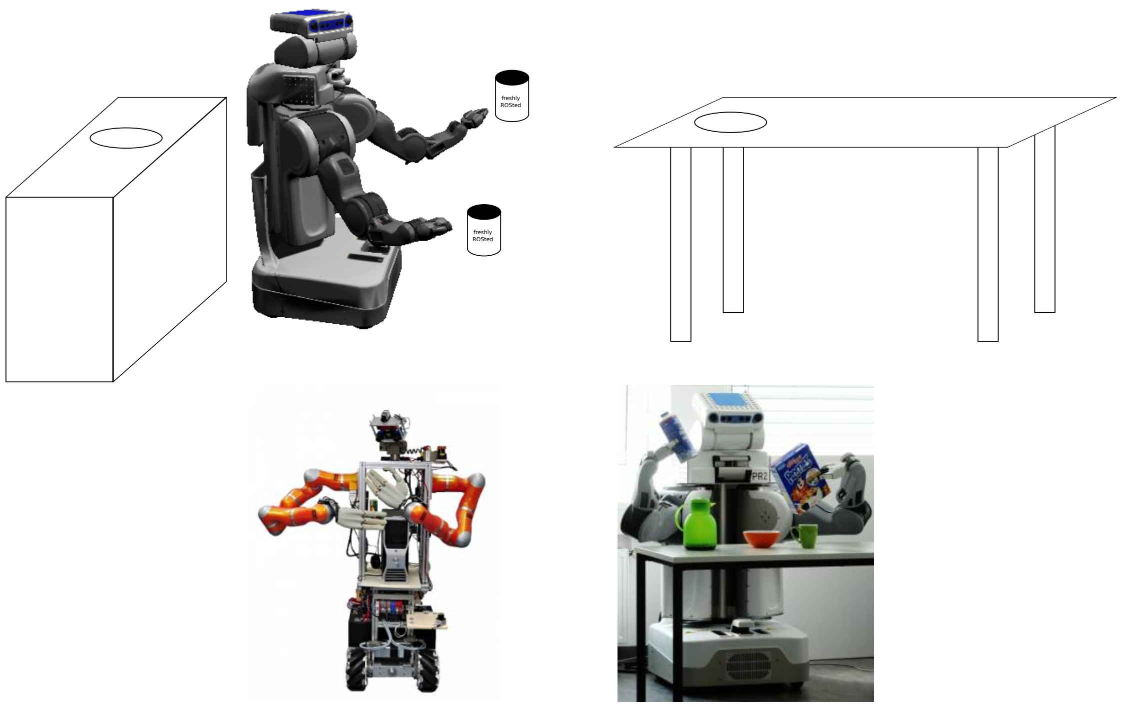


Figure 2.6.: Above: A table setting scenario. A mobile robot equipped with two arms and grippers has the task to set a table with two plates and two cups. Below: The two robots TUM-Rosie and TUM-James. Photographs © Intelligent Autonomous Systems group, Prof. Michael Beetz.

TUM-James is a personal robot 2 (PR2) built by Willow Garage[2]. It has two arms with seven degrees of freedom each, four in the arm, three in the wrist and one in the gripper. It moves via an omni-directional base. TUM-Rosie is custom built from own and commercial parts. It has two arms with seven degrees of freedom from KUKA[3][4], each with a four fingered DLR-HIT hand[5]. It moves using an omni-directional base from KUKA.

The robot's advanced vision perception capabilities allow it to dynamically plan its actions within the kitchen environment. However, occasionally recognition might fail, for example the position of the plate might not be perceived completely accurate. In our example we thus assume that grasping an item might fail with a certain low probability. We further assume that one of the arms has been observed to behave less reliable lately, for example because the pressure sensor in the fingertips gives wrong results unusually often. To work around this, the engineers have updated the robot's internal model with a higher failure probability for that arm. We created a simplified PHCA model, wherein each arm has a certain probability to transition to a failure mode and stay in that mode.

The scenario is now the following: The robot has the task to fetch and place four items on the table, two plates and two cups. Accordingly, it has to achieve four goals, which we denote simply as Plate0, Plate1, Cup0, Cup1. Initially, it stands by the place the items are stored at, for example a cupboard. Then, it starts executing plan $\mathcal{P}$, a sequence of the following steps:

> 1. pick up Plate0, 2. move to table, 3. place Plate0 on table and scan for placed item, 4. move back, 5. pickup Cup0 and Cup1, 6. move to table, 7. place Cup0 on table and scan, 8. place Cup1 on table and scan, 9. move back, 10. pickup Plate1, 11. move to table, 12. place Plate1 on table and scan for placed item.

Figure 2.7 illustrates the scenario for the first four time points. At $t = 3$ we expect to detect the RFID tag from Plate0, but no signal is received. What is most likely to have happened and what should be done? Using plan assessment, the AI controller computes the following information:

- The most probable trajectory explains the signal with a problem with the less reliable arm.

[2]`http://www.willowgarage.com/pages/pr2/overview` (12.2011)

[3]`http://www.kuka-robotics.com/germany/de/` (12.2011)

[4]KUKA LWR-4 arm, which resulted from a cooperation between KUKA and DLR (Deutsches Zentrum Luft- und Raumfahrt) `http://www.robotic.dlr.de/fileadmin/robotic/haddadin/KUKA-DLR_LWR_ISR_2010_v6.pdf` (12.2011)

[5]`http://www.dlr.de/rm/en/desktopdefault.aspx/tabid-3802/6102_read-8918/` (12.2011)

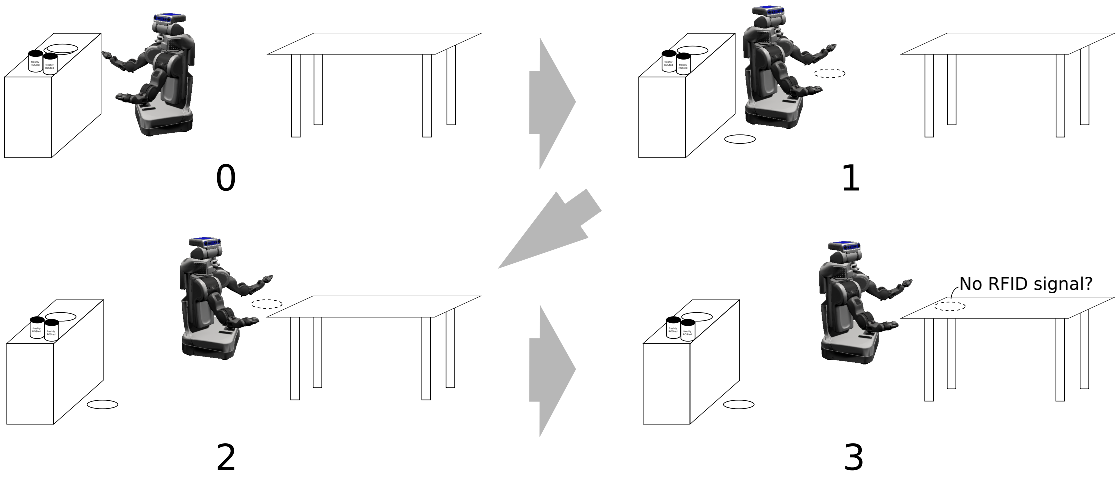


Figure 2.7.: Illustration of the execution of the first 4 plan steps. Due to a malfunctioning arm, the plate slips and falls to the ground. This is unnoticed at first. Only at time $t = 3$, after the robot placed the plate it believes to hold onto the table, but the table doesn't detect the plate's RFID tag, the problem becomes apparent.

- Plate0 has success probability 0. Intuitively this is clear, since the missing RFID signal clearly indicates that Plate0 was not placed right.
- Plate1 also has success probability 0. This makes sense since in our example, which follows the practice in the CoTeSys laboratories, the robot needs both arms to transport the plate. This is reflected in the model, and thus the AI controller can infer that this plate cannot be transported anymore.
- Cup0 has success probability .91. This cup happens to be planned for transport with the good arm. There's only a small chance that this arm started malfunctioning and thus caused the missing RFID signal.
- Cup1 has success probability .09. It is planned for transport at the same time as Cup0 with the other arm, which has been identified as most probable cause for the contingency.

Given, for example, a success threshold of .85 the AI controller could now conclude that Plate0 is lost, but Cup1 may still be placed on the table by re-planning to transport it with the good arm. The robot cannot transport Plate1 anymore, but could signal the problem to a human in the household, asking to place the plate instead. Further reactions could involve further investigation of the most probable cause using, for example, a visual inspection of the arm or an internal diagnostic check.

One might ask at this point why one would only use RFID scanners as sensors when the robots are equipped with many more sensors and powerful recognition capabilities. Despite the fact that this scenario is strongly simplified, it is reasonable to assume that computationally intensive perception capabilities are only used when needed, as illustrated above, and that during normal operation the robot relies on simpler techniques such as RFID.

We made a number of further simplifications in this example. We created a simple PHCA model of the table with its scan capability, the robot with its two arms and each item with one state for being ok and one for a missed goal. Specifically, we left out the actual robot movement or pick-and-place, the distinct areas on the table (only modeling wether placing was successful) and all locations that might be involved apart from the table. Considering faults, we modeled failed grasping as a failure mode of the arm that leads to lost items. A consequence is that Plate1 receives probability 0 in the above scenario. A more realistic PHCA model might reflect that a problem with the arm won't always lead to lost items. Our model as well as the plan and the observations for the above scenario are given in the appendix (see B.2).

As is the case for our main example, plan assessment has to rely on existing AI components. These are being developed within CoTeSys in the form of the cognitive robot abstract machine (CRAM) architecture [17]. It takes care of planning, plan modifications and many complex perception and action tasks such as recognizing objects or planning paths.

3. Background: Modeling and Model-Based Reasoning

This chapter provides the technical background for this work. In particular, it introduces background knowledge for two ingredients for our solution approaches to plan assessment. The first ingredient is generic problem solving with constraint optimization and probabilistic inference, the second modeling of technical systems using high-level description languages.

The chapter first discusses how problems in technical domains can be formalized from the viewpoint of the very general constraint optimization. Then, it introduces formal prerequisites, problem statements and general solution approaches for constraint optimization and constraint satisfaction. Of special interest are methods that enumerate more than just the best solution, i.e. the second best, third best, and so on. Next, probabilistic modeling and inference used in this work is formally introduced. Then, the chapter introduces the probabilistic hierarchical constraint automata (PHCA) that we use in this work to model the complex and uncertain behavior of systems. Finally, it explains Bayesian logic networks, which we have chosen as translation target for PHCAs to leverage all sorts of probabilistic reasoning methods for plan assessment.

3.1. Formal Models as Collections of Variables, Domains and Partial Objective Functions

To formalize the plan assessment problem we can use, as a basis, an elegant description as a search for feasible system behaviors. As a starting point, the set of *possible* behaviors must be defined. In general, this set is called the *problem space*, and the behaviors we are interested in are solutions. A very simple problem space for the example in section 2.1.3 could be a set of tuples of the following form, given that we are only interested in the machining station fault and one maze:

$$\begin{aligned}\{\quad & (\mathsf{machingBroken}, \mathsf{alarm}, \mathsf{mazeOk}),\\ & (\mathsf{machingBroken}, \mathsf{alarm}, \mathsf{mazeFlawed}),\\ & (\mathsf{machingBroken}, \mathsf{noAlarm}, \mathsf{mazeOk}),\\ & (\mathsf{machingBroken}, \mathsf{noAlarm}, \mathsf{mazeFlawed}),\\ & (\mathsf{machingOk}, \mathsf{alarm}, \mathsf{mazeOk}),\\ & (\mathsf{machingOk}, \mathsf{alarm}, \mathsf{mazeFlawed}),\\ & (\mathsf{machingOk}, \mathsf{noAlarm}, \mathsf{mazeOk}),\\ & (\mathsf{machingOk}, \mathsf{noAlarm}, \mathsf{mazeFlawed}) \quad \}\end{aligned}$$

The task is then to identify the sub-set of this space whose elements can be considered feasible behaviors. Typically, one also is interested in sorting this set according to some measure of preference. In our case, the measure is probability. For the diagnosis part of plan assessment, we are interested in the most probable behavior. For the probabilistic part, we will see that ideally we consider all feasible behaviors. Realistically, we will have to restrict ourselves to a smaller set of solutions, for example the k most probable.

So in general, we have the description of the problem space and some objective as input to some algorithm. The objective allows to discriminate solutions from non-solutions or to choose preferred solutions over other solutions. The problem space description is what we denote as *model*. From it, the algorithm instantiates (often implicitly) the problem space. The output of this algorithm is the set of (preferred) solutions.

The algorithms we speak of here are known as *search* algorithms, because they implement clever strategies to search for the problem space elements that we want, the solutions. A thorough text book treatment of searching in AI can be found in [140, chapter 4, p. 94–136].

The problem space is constructed by mapping what we know about the problem onto three basic elements: variables, sets of possible values associated with these variables and partial objective functions. Typically, the space is defined as the cartesian product of the mentioned value sets, which are called *domains*. A domain can be seen as representing knowledge about possible choices along a particular dimension. For example, the knowledge that we may choose among whether the maze product is flawed or not (when searching for a solution) is represented as one dimension and therefore as one domain: $\{\mathsf{mazeOk}, \mathsf{mazeFlawed}\}$.

The actual possibility of choice is formally represented as *variables* attached to the domain. To represent possible solutions during search, variables are assigned values from their associated domain. These variables are not like variables known from popular programming languages such as Java or C++. They are more like variables in an equation system that we want to solve.

Given the problem space, how do we recognize some of its elements as (preferred) solutions? We need an objective that tells us what is allowed (solution), what is preferred and what is forbidden. Preference can be expressed in many ways like lower cost, higher value, higher probability and so on. As mentioned, our preference measure is probability. Formally, the objective is represented through local, partial objective functions.

In the technical systems domain, where behaviors are described or programmed using finite state machines, the problem space is typically discrete: One can only chose from countable many values to be assigned to variables, in our case only from a finite set of values. Preference is then typically expressed locally, for example as probabilities for certain transitions between automata states. These locally expressed preferences are most naturally represented with local, partial objective functions, where each contributes to the overall objective (probabilities of system behaviors in our case).

The description of the problem space in this basic form needn't be created manually. In fact, an important scheme in this work is that this representation is generated automatically from a higher formal language that allows easy modeling with expressive constructs such as hierarchical state machines.

To sum up: To solve a problem, we need a problem space. This space is instantiated during search for solutions from a simple description: a collection of variables, their associated domains and a set of objective functions. Variables and their domains represent what is generally thinkable or possible, while the partial objective functions reduce this further to what is *allowed* or *preferred.* We call this description, or any other description that we can convert into such a basic description, a *model.* The model describes, for example, the behavior of a technical system. We assume the model to be manually designed in an expressive formal language that we need to translate into our basic description. Finally, note that a model is usually suited for solving a number of different problems, it's not the problem statement itself.

3.2. Mathematical Notation in this Work

Before we continue, we establish some mathematical notation that will be used throughout this work. Variables are represented with indexed or subscripted capital letters: $X_1, ..., X_n$

or $O_1, ..., O_l$ or X_{Status}. Sometimes we directly use short words in mathematical sans serif font to denote variables, e.g. Status. Sets of variables and sets in general will be represented with capital letters, e.g. X or O, or short words in mathematical italic font, e.g. Cmd. We also use the notation $\{G_i\}$ to denote a set of indexed items (usually variables) $\{G_1, \ldots, G_i, \ldots, G_n\}$, to avoid cluttering formulas with extra symbols.

Variable values or values in general will be denoted with lower case indexed letters such as $x_1, ..., x_n$ or $o_1, ..., o_l$, or with words in mathematical sans serif font starting with lower case, e.g. marked. Vectors will be represented with bold print, e.g. vectors of variables $\mathbf{X} = (X_1, ..., X_m)$, $\mathbf{O} = (O_1, ..., O_l)$ or vectors of values $\mathbf{x} = (x_1, ..., x_n)$, $\mathbf{o} = (o_1, ..., o_l)$. For a vector $\mathbf{x}$, $\mathbf{x}(i)$ denotes its i-th element. Sometimes we use special index sets, whose vector notation will be $\mathbf{O}^{i:i+n} = (O_i, O_{i+1}, ..., O_{i+n})$ or $\mathbf{x}^I = (x_i)_{i \in I}$. The former denotes an index running from i to $i+n$, the latter an arbitrarily indexed vector according to set I. These can also be vectors of vectors, such as $\mathbf{o}^{0:t} = (\mathbf{o}_0, ..., \mathbf{o}_t)$.

In general, we use round brackets for tuples and sequences, e.g. (X, D, C). Two exceptions apply, where we use angular brackets: 1) If we want to set apart an important mathematical structure, e.g. for a deterministic finite state machine $\langle \Sigma, S, s_0, \delta, F \rangle$. 2) If we want to enhance readability. In a few places readability is greatly enhanced by using angled brackets for sequences instead of round: $\langle x_i \rangle_{i \in I}$, or shorter $\langle x_i \rangle$.

The notation for variable assignments is usually $X_i = x_i$, with X_i being the variable and x_i the assigned value. A more formal treatment of assignments follows in section 3.3.1. In higher-order logical formulas, we use camel case words in mathematical italic font to denote predicates, starting with lowercase letters, e.g. $\mathit{locMarked}(t, l) \Rightarrow \mathit{behaviorIsConsistent}(t, l)$.

3.3. Constraint Satisfaction and Optimization

Many real-world problems, such as scheduling the observation slots on a scientific satellite [18], choosing the right land patches to buy to preserve a species [150], diagnosing integrated circuits [60] or assigning frequencies to radio towers [35], are combinatorial in nature. That is, a countable or finite number of elements suggest themselves as basis for formalization, such as discrete time units, available land patches, circuit elements (adders, multipliers, etc.) or bands of frequencies. Another example is the plan assessment problem posed for technical systems such as manufacturing plants. The behaviors of technical systems are, to a large extend, described using discrete transitions between discrete states.

These problems can be captured in the general framework of constraint optimization. Section 3.1 already described informally how problems can be captured with a set of variables, their domains and a set of partial, local objective functions. Constraint optimization formalizes this and provides general algorithms to search for solutions. It uses such descriptions to describe the problem space and then, in case of an optimization problem, tries to find the optimal assignment to the variables, representing the solution to the problem. The problem of finding a merely satisfactory solution (without differentiating according to some preference) is called constraint satisfaction, and is a special case of constraint optimization.

In this section we introduce basic elements of constraint satisfaction and optimization needed for solving the plan assessment problem. A more thorough introductory treatment of constraint satisfaction and optimization (among others) can be found in the very good text books [51] and [66], and in [148]. In particular, constraint optimization is treated in [51] in chapter 13 and in [66] in chapter 9. Many of the concepts defined in this section are similar to those used in these references.

3.3.1. Formal Definitions and Problems

We now give the formal definitions that are necessary for formalizing and solving problems using constraint optimization. We start with a definition of the basic elements, e.g. variables, domains, assignments, etc., and then define the formal problem statements that are relevant for this work.

Definition 1. Variables, domains, assignments, valuation structures, constraint functions, consistent/inconsistent assignments, constraint net.

1. $X = \{X_1, \ldots, X_n\}$ is a set of n *variables*.
2. $D = \{D_1, \ldots, D_n\}$ is a collection of finite sets such that D_i is the *domain* of X_i, i.e. the set of values that can be assigned to X_i. We may also denote the domain of X_i by D_{X_i}. The size of the domains are written as $|D_i| = d_i$. We write D_Y to denote the Cartesian product $\prod_{X_i \in Y} D_i$ for some subset of the variables $Y \subseteq X$. Accordingly, D_X denotes the Cartesian product $\prod_{X_i \in X} D_i$ of all domains.
3. An *assignment* to a (arbitrarily indexed) set $(X_j)_{j \in J} \subseteq X$ of variables is a set of tuples $\{(X_{j_1}, x_1), ..., (X_{j_l}, x_l)\}$, where $x_1 \in D_{j_1}, ..., x_l \in D_{j_l}$. Typically, we use the alternate notation $(X_{j_1}, ..., X_{j_l}) = (x_1, ..., x_l)$ or $\mathbf{X}^J = \mathbf{x}^{1:l}$, which is common in probabilistic reasoning, to refer to an assignment. If indexing is clear from the

context, we further abbreviate to $\mathbf{X} = \mathbf{x}$. We also call the value tuple $(x_1, ..., x_l)$ an assignment, since usually it is clear from the context which variables are being assigned. Sometimes we refer explicitly to unassigned variables with the *empty value* ε: $(X_{j_1}, ..., X_{j_l}) = (\varepsilon, \ldots, \varepsilon)$. The value ε is naturally *never* element of any variable domain.

4. A *full assignment* is an assignment to all variables in X, i.e. $\mathbf{X}^{1:n} = \mathbf{x}^{1:n}$, $\mathbf{x}^{1:n} \in D_X$. Any other assignment is called a *partial assignment.* Any partial assignment can be written like a full assignment by explicitly referring to the unassigned variables with an "assignment" of the empty value ε.

5. $\overline{\varepsilon} = \varepsilon^{1:i}$ is called the *empty assignment* $(1 \leq i \leq n)$. We don't make a difference between empty assignments of different length and call them all "empty assignment".

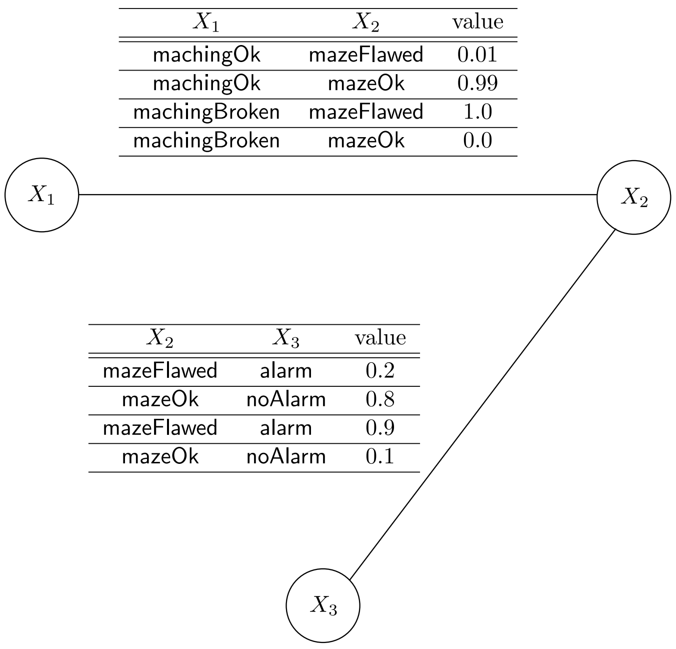
6. A *valuation structure* [149] is a tuple $(E, \leq, \otimes, \bot, \top)$ with a finite set E of preference values, a total order $\leq$ on that set, minimum $\bot \in E$ (the best value) and maximum $\top \in E$ (the worst value). $\otimes : E \times E \to E$ is a commutative, associative and monotonic operation with identity $\bot$ $(\bot \otimes v = v,\ v \in E)$ and absorber $\top$ $(\top \otimes v = \top,\ v \in E)$.

7. $C = \{c_1, \ldots, c_r\}$ is a finite set of r *constraint functions.* A constraint function c maps assignments to preference values $c : D_V \to E$. The set $V \subseteq X$ is called the *scope* of c, $|V|$ the *arity* of c. These constraint functions are also called *soft constraints,* because often they represent soft restrictions on variable assignments, as opposed to hard constraints that either allow or forbid partial assignments. However, note that these functions can be both hard or soft constraints. In this work, we refer to them simply as *constraints.*

8. If an assignment is mapped to $\top$, it is called an *inconsistent assignment.*

9. A *constraint net* is a tuple $\mathcal{R} = (X, D, C)$. The term "net" reflects that $\mathcal{R}$ implicitly defines a graph with variables X as nodes and constraints C as edges. Since constraints may very often be defined over more than two variables, the edges usually are hyper edges (connecting more than two nodes) and the graph accordingly a hyper graph.

Constraint nets are a low-level encoding of models. We chose them as one possible translation target for our higher level descriptions using probabilistic automata, that is PHCA (which will be explained in section 3.6 later in this chapter). This allows us to solve problems such as plan assessment as constraint optimization problem. We define



X_1	X_2	value
machingOk	mazeFlawed	0.01
machingOk	mazeOk	0.99
machingBroken	mazeFlawed	1.0
machingBroken	mazeOk	0.0

X_2	X_3	value
mazeFlawed	alarm	0.2
mazeOk	noAlarm	0.8
mazeFlawed	alarm	0.9
mazeOk	noAlarm	0.1

Figure 3.1.: Example constraint net with three variables, representing a machining station working or being broken, a maze product being well manufactured or flawed, and an alarm being quiet or triggered.

two variants of constraint optimization problems based on constraint nets: probabilistic constraint optimization problems (PCOP or COP) and weighted constraint satisfaction problems (WCSP). Both are specializations of a valued constraint satisfaction problem (VSCP) [149]. Before we do that, however, let us illustrate the definitions with a small example.

Recall the very abstract problem space for our cognitive factory example from the beginning of this chapter. It models a single machining station, one maze product and the alarm. Accordingly, we use three variables X_1, X_2, X_3 for the machining station, the maze and the alarm. Their domains are $D_1 = \{\mathsf{machingBroken}, \mathsf{machingOk}\}$, $D_2 = \{\mathsf{mazeOk}, \mathsf{mazeFlawed}\}$ and $D_3 = \{\mathsf{alarm}, \mathsf{noAlarm}\}$. Strongly abstracted, a broken machining station leads to a flawed maze and a flawed maze in turn triggers an alarm. For this example we formally express these relations as two constraint functions c_1, c_2. They capture these links, but also describe a system that is not perfect: false alarms may happen and sometimes the maze may become flawed on a properly working machining station. The functions map to probability values, that is we require a valuation structure with $E = [0, 1]$ and $\otimes$ being the standard product. Figure 3.1 shows the constraint net for this example, along with the tables defining the two constraints.

Problem 1. Probabilistic constraint optimization (PCOP/COP): A probabilistic constraint optimization problem is a constraint net $\mathcal{R} = (X, D, C)$ with an associated valuation structure $([0, 1], \leq, \cdot, 1, 0)$, where $\cdot$ is the standard product on the real numbers, and the associated task to find the most probable full assignment $\overset{*}{\mathbf{x}}$:

$$\overset{*}{\mathbf{x}} = \arg\max_{\mathbf{x} \in D_X} \prod_{c \in C} c(\mathbf{x})$$

$\overset{*}{\mathbf{x}}$ is the *optimal solution* to $\mathcal{R}$. Any other consistent assignment $\mathbf{x}$, i.e. with $\prod_{c \in C} c(\mathbf{x}) > 0$, is called a *solution*.

Next we define weighted constraint satisfaction problems (WCSP).

Problem 2. Weighted constraint satisfaction (WCSP): A weighted constraint satisfaction problem is a tuple $(\mathcal{R}, \top_{\text{ub}})$, where $\mathcal{R}$ is a constraint net and $\top_{\text{ub}} \in \mathbb{N}$. It has an associated valuation structure $(\{0, \ldots, \top_{\text{ub}}\}, \leq, \oplus, 0, \top_{\text{ub}})$ representing costs, where $\oplus$ is defined in terms of the standard addition $a \oplus b := a + b$ **if** $a + b < \top_{\text{ub}}$ **else** $\top_{\text{ub}}$, and the associated task to find the least cost full assignment $\overset{*}{\mathbf{x}}$:

$$\overset{*}{\mathbf{x}} = \arg\min_{\mathbf{x} \in D_X} \sum_{c \in C} c(\mathbf{x})$$

$\overset{*}{\mathbf{x}}$ is the *optimal solution* to $\mathcal{R}$. Any other consistent assignment $\mathbf{x}$, i.e. with $\sum_{c \in C} c(\mathbf{x}) < \top_{\text{ub}}$, is called a *solution*.

For plan assessment, we are going to need a natural extension to both problems, namely computing the k best solutions instead of just the single optimal one. There are a number of other motivations for computing k best solutions, as is detailed in 3.4. For now, we will focus on constraint reasoning.

Problem 3. k-best probabilistic constraint optimization (k-best PCOP): A k-best PCOP is a tuple $\mathcal{R}(k) = (X, D, C, k)$ where $k \in \mathbb{N}$ and X, D, C as defined for a PCOP. Its associated task is to compute the set of k most probable full assignments $(\overset{*}{\mathbf{x}}_i)_{\leq k}$, indexed such that more probable assignments have lower indices:

$$(\overset{*}{\mathbf{x}}_i)_{\leq k} = \left(\arg\max_{\mathbf{x} \in D_X}[j] \prod_{c \in C} c(\mathbf{x}) \right)_{j \in \{1, \ldots, k\}}$$

$(\overset{*}{\mathbf{x}}_i)_{\leq k}$ is the sequence of the *k-best solutions* to $\mathcal{R}$. Any other consistent assignment $\mathbf{x}$, i.e. with $\prod_{c \in C} c(\mathbf{x}) > 0$, is called a *solution*.

$(\mathbf{x}_i)_{\leq k}$ is, in general, a sequence ordered by preference of an assignment: $(\mathbf{x}_i)_{\leq k} := (\mathbf{x}_i)_{i\in\{1,\dots,k\}}$ with $\bigotimes_{c\in C} c(\mathbf{x}_l) \leq \bigotimes_{c\in C} c(\mathbf{x}_m)$ **iff** $1 \leq l \leq m \leq k$. In case of a PCOP, the ordering is specialized as $\prod_{c\in C} c(\mathbf{x}_l) \geq \prod_{c\in C} c(\mathbf{x}_m)$ **iff** $1 \leq l \leq m \leq k$. The operator $\arg\max[k]$ computes the k-th best solution as defined through the algorithm 3.1.

Algorithm 3.1 Function defining $\arg\max[k]$.

```
1: function ARGMAXK($k$, $D_X$, term to maximize $T$)
2:     $S := D_X$, $r \leftarrow k$-length array
3:     for $i \leftarrow 1, ..., k$ do
4:         $\overset{*}{\mathbf{x}} \leftarrow \arg\max_{\mathbf{x}\in S} T$                ▷ Compute best solution
5:         $r[i] \leftarrow \overset{*}{\mathbf{x}}$
6:         $S \leftarrow S \setminus \{\overset{*}{\mathbf{x}}\}$     ▷ Exclude solution just found from possible assignments
7:     end for
        return $r$
8: end function
```

A k-best PCOP has two special cases, namely computing the single best solution ($k = 1$) and computing all solutions (termed as $k = \infty$). The former, of course, is equivalent to the original PCOP definition. A k-best WCSP can be defined analogous to a k-best PCOP, only replacing $\arg\max$ with $\arg\min$ (with $\arg\min[k]$ analogous to $\arg\max[k]$) and $\prod$ with $\sum$: $(\overset{*}{\mathbf{x}}_i)_{\leq k} = \left(\arg\min_{\mathbf{x}\in D_X}[j] \sum_{c\in C} c(\mathbf{x})\right)_{j\in\{1,\dots,k\}}$. The sequence $(\overset{*}{\mathbf{x}}_i)_{\leq k}$ is then ordered such that $\sum_{c\in C} c(\mathbf{x}_l) \leq \sum_{c\in C} c(\mathbf{x}_m)$ **iff** $1 \leq l \leq m \leq k$. In fact, any (k-best) PCOP can be converted into an equivalent (k-best) WCSP by negating and logarithmizing the probability values in the PCOP constraints. Specifically, one has to apply the function

$$f(x) = \begin{cases} -\alpha \log x & \text{if } x > 0 \\ 0 & \text{else} \end{cases}$$

where α is a large integer to minimize precision loss. In our examples, this loss has been negligible.

3.3.2. Basic Algorithms and State-of-the-Art Techniques

Algorithms that solve the above defined problems are often a composition of two general mechanisms:

1. Classical AI search for solutions
2. Problem reformulation

Searching for solutions is done by repeatedly assigning values to variables, taken from their associated domains, and testing these assignments against the objective. Two variants are possible: local search, which modifies full assignments to generate new assignments, and systematic search, which finds values for each variable in turn until a full assignment is created. It thereby traverses a tree-structure (either implicitly or based on an explicit tree, depending on the algorithm), where nodes correspond to partial assignments and leaves to full assignments, and thus, potential solutions. In this work, we focus on the latter. Search heuristics help with, e.g., clever choices of the next value to assign. One of the most famous heuristic search procedures is the A* algorithm [79, 53]. The core procedure in many constraint solvers is called branch-and-bound. In this work we used implementations of both of these algorithms in two different off-the-shelf constraint optimization tools.

The second mechanism, which is called inference or propagation in constraint reasoning (not to be confused with inference in probabilistic reasoning), is based on turning implicit constraints in a constraint net $\mathcal{R} = (X, D, C)$ into explicit ones and adding them to the set C of the formulation. Implicit constraints are those that are entailed by C, i.e. they are satisfied whenever all constraints in C are satisfied. This process can be repeated until a single constraint over all variables explicitly states solutions. Most often, however, this process is combined with search, where for example the new constraints are exploited by heuristics.

Branch-and-Bound and A* as Basis for Algorithms Used in this Work

In this work, we use the off-the-shelf constraint solvers Toolbar[1] and Toulbar2 [2]. Both accept WCSPs as input. We extended both solvers with implementations of k-best algorithms. For Toolbar we implemented an approach that first generates a heuristic using an algorithm called mini-bucket elimination, then utilizes the heuristic in an A*-search for the k-best solutions. The approach was first introduced in [99], albeit only for the case $k = 1$. A text book description of this approach as well as mini-bucket elimination can be found in [51, chapter 13, pp. 379]. In Toulbar2 we use its branch-and-bound procedure, that we extended to generate k-best solutions. The extensions are detailed in section 5.2.1.

Branch-and-Bound seems to be the preferred search procedure to be combined with inference methods in constraint optimization, yet is not as well known A* when it comes to heuristic search. Therefore, we now describe the branch-and-bound procedure used in tools such as Toulbar2 in some detail, while we restrict our account of A* to a short

[1]`https://mulcyber.toulouse.inra.fr/projects/toolbar/` (03.2011)
[2]`https://mulcyber.toulouse.inra.fr/projects/toulbar2` (03.2011)

recap. The A* algorithm is described in more detail in many text books, for example in [140, chapter 4].

Branch-and-bound methods originated in the field of operations research [112] and were applied for constraint optimization several years ago [111] in the wake of the upcoming soft constraint propagation [42, 47]. The branch-and-bound procedure that is used in tools like Toulbar2 can be seen as sort of a depth-first walk of the search tree with short-cuts given by heuristics. With costs as preference values, it roughly works as follows: Search nodes are expanded in a depth-first manner. At each expansion step, lower bounds for potential solutions in sub-trees of the current node are computed using the mentioned soft constraint propagation (among others) and compared against an upper bound. The upper bound is given externally, and later updated with the value of the best solution found so far. If for a sub-tree the lower bound is larger then the upper bound, all possible solutions in that sub-tree can only be worse than the optimal solution (which is guaranteed to be cheaper than the upper bound), and therefore this sub-tree can be pruned. Among the remaining sub-trees, the next is chosen for traversal. The algorithm backtracks whenever it reaches a full assignment, i.e. a potential solution. In case its cost is lower than the upper bound, the upper bound is updated with this cost. The algorithm stops when all search nodes have been either expanded or pruned.

Algorithm 3.2 shows a basic branch-and-bound algorithm that exploits soft constraint propagation implemented in the heuristic h, i.e. it computes the lower bound for costs of unexplored sub-trees. Cost function g evaluates an assignment against the constraints C in the given problem, giving the cost of the current partial solution $\mathbf{x}'$. The sum $h + g$ yields an optimistic cost estimate for all possible solutions in a particular sub-tree. Optimistic means that the cost will never be overestimated, which implies that no sub-tree that could contain the optimal solution will be pruned. This estimate is compared against the cost of the best solution found so far, $\overset{*}{\mathbf{x}}$. If a new full assignment $\overset{*}{\mathbf{x}}'$ is found, it is stored as new current best solution if it is better than the previous current best solution.

Like branch-and-bound, A* search for combinatorial optimization works by traversing a search tree. However, it remembers all partial assignments created so far and always chooses the most promising (according to an optimistic heuristic) partial assignment for expansion. This can be imagined as gradually expanding a search fringe that stretches towards the optimal solution. It is important to mention that A* is probably more known as an optimization procedure for shortest path problems in graphs. The version of A* for combinatorial optimization is a specialization: There is no explicit set of goal nodes as for the graph-based version, the algorithm stops as soon as the first full assignment is generated. This corresponds to reaching a leaf node in the search tree. The property of

Algorithm 3.2 Basic branch-and-bound algorithm to solve weighted constraint satisfaction problems (WCSP).

```
1: function BRANCHANDBOUND$((\{X_1,\ldots,X_n\},\{D_1,\ldots,D_n\},C)$, $\top_{\text{ub}}$, heuristic $h$, $g$)
       return BNBRECURSE$((\{X_1,\ldots,X_n\},\{D_1,\ldots,D_n\},C)$, 0, $\overline{\varepsilon}$, $\overline{\varepsilon}$, $h$, $g$)
2: end function
3: function BNBRECURSE$((X,D,C)$, $i$, $\mathbf{x}$, $\overset{*}{\mathbf{x}}$, $h$, $g$)
4:     if $i > |X|$ then
5:         return $\mathbf{x}$
6:     end if
7:     for $v \in D_i$ do
8:         $\mathbf{x}' \leftarrow \mathsf{assign}(\mathbf{x}, X_i, v)$
9:         if $h(\mathbf{x}') + g(\mathbf{x}') \leq g(\overset{*}{\mathbf{x}})$ then
10:            $\overset{*}{\mathbf{x}}{}' \leftarrow$BNBRECURSE$((X,D,C), i+1, \mathbf{x}', \overset{*}{\mathbf{x}}, h)$
11:            if $g(\overset{*}{\mathbf{x}}{}') < g(\overset{*}{\mathbf{x}})$ then
12:                $\overset{*}{\mathbf{x}} \leftarrow \overset{*}{\mathbf{x}}{}'$
13:            end if
14:        end if
15:    end for
16:    return $\overset{*}{\mathbf{x}}$
17: end function
```

admissibility of the heuristic (i.e. that it is optimistic) guarantees that this assignment will be the optimal one [53].

Techniques used in State-of-the-Art Constraint Optimizers

Toolbar and Toulbar2 use extended versions of the basic algorithm 3.2 that implement many additional techniques and heuristics. We already mentioned soft constraint propagation [42, 47]. A more basic, widely used set of extensions are heuristics for the choice of variable and value orderings [48]. Another are preprocessing steps, in particular tree-decomposition [24]. This technique tries to discover a tree-structure in the given problem and make it available for the algorithm to exploit. Solving a problem while exploiting its underlying tree-structure reduces the complexity from an exponential dependency on the number of all variables to an exponential dependency only on the number of variables in the largest clique of the constraint net graph. It is important because it can be used to exploit the typically well structured design of technical systems, for example in integrated circuits [60]. One way is to first generate a tree-decomposition [24], then apply a tree-structure algorithm such as cluster tree elimination [100]. Another is to exploit tree-decomposition during branch-and-bound using *structural valued goods*[153].

3.4. K-Best Solution Generation

For many optimization problems it makes sense to consider, besides the best solution, the 2nd, 3rd, 4th, ... k-th best solution. For example, in GPS navigation, the shortest route may include a highway that the user wants to avoid because she knows (unlike her navigation device) about a recent construction site. It might help in this case to look for an alternative route among the 2nd, 3rd, ... best solution. Another example is medical diagnosis, where a diagnosis might not be the most probable, however it might identify a very serious illness, which is best being further investigated despite being unlikely. Finally, in plan assessment, the k most probable trajectories are used to approximate the distribution over all trajectories. This works well if the distribution is peaked. We elaborate this point further in Section 5.2.5.

In general, we can see two reasons to perform k-best optimization:

1. The real optimum is not the one computed, but is among the k best solutions. Reasons could be that the underlying model is wrong, but still approximately correct, that not all relevant aspects of the domain were formalized, e.g., because it was too expensive, or that the environment changed, causing the model to become a coarser approximation. Enumerating the k best solutions gives human experts the opportunity to easily choose the real optimum.

2. The k best solutions can be combined to generate new information, as in plan assessment.

We will see in chapter 5 that plan assessment can be conveniently developed as an extension of model-based diagnosis that generates a set of k most probable system behaviors. This set then allows to approximate a goal success probability $Pr(\mathcal{G}_i \,|\, \mathbf{o}^{0:t})$ by summing over goal-achieving trajectories among the k most probable. In that chapter we also introduce the two k-best algorithms that we use to compute most probable trajectories in decreasing order. They are implemented in Toolbar and Toulbar2, respectively.

First versions of k-best algorithms for combinatorial optimization were developed in the 1970ies in the area of Operations Research [113, 131]. An algorithm for a k-best version of most probable explanation was presented in [151]. Besides this, k-best approaches seem rare in the probabilistic reasoning community. An extensive body of research exists for k-best graph algorithms, e.g. k-shortest path. [29] is an early comparative study of such algorithms. A more recent work that solves a problem approximately by enumerating

k most probable solutions is [45]. It addresses the problem of visually identifying and tracking a number of closely spaced objects. A general solution is to track multiple hypotheses of the objects' whereabouts, and [45] describes an approach that enumerates the k most probable hypotheses.

3.5. Probabilistic Inference

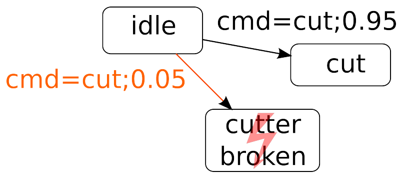


Figure 3.2.: Example for probabilistic choice between transitioning (upon receiving command `cut`) to the nominal location `cut` or to the failure location `cutter broken`.

In this section we introduce basic elements necessary to understand probabilistic reasoning. The key mathematical structure is the widely used Bayesian network [135], or Bayesian net for short. This structure is analogous to the constraint nets defined earlier in section 3.3.1. In fact, constraint nets generalize Bayesian nets. We exploit this fact and use the definitions from that section as basis for the formalities in this section.

Probabilistic reasoning addresses problems that arise out of the need to make decisions based on uncertain knowledge. In our case, this is uncertain knowledge about the internal states of technical systems. For example, we might see an alarming sensor signal in our factory, but cannot conclude directly its internal state: did the cutter of the machining station break and lead to that signal, via a damaged product? Or is maybe the assembly station at fault? The sometimes uncertain interactions that allow us, in the end, to derive a conclusion are defined in a probabilistic model, e.g. a Bayesian net.

The mathematical foundations are laid out in probability theory, which captures uncertain knowledge with the basic elements of random variables and distributions over the values these variables may take. Specifically, a set of random variables X and their domains D span a problem space, which captures possible solutions of a given problem. X and D correspond directly to the sets X and D defined in section 3.3.1. D is again restricted to finite sets. Distributions can be seen as special partial objective functions, mapping variable assignments to probability values $[0, 1]$. A distribution encodes uncertain knowledge about a part of the problem domain. For example, we may know the probability with which a machining station, upon receiving a command, transitions to a fault location instead of the nominal location, see figure 3.2. This "choice" is encoded with a conditional probability function $Pr(l_i \mid l_j)$, which gives the probability that the transition to location l_i happens, given that we are in location l_j. In the example, the probability that the station transitions from location idle to location cutter broken is encoded as $P(\textsf{cutter broken} \mid \textsf{idle}) = 0.05$.

Conditional probability functions specialize partial objective functions not only with respect to the codomain $[0, 1]$. They also add the semantics of uncertain knowledge about a specific variable, called child variable, being influenced because we know for sure the values of a set of parent variables. That is, a specific distribution over values of the child is valid *under the condition* that we know a particular assignment for the parents. In the probabilistic transition example in figure 3.2 we see the distribution $Pr(\cdot \mid \textsf{idle}) = \{0.05, 0.95\}$. For a different parent location this distribution would change, in particular for any location that is not idle, the probability of transitioning to, say, broken cutter would be 0. In Bayesian nets, this semantic of conditioning is made explicit in a directed acyclic graph with the random variables as nodes.

3.5.1. Formal Definitions and Problems

In addition to definition 1 we define the following mathematical structures specifically for probabilistic reasoning:

Definition 2. Conditional probability functions, evidence variables, consistency among assignments, Bayesian net. Let $X = \{X_1, \ldots, X_n\}$ be the set of random variables and $D = \{D_1, \ldots, D_n\}$ their associated domains.

1. $P = \{Pr_1, \ldots, Pr_n\}$ is a finite set of n *conditional probability functions*. A conditional probability function Pr maps assignments to probability values $Pr : D_V \to [0, 1]$. The set $V \subseteq X$ is called the *scope* of Pr. The partition $\{V_C, V_P\}$ of V with $V_C = \{X_i\}$ and $V_P = V \setminus V_C$ determines the conditioning relationship among the scope variables. A single variable X_i, the child, is conditioned through the knowledge about the remaining variables, the parents. This partition is denoted as $Pr(X_i \mid X_{j_1}, \ldots, X_{j_m})$, with $\{X_{j_1}, \ldots, X_{j_m}\} = V_P$. V_P is also denoted as *parents*(X_i). Note that $n = |X|$, i.e. there is exactly one function for each variable.

2. $X_E \subseteq X$ is a set of designated *evidence* variables. This means the values of these variables are (at least partially) expected to be known. An assignment to variables in X_E (either partial or to all of them) is denoted as evidence $\mathbf{e}$.

3. A full assignment $\mathbf{x}$ is said to be consistent with a partial assignment $\mathbf{y}^{(i,\dots,j)}$ $(i < j \leq n)$, iff the full assignment has the same values as $\mathbf{y}^{(i,\dots,j)}$ in the according positions, i.e. $\mathbf{x}^{(i,\dots,j)} = \mathbf{y}^{(i,\dots,j)}$. This is written as $\mathbf{x} \vdash \mathbf{y}^{(i,\dots,j)}$. In particular, we write $\mathbf{x} \vdash \mathbf{e}$ to indicate that $\mathbf{x}$ is consistent with the evidence.

4. The concatenation of two partial assignments $\mathbf{y}$ and $\mathbf{z}$, meaning their co-occurrence, is written as $(\mathbf{y}, \mathbf{z})$. We leave out the parentheses if the meaning is clear without them, e.g. in $Pr(\mathbf{y}, \mathbf{z})$.

5. A *Bayesian net* is a tuple $\mathcal{R}_B = (X, D, G, P)$, where G is a directed, acyclic graph with X as its nodes and a directed edge for each parent→child relationship. Thereby it makes the conditioning relations among the variables explicit. Each net $\mathcal{R}_B$ defines a full joint distribution over all assignments X, given as $Pr_B(X_1, \dots, X_n) = \prod_{i=1..n} Pr(X_i \mid \mathsf{parents}(X_i))$.

$\mathcal{R}_\mathcal{B}$ corresponds to a special constraint net with valuation structure $([0,1], \leq, \cdot, 1, 0)$ and with the added information of the conditioning relations given in graph G. G can be seen as sort of an explicit specialization of the implicitly defined (hyper) graph, where an undirected (hyper) edge connecting parents with a child is specialized to a set of directed edges for each parent→child relationship. Another specialization is the explicitly defined set of evidence variables, which represent certain knowledge about the problem (as opposed to uncertain knowledge represented in the other variables). For more information on the relations between constraint and probabilistic reasoning we refer to [148] and [100].

Like constraint nets, Bayesian nets are used as low-level encodings of models of technical systems, with possible observations and commands being mapped to evidence variables. In this work we use them, like constraint nets, as translation target for our high level descriptions with PHCA. This allows to solve plan assessment by solving probabilistic reasoning problems that can be defined over Bayesian nets. Next, we will define three well known general probabilistic reasoning problems over Bayesian nets.

Problem 4. Most probable a posteriori hypothesis (MAP): Let $Y, Z \subseteq X$ with $Y \cap Z = \emptyset$, $Y \cup Z = X$ and $X_E \subseteq Z$. Then, the most probable a posteriori hypothesis problem is a Bayesian net $\mathcal{R}_B = (X, D, G, P)$ with the associated task of finding the most probable partial assignment $\overset{*}{\mathbf{y}}$ to Y, given evidence $\mathbf{e}$:

$$\overset{*}{\mathbf{y}} = \arg\max_{\mathbf{y} \in D_Y} \sum_{\mathbf{z} \in D_Z, (\mathbf{y},\mathbf{z}) \vdash \mathbf{e}} Pr_B(\mathbf{y}, \mathbf{z})$$

Problem 5. Most probable explanation (MPE): The most probable explanation problem [135, Chapter 5, p. 250] is a Bayesian net $\mathcal{R}_B = (X, D, G, P)$ with the associated task of finding the most probable full assignment $\overset{*}{\mathbf{x}}$, i.e. with highest probability value, given evidence $\mathbf{e}$:

$$\overset{*}{\mathbf{x}} = \underset{\mathbf{x} \in D_X, \mathbf{x} \vdash \mathbf{e}}{\arg\max}\ Pr_B(\mathbf{x})$$

Our definition of the MAP problem follows [90]. The MAP problem is relevant for plan assessment in that the most probable a posteriori hypothesis, for our models, encodes the most probable system behavior (in the relevant variables Y) and thus corresponds to the most probable diagnosis. The MPE problem is a specialization of MAP: in case all evidence variables are known and no irrelevant variables exist (i.e. $X_E = Z$), MAP becomes equal to MPE, because there is then only one partial assignment to Z consistent with the evidence (the evidence itself), which means the sum effectively vanishes.

The MPE problem is connected to the COP (PCOP) problem of finding most probable full assignments. Given a random variable X_i, let $I_{parents(X_i)}$ be the index set that appropriately picks values from full assignment $\mathbf{x}$ to be assigned to the parents of X_i. Then with

$$\underset{\mathbf{x} \in D_X, \mathbf{x} \vdash \mathbf{e}}{\arg\max}\ Pr_B(\mathbf{x}) = \underset{\mathbf{x} \in D_X, \mathbf{x} \vdash \mathbf{e}}{\arg\max} \prod_{i=1..n} Pr(X_i = \mathbf{x}(i) \mid parents(X_i) = \mathbf{x}^{I_{parents(X_i)}})$$

we have generally the same problem formulation, only missing two further adaptations: Each Pr is represented as a constraint function in C, and evidence must be encoded as additional constraints mapping inconsistent variable values for X_E to 0. Then, solving the resulting COP gives the most probable explanation.

Problem 6. Marginals: The marginal computation problem [135, Chapter 4.5, p. 223] is a Bayesian net $\mathcal{R}_B = (X, D, G, P)$ with the associated task of computing, for a given query $X_i = x$ and evidence $\mathbf{e}$, the marginal probability $Pr(X_i = x \mid \mathbf{e})$:

$$Pr(X_i = x \mid \mathbf{e}) = \sum_{\mathbf{x} \in D_X, \mathbf{x} \vdash \mathbf{e}} Pr_B(X_1 = \mathbf{x}(1), \ldots, X_i = x, X_{i+1} = \mathbf{x}(i+1), \ldots, X_n = \mathbf{x}(n))$$

The marginal problem connects to plan assessment in that computing success probabilities $Pr(\mathcal{G}_i \mid \mathbf{o}^{0:t})$ can be mapped to computing marginal probabilities for assignments to designated goal variables, $G_i^{t_{\text{end}}} = \mathsf{marked}$. These variables, with associated domain $\{\mathsf{marked}, \mathsf{unmarked}\}$, encode a goal as automata locations that must be reached, i.e. marked, at time t_{end}. The superscript ^{t} is used to denote that a variable represents a partial state at some time point t.

The introduced probabilistic reasoning problems are, in full generality, NP-hard problems [138, 41] because they require computing (in the worst case) the full joint probability Pr_B. This corresponds to enumerating all full assignments and is therefore exponential in the number of variables. MAP is generally harder than MPE and the marginal problem [134]. The next section will introduce two classes of efficient algorithms that are particularly interesting for this work.

3.5.2. Basic Algorithms and State-of-the-Art Techniques

Many algorithms for graphical models such as Bayesian nets have been developed. A good text book reference for graphical models, exact and approximate inference methods is [21, chapters 8, 10 and 11]. We focus on two classes of algorithms that are relevant for this thesis: junction tree algorithms and sampling.

Junction tree algorithms are exact inference methods that exploit the global tree-like structure that many problems or models exhibit. This is especially true for models that are derived from engineered technical systems, where well-structured design is a central paragon. That makes these types of algorithms relevant for this work.

Sampling denotes a widely used class of approximative methods. They are attractive because they are anytime algorithms, meaning that they give approximate results anytime when halted before termination. Furthermore, stochastic guarantees on the approximation error can be given.

Junction Tree Algorithms and Arithmetic Circuits

Junction tree algorithms are dynamic programming methods based on exploiting a tree-structure hidden in the Bayesian net graph G. As a first step, this structure, called a join tree, must be discovered and made explicit. This step is often considered an offline compilation step, as probabilistic queries can then be answered for different evidences without the need to recreate the join tree.

This concept of offline/online computation is often used to speed up computations during the time when solutions to problems are actually requested. This time is called online phase. Before that, in the offline phase, computationally intensive preparations are done. This works because often computationally hard problems can be split into a hard step, ideally performed only once, and an easy step that has to be done for each user input. A naive approach would be to simply precompute all solutions and store them such that they can be retrieved quickly. Since this requires a huge amount of space, the approaches are usually more sophisticated, providing intelligent trade-offs between space and time complexity as well as offline and online phase. In case of junction tree algorithms, the online phase (after generating the join tree) still has exponential complexity. The arithmetic circuit approach that we introduce in a moment pushes much more of the effort into the offline phase, rendering the online phase extremely fast.

The join tree consists of connected "super-nodes" that cluster random variables. Clusters are connected only if they share variables. They can be seen as sub-problems of the original problem. The second step consists of a 2-stage process message passing between clusters, first towards the root node (which has to be chosen), then outwards towards the leaves. A message is a function that encodes the effect of the sending cluster on the receiving cluster. For a more detailed description we refer to [89], or to [100] for the description of a more general version of such an algorithm called cluster-tree elimination. Generalized junction tree algorithms as described in [100] can be used to compute MPE and marginals. Which problem they solve depends on choosing different sets of operators over conditional probability functions, one that combines functions (e.g. via product), and one that projects a distribution onto chosen variables (e.g. by summing over random variables for marginalization, or maximizing for MPE). A concrete implementation of a junction tree algorithm for MPE is patented in [88]. The generalized join trees these algorithms use correspond to the tree-decompositions, mentioned in section 3.3.2, which tools like the constraint solver Toulbar2 can exploit.

In this work we focus on a framework that subsumes the junction tree approach [38, 91, 37, 46]. It is based on compiling Bayesian nets into structures called arithmetic circuits, which are efficient representations of the distribution Pr_B as polynomials. The main advantage is that once an arithmetic circuit is computed, things such as success probabilities can be computed extremely fast, for *different observations*. That is, the arithmetic circuit need not be recreated if the observations change. Next, we recap the approach following the explanations and notations given in [46].

For each Bayesian net, we can represent its distribution with a unique polynomial of the form

$$f = \sum_{\mathbf{x} \in D_X} \prod_{x\mathbf{u} : \mathbf{x} \vdash x\mathbf{u}} \lambda_x \theta_{x \mid \mathbf{u}}$$

The sum ranges over all possible full assignments. The product ranges over all partial assignments, denoted $x\mathbf{u}$, that correspond to scopes of conditional probability functions and with which the current full assignment $\mathbf{x}$ is consistent. The single product thus gives the probability value of $\mathbf{x}$.

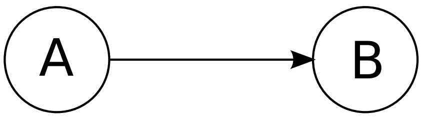


Figure 3.3.: Simple example Bayesian net.

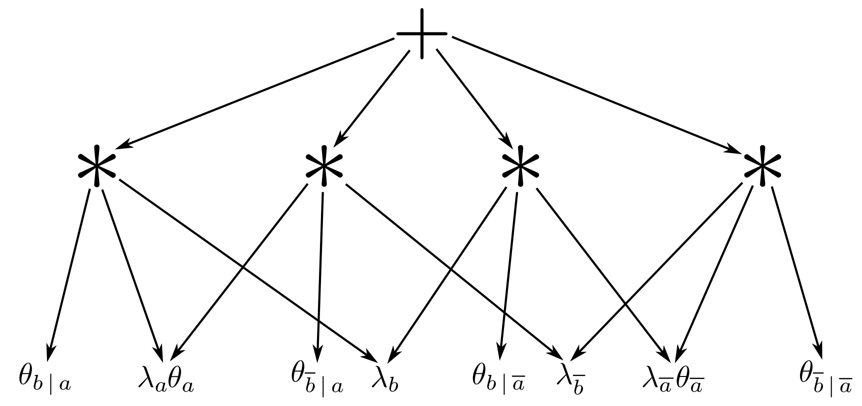

+
*
*
*
*
$\theta_{b\,|\,a}$
$\lambda_a\theta_a$
$\theta_{\overline{b}\,|\,a}$
λ_b
$\theta_{b\,|\,\overline{a}}$
$\lambda_{\overline{b}}$
$\lambda_{\overline{a}}\theta_{\overline{a}}$
$\theta_{\overline{b}\,|\,\overline{a}}$

Figure 3.4.: Arithmetic circuit for the Bayesian net in figure 3.3.

The polynomial consists of two types of variables. The λ_x variables indicate whether some value x is consistent with the evidence or not. If it is not consistent we set $\lambda_x = 0$, which "erases" the complete product, and thus the full assignment being inconsistent with evidence, from the sum. The θ variables represent conditional probabilities: $\theta_{x\,|\,\mathbf{u}} = Pr(X_i = x\,|\,parents(X_i) = \mathbf{u})$ (assuming that $\mathbf{u}$ is appropriately indexed). Note that there is an overlap of notation considering the symbol θ: To be consistent with the notation in [46] we chose this symbol. However, outside this section it has a different meaning, which we introduce later.

f is a function that computes the probability of a given evidence: $f(\mathbf{e}) = Pr(\mathbf{e})$. Other probabilistic results, in particular marginals, can be computed with *derivatives* of f: $Pr(X_i = x\,|\,\mathbf{e}) = \frac{1}{f(\mathbf{e})}\frac{\partial f}{\partial \lambda_x}(\mathbf{e})$. To illustrate, consider the example net in figure 3.3 with random variables A, B. Both take two values, $a, \overline{a}$ and $b, \overline{b}$ respectively. The corresponding polynomial is

$$f = \lambda_a\lambda_b\theta_a\theta_{b\,|\,a} + \lambda_a\lambda_{\overline{b}}\theta_a\theta_{\overline{b}\,|\,a} + \lambda_{\overline{a}}\lambda_b\theta_{\overline{a}}\theta_{b\,|\,\overline{a}} + \lambda_{\overline{a}}\lambda_{\overline{b}}\theta_{\overline{a}}\theta_{\overline{b}\,|\,\overline{a}}$$

With evidence $\mathbf{e} = ab$, for example, we get $f(\mathbf{e}) = \theta_a\theta_{b\,|\,a}$. If we want to know $Pr(A = a\,|\,b)$, now with evidence $\mathbf{e} = b$, we can compute this with the derivative for λ_a: $Pr(A = a\,|\,b) = \frac{1}{\lambda_a\theta_a\theta_{b\,|\,a}+\lambda_{\overline{a}}\theta_{\overline{a}}\theta_{b\,|\,\overline{a}}}\frac{\partial f}{\partial \lambda_a}(b) = \frac{\theta_a\theta_{b\,|\,a}}{\lambda_a\theta_a\theta_{b\,|\,a}+\lambda_{\overline{a}}\theta_{\overline{a}}\theta_{b\,|\,\overline{a}}}$, since $\frac{\partial f}{\partial \lambda_a} = \lambda_b\theta_a\theta_{b\,|\,a} + \lambda_{\overline{b}}\theta_a\theta_{\overline{b}\,|\,a}$. We can draw the connection to plan assessment here if we imagine that the products $\lambda_a\lambda_b\theta_a\theta_{b\,|\,a}$,

$\lambda_a \lambda_{\bar{b}} \theta_a \theta_{\bar{b}\,|\,a}$, etc. are the probabilities of possible system trajectories. In the example above, $\theta_a \theta_{b\,|\,a}$ would be goal achieving (for the goal $A = a$), while $\lambda_a \theta_a \theta_{b\,|\,a}$ and $\lambda_{\bar{a}} \theta_{\bar{a}} \theta_{b\,|\,\bar{a}}$ are all the trajectories consistent with observation $\mathbf{e} = b$.

The polynomial can be efficiently represented, and more importantly, evaluated and differentiated with an arithmetic circuit. An arithmetic circuit is a rooted, directed acyclic graph with product nodes, addition nodes and terms such as λ_a, $\theta_{a\,|\,b}$ or $\lambda_a \theta_a$ as leaves. Figure 3.4 shows an arithmetic circuit for the simple example net in figure 3.3. Arithmetic circuits are not unique; consequently, one of the interesting problems related to arithmetic circuits is finding a compact one for a given Bayesian net.

The work in [46] details how the global structure of a net given as its join tree can be exploited to create compact arithmetic circuits. This is possible because join trees actually embed arithmetic circuits. Furthermore, it has been shown that the message passing stage 1 of junction tree algorithms (towards the root of the join tree) corresponds to computing $f(\mathbf{e})$ and stage 2 (outwards to the leaves of the join tree) to computing the derivatives of f. However, arithmetic circuits go beyond the possibilities of join trees. For example, the authors of [46] show how to exploit local structure in the form of repeated probability values to create more compact arithmetic circuits.

We use the publicly available implementation provided by the authors of [38, 91, 37, 46], Ace 2.0 [3]. The tool can compute marginals out-of-the-box. However, it apparently cannot compute MAP right away. This means we can solve plan assessment only partially with this tool. Still it is worthwhile investigating this off-the-shelf tool: First of, as recent results show [37], it performs very good on particularly difficult problems. And second, an algorithm to compute MAP exactly with arithmetic circuits, ACEmap, has already been introduced in [90]. It seems a matter of time until this algorithm will be included in the official public distribution.

Although it seems that arithmetic circuits have not yet been used to compute MPE, we think it should be possible to implement this with reasonable effort, considering that generalized junction tree algorithms can solve this problem by essentially replacing summation with maximization. Specifically, we speculate that arithmetic circuits could be used in a similar way to compute MPE by replacing the addition nodes with maximization nodes and adapting the evaluation procedures given in [46].

[3]`http://reasoning.cs.ucla.edu/ace/` (03.2011)

Sampling and SampleSearch

Sampling is an attractive class of anytime algorithms to compute approximate solutions to the probabilistic problems defined in section 3.5. In particular, in section 7.2 we investigate how to compute success probabilities with them with a stochastic accuracy guarantee. These algorithms approximate the full joint distribution $Pr(X_1, \ldots, X_n) = \Pi_i Pr(X_i \mid \mathit{parents}(X_i))$ by randomly generating a number of full assignments that accord to the given conditional probabilities in P and are consistent with the evidence. Among these sampled assignments appear those more often, which are more likely due to $Pr(X_1, \ldots, X_n)$. This allows to compute the probability of arbitrary partial assignments given the evidence, in particular, marginals.

In the context of plan assessment, sampling full assignments, in essence, corresponds to sampling potential system trajectories. For example, if we have a goal encoded by $G^{t_{\text{end}}} = \mathsf{marked}$, for example the maze product being flawless and finished by t_{end}, then we would like to compute $Pr(G^{t_{\text{end}}} = \mathsf{marked} \mid \mathbf{o}^{0:t})$ (where the observations $\mathbf{o}^{0:t}$ are the evidence $\mathbf{e}$). Let $G^{t_{\text{end}}} = \mathsf{marked}$ be denoted by g. Then we can approximate the probability in question as $Pr(g \mid o^{0:t}) \approx \frac{m_g}{m}$, where m is the number of all sampled trajectories and m_g the number of sampled trajectories where the maze product is considered ok at time t_{end}.

Sampling algorithms differ mainly in how exactly they use the conditional probability functions in P to correctly sample the full assignments. Popular methods are Gibbs sampling [36], likelihood weighting [71], backward simulation [70] and, more recently, SampleSearch [75]. Here we are mainly interested in SampleSearch because it seems especially well suited for models that we consider in the context of plan assessment. These models are to a large extent deterministic, because they model systems with mostly deterministic behavior and only a small amount of uncertainty. For these kind of models, sampling can quickly run into the so-called rejection problem: A large number of samples is drawn that, in the end, have probability 0 because of some deterministic conditional probability function evaluating to 0, and have to be rejected. SampleSearch remedies this by systematically searching for samples consistent with the deterministic parts of the model. It achieves this by combining sampling with standard constraint reasoning techniques.

3.6. Modeling Complex, Uncertain System Behavior with PHCA

In this section, we will explain how systems such as manufacturing plants can be modeled with probabilistic hierarchical constraint automata (PHCA), the model-framework chosen for this work. PHCAs are part of a toolset developed under the paradigm of model-based programming. Before we dive into the details of PHCA models, we will briefly state some important points about this paradigm. Formal definitions of PHCAs and closely related concepts will not be given here, instead we focus on explaining things informally with examples and postpone the formalities to chapter 4. There, they will be used as basis for a formal definition of the plan assessment problem.

3.6.1. Model-Based Programming

Model-based programming was developed by Williams et al. [164, 163, 162, 132] as extension of existing engineering concepts towards the realm of artificial intelligence problems. Most notably, the paradigm introduces a novel concept of programming technical systems based on a model of its internal, not directly observable states. The model describes these *hidden* states, transitions between them and how states connect to sensor signals. The transitions can be conditioned on command variables and may be probabilistic. This allows to model possible failures of system components. Control programs for systems are then written based on the hidden states, rather than directly on sensor signals and command variables. An underlying execution component then constantly deduces, from observations and the model, the most likely hidden state and uses it as starting point to find actions to achieve goal states derived from the model-based control program. The idea is realized in the reactive model-based programming language (RMPL), which compiles to a compact encoding of hidden Markov models, the probabilistic hierarchical constraint automata. Listing 3.1 shows a short RMPL control code sample reproduced from [163].

The language is adapted to what engineers are used to from engineering languages such as Statecharts[78] or Esterel[20], e.g., parallel execution, hierarchical composition, complex conditioned execution, etc. This moves engineering a big step towards controlling systems with AI techniques such as belief state update [163]. This thesis ties in with this paradigm by developing plan assessment based on PHCAs. We will focus on the modeling part of this paradigm, i.e. we do not address issues related to control programs.

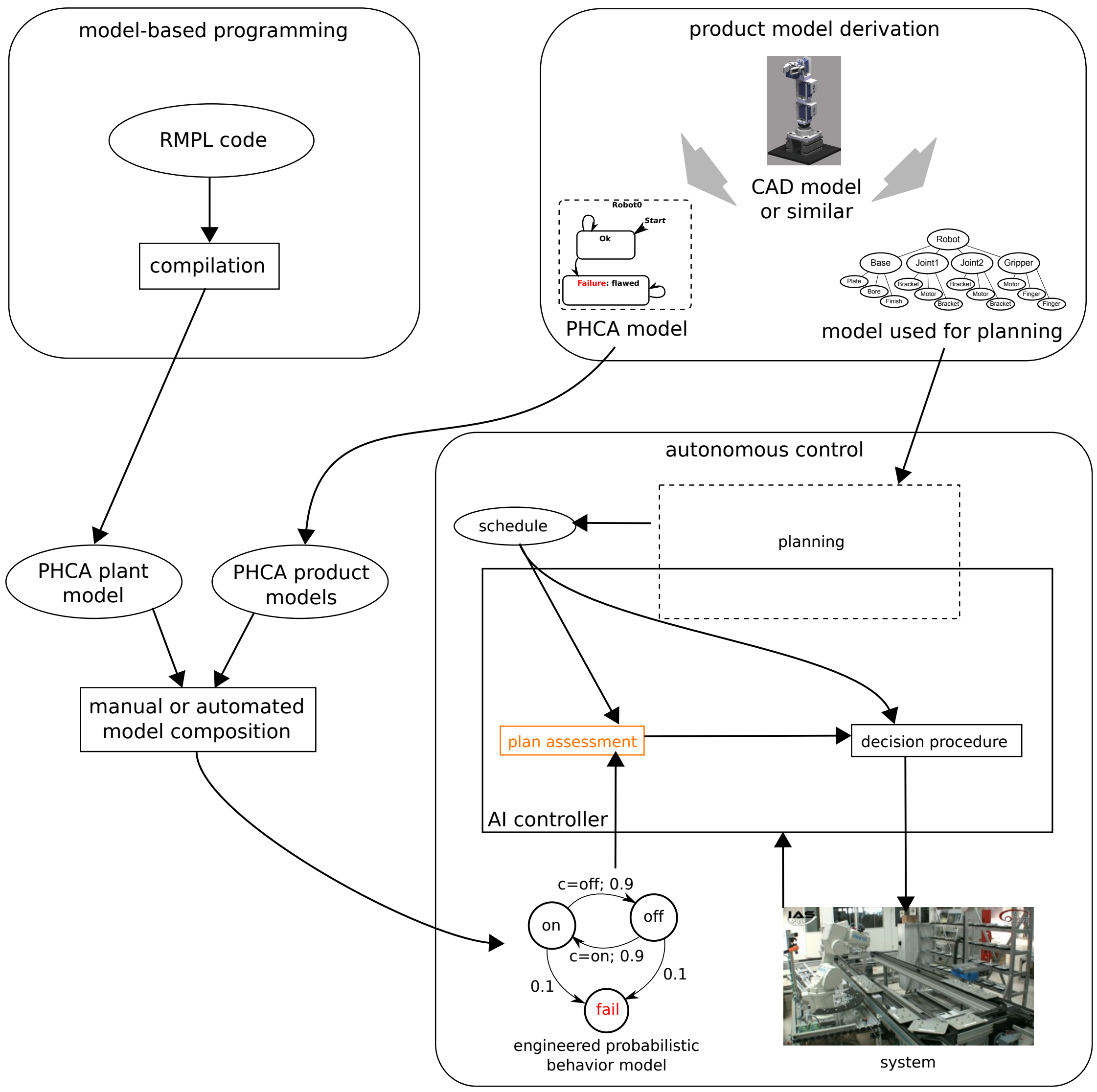


Figure 3.5.: Illustration of how model-based programming, product modeling, planning, plan assessment and autonomous decision making could interact and work together.

```
OrbitInsert () :: {
    do {
        EngineA = Standby,
        EngineB = Standby,
        Camera = Off,
        do {
            when EngineA = Standby ∧ Camera = Off
                donext EngineA = Firing
        } watching EngineA = Failed,
        when EngineA = Failed ∧ EngineB = Standby ∧ Camera = Off
            donext EngineB = Firing
    } watching EngineA = Firing ∨ EngineB = Firing
}
```

Listing 3.1: A piece of control code in the reactive model-based programming language (RMPL).

Figure 3.5 illustrates the interplay of model-based programming, product modeling, planning, plan assessment (highlighted) and autonomous decision making in the types of autonomous manufacturing scenarios we consider. Section 4.2 has more on product models and their import for plan assessment.

3.6.2. PHCA Modeling

Now we describe, informally with examples, how system models using PHCAs can be built. We use the plant of our example scenario in section 2.1.3 for illustration. Figure 3.6 shows a PHCA model of this plant. We will look into the concepts of locations and behavior constraints, hierarchical composite locations, probabilistic guarded transitions and how start states are chosen. In particular, we will (informally) introduce the important concept of the PHCA state. When it comes to solving the plan assessment problem we are interested in system trajectories, which are sequences of PHCA states. We base our explanations of general PHCA concepts on information found in [129, 164, 163, 162].

Locations and Behavior Constraints The basic element of PHCA system models is the location. For instance, the assembly station model of the example plant, shown in figure 3.7, consists of five locations. One of them is itself composed of two other locations, which means it is a composite location. PHCA locations are used to encode system modes. A mode characterizes the internal, and thus usually hidden state of a technical system. For example, the state of the machining station 0 (see figure 3.9), after transitioning from Idle,

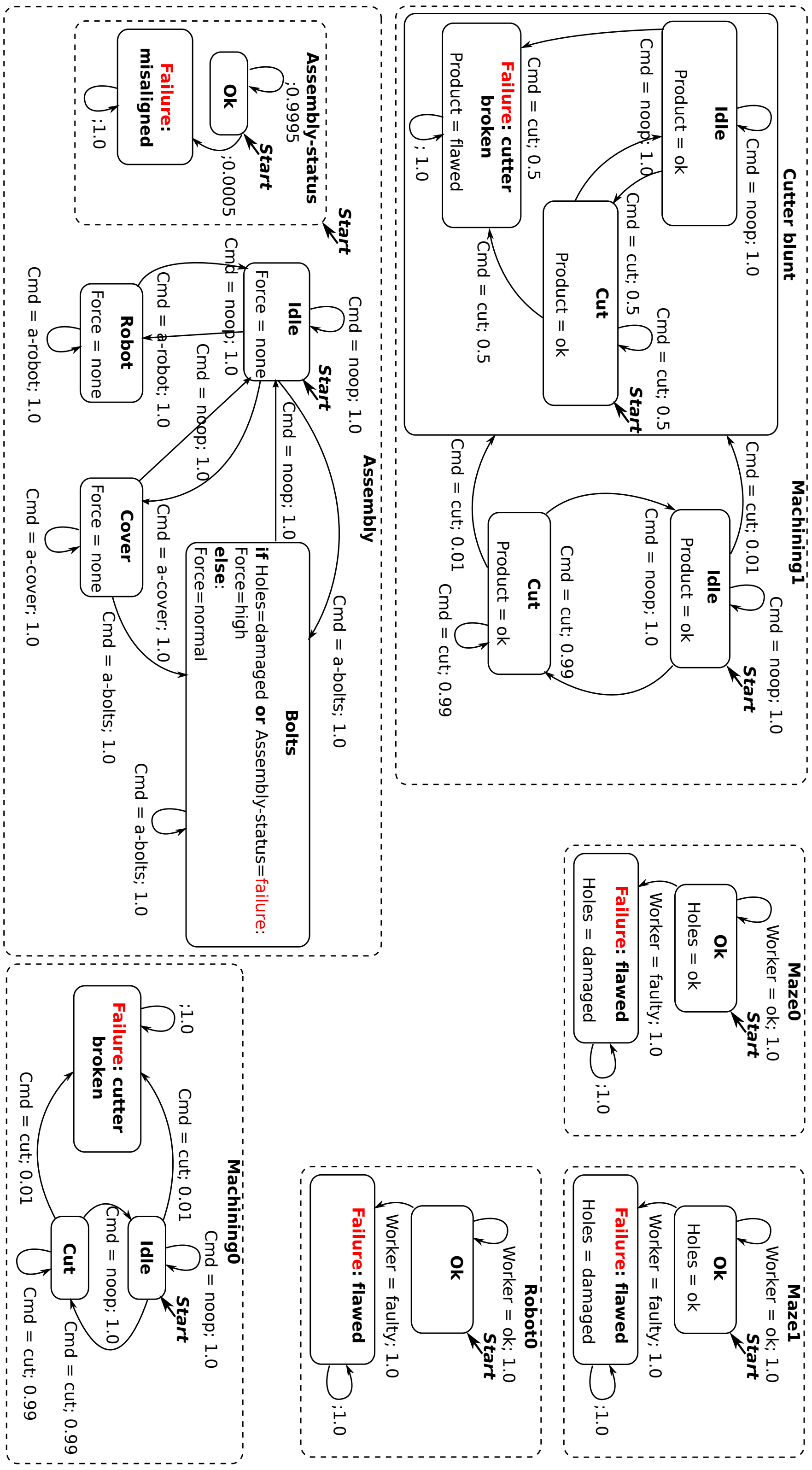


Figure 3.6.: A PHCA model of a plant like the one in the example scenario in section 2.1.3.

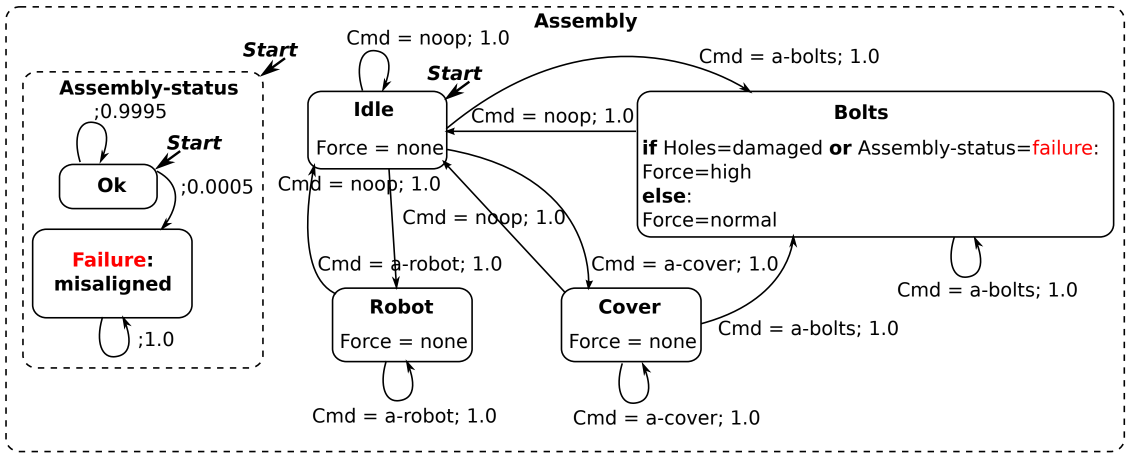


Figure 3.7.: Assembly station sub-automaton.

may be nominal (Cut) or a failure state (Failure : cutter broken), but we cannot observe that directly. The state, or sequence of states, must be estimated from partial information, in our example a sensor alarm elicited by the assembly station if too much force was applied. In the context of PHCA models we refer to PHCA states rather than modes. A PHCA state is a set of locations that is considered active or marked. A sequence of such sets of marked locations forms a system trajectory, the key computational element we are interested in.

Since PHCA states, and therefore location markings, are hidden, it is crucial to model the relationship with observations that can be expected in certain locations. This is achieved by encoding possible observations with constraints over variables with finite domains. Each PHCA model has a set of designated variables, which can be used to encode observations, commands or model internal dependencies. Each location has one constraint, called behavior constraint, which encodes the observations possible in the hidden state encoded by this location. It can be as simple as an assignment, or more complex, as the behavior constraint of the location Bolts of the assembly station:

Bolts
if Holes=damaged **or** Assembly-status=failure:
Force=high
else:
Force=normal

The location encodes the assembly pushing bolts or pins into the maze to fixate the glass cover on top of the alloy base plate of a maze. Its behavior constraint encodes that an alarm will be seen if the maze being worked has damaged holes or if the assembly station itself is faulty, i.e. the assembly robot arm is misaligned. In general, constraints

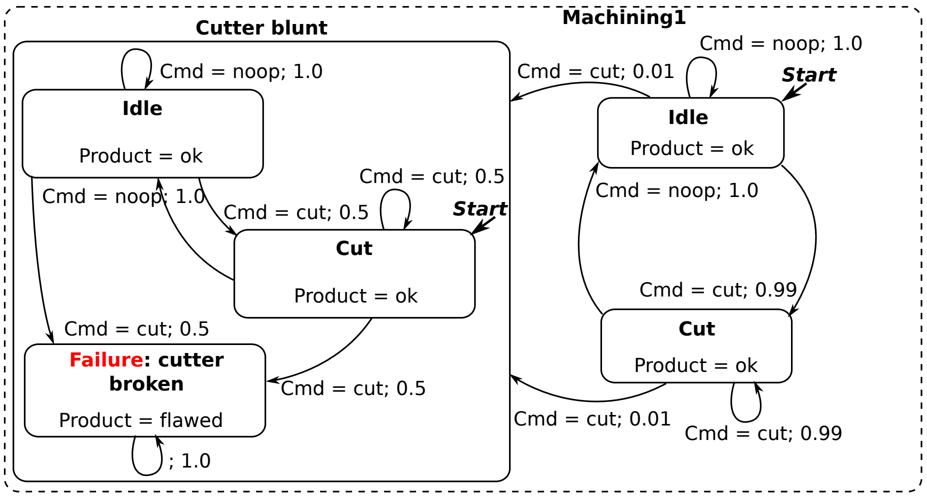


Figure 3.8.: Machining station sub-automaton that has a more complex fault model: it models the cutter becoming blunt before breaking.

in PHCAs are encoded as propositional formulas, where propositions are assignments of values to variables. The propositions can be negated with $\neg$ and combined with $\vee, \wedge$. In the graphical depictions we took the liberty to use a somewhat more expressive language to make constraints such as the one above easier to read. Its propositional formula is (in conjunctive normal form)

$$\begin{aligned}(\mathsf{Holes} = \mathsf{damaged} \vee \mathsf{Assembly\text{-}status} = \mathsf{failure} \vee \mathsf{Force} = \mathsf{normal}) \quad &\wedge \\ (\mathsf{Holes} = \mathsf{ok} \vee \mathsf{Force} = \mathsf{high}) \quad &\wedge \\ (\mathsf{Assembly\text{-}status} = \mathsf{ok} \vee \mathsf{Force} = \mathsf{high}) \quad &\end{aligned}$$

Behavior constraints do not hold all the time, but become active, or enforced, if their location becomes marked, i.e. is part of the PHCA state. This allows a very flexible modeling of complex observation interactions.

Composite Locations Every location is itself a PHCA, which in particular means they can be hierarchically composed of other locations. Locations therefore fall into the two classes of composite locations, which are composed of other locations, and primitive locations, which do not contain any other locations. The composition hierarchy is rooted in the complete model itself, i.e. the model is the root location. The root location is composed of the locations, or sub-automata, for top level system components. In our example, this corresponds to plant stations and product models (maze, robot). The products are being modeled with a location composed of two primitive locations, one

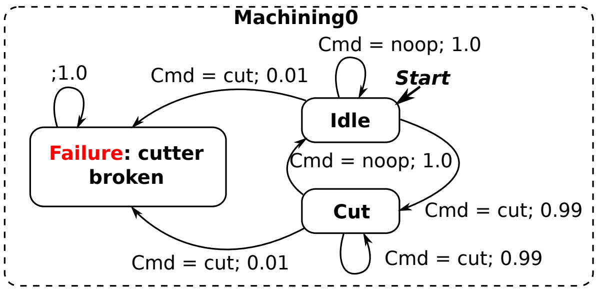


Figure 3.9.: Machining station sub-automaton.

modeling the (yet unfinished) product being fine during the production process, and one modeling that something went wrong and flawed the product. Machining station 1 and the assembly station are examples of composite locations containing yet other composite locations. Machining station 1 contains a composite location which is the target of transitions, meaning it becomes marked only when being transitioned to. All other composite locations seen so far are also start locations, which means they become marked right from the beginning. One important consequence of this is that these composite locations are marked concurrently, i.e. they “run” in parallel. We detail this some more later in this section.

Probabilistic Transitions and Guard Constraints In PHCAs, the potentially uncertain evolution of the system state, represented as the PHCA state, is encoded with guarded, probabilistic transitions. Guarded means that transitions may only be taken if certain conditions are met, e.g. a certain command is given. Probabilistic means that even if said condition is fulfilled, the transition may be chosen randomly according to a given probability distribution.

A transition has a single source, which must be a primitive location, and has at least one target. The target may be an arbitrary location, primitive or composite, anywhere in the model. The original PHCA formalism allows for multiple transition targets at once. The PHCA models considered in this work, however, only have transitions with single targets.

Guards are defined as constraints over, for example, command variables. However, one can also formulate more complex constraints over any of the variables of the PHCA model. A guarded transition may be taken (potentially subject to random choice) if its guard constraint is entailed by the behavior constraints of marked locations. This means the guard constraint must be satisfied for all assignments to PHCA variables that are consistent with the behavior constraints of currently marked locations.

Each transition has a guard constraint and a probability value. Transitions are probabilistic in the sense that all outgoing transitions of a location, whose guards are entailed at a given time point, must form a probability distribution. A practical way to achieve this is to use mutually exclusive guards to partition the set of outgoing transitions such that all partitions but one have inconsistent guards at any time point. Then, probability distributions can be defined over these partitions. As an example, consider again machining station 0 and its location Idle, which has three outgoing transitions (see figure 3.9). They are partitioned by mutually exclusive guards Cmd = noop (1 transition) and Cmd = cut (2 transitions). "noop" means "no operation". As for the distributions, the single transition of the partition according to Cmd = noop has of course probability 1.0. The two transitions of Cmd = cut have probability 0.99 (Idle to Cut) and 0.01 (Idle to Failure : cutter broken).

Initial PHCA State and Start Locations Every location may be a start location. Being a start location means two things:

1. The location is chosen for the initial marking of the PHCA at the beginning, i.e. the first time point.
2. The location is becoming marked if its parent composite location is becoming marked, e.g. because it is being transitioned to.

In other words, start locations determine how possible evolutions of system behaviors start, either globally at the very beginning, or locally within composite locations. PHCA allow to define probability distributions over sets of start locations, which means that start locations can be chosen probabilistically as well as that multiple start locations may be initially marked at once. In practice, distributions will be used that are easy to factorize, since they are computationally much easier. In the models in this work we only use deterministic starts.

3.6.3. PHCA State Evolution as Possible, Synchronously Parallel Evolutions

The concurrent marking of PHCA locations corresponds to synchronous parallel execution. For example, the concurrent marking of the composite locations encoding the assembly station and the machining station (0 or 1) is analogous to them being synchronously executed, by e.g. sending commands to them at each tick of a clock. However, the PHCA model itself is **not** executed. When talking about the model, it is better to talk about possible, synchronously parallel evolutions of sub-components induced by execution. That is, when executing pre-planned operation steps, the PHCA model represents the possible, synchronously parallel behaviors of the components of a system. Why is this distinction important?

This distinction affects how to interpret communication among parts of a PHCA model and, in particular, conflicts such as two components writing to the same variable. The simple explanation is: a communication conflict such as two components writing *different* values to the same variable cannot happen, unless the model is incorrect. Let's examine this in more detail. Components of a system, modeled as composite locations, can communicate through global PHCA variables. However, this is more of an abstract dependency among these locations rather than communication in terms of reading and writing variables. The behavior constraints of these locations determine, for a given PHCA state that marks these locations, which values variables can take. For example, if in a given PHCA state the assembly station transitions to location Idle, it enforces its behavior constraint Force = none (This enforcing could be seen as an abstract "writing" of, in this case, the value none to variable Force). It is important to note that the order of constraint enforcing doesn't matter, it can be seen as one big, atomic step. It could now be that two constraints are enforced that disagree on the value of a certain variable. For example, another location's behavior constraint could be Force = high (given that Force is a global variable). If this location also becomes marked in the mentioned PHCA state, these two constraints become inconsistent, rendering the PHCA state inconsistent. In probability terms, any system trajectory containing this PHCA state has probability 0 of occurring. In constraint based approaches to plan assessment, similar conflicts are actually used to rule out impossible system trajectories. However, if trajectories should in fact be possible that mark the two locations concurrently, then something is wrong with the PHCA model or, worse, the system design.

3.7. Bayesian Logic Networks: Bayesian Networks Generalized To First-Order Logic

The PHCA modeling framework, like any automata framework, doesn't explicitly represent time steps. From the point of view of logic PHCA models implicitly *quantify* over time, which makes the underlying logic a first-order logic. This indirectly affects how we try to solve plan assessment, given that we want to automatically translate PHCA models to, for example, probabilistic models that off-the-shelf tools understand. The problem is to choose a probabilistic modeling framework as a translation target. One could simply use Bayesian nets as target, which however are propositional in nature (i.e. like boolean formulas without quantifiers). Since PHCAs are first-order in nature, they would be conceptually closer to first-order probabilistic frameworks. It turns out that just such a framework exists, which defines a first-order generalization of Bayesian nets, called Bayesian logic networks (BLNs) [95]. These are thus a natural choice over Bayesian nets.

Bayesian logic networks can be seen as templates for the construction of Bayesian networks. A BLN may be instantiated to either a ground Bayesian net or a ground mixed network that explicitly represents logical constraints on the distribution [124]. Probabilistic inference is often performed in such ground models. This section reproduces and explains the elements of BLNs essential to this work. For a detailed account we refer to [95] and [96]. Note that parts of this section have been published earlier in [118].

Frameworks such as the BLN framework have been developed and used in probabilistic reasoning for the last 10 years, and they are becoming increasingly important. They are known under the term of statistical relational reasoning, which means that instead of modeling concrete objects, abstract relations between classes of objects are modeled. For example, with Bayesian nets one would model transitions for each concrete time point. With statistical relational models, one instead can abstractly describe the transition relation between states, independent of time. This is much more in line with the automata-style modeling in PHCA. Going from concrete objects to abstract relations can be understood as "lifting" the modeling from propositional logic to first-order logic. In particular, quantifiers over objects are introduced.

An added benefit of BLNs as translation target is a greater flexibility with respect to the choice of inference methods. Specifically, recent research in statistical relational reasoning tries to (partially) lift the inference problem to the first-order level in order to exploit repeated sub-structures in ground models [137]. By translating to BLNs, the gate is already open for future work exploiting these novel algorithms.

3.7.1. Bayesian Logic Networks Explained with an Example

We illustrate the BLN framework with a manually created model for a class of scenarios of the type described in section 2.1.3. This model was created as part of joint work in [118]. It simplistically represents every station and every product with two states, Ok or $\neg\mathsf{Ok}$, i.e. broken. Relations such as a station working a product are represented as first-order predicates, encoding templates for random variables. The model is shown in figure 3.10.

Key elements of a BLN $\mathcal{B}$ are abstract random variables, typed entities, fragments and first-order logical formulas. Abstract random variables are parametrized random variables that correspond to either predicates or non-boolean functions. They encode, for example, templates for partial states such as a station being faulty, abstract relations such as stations working on specific products or a template for a probabilistically chosen transition. The parameters or arguments of an abstract random variable refer to abstract, typed entities, which allows, in particular, quantification over time. If we see arguments as meta-variables in first-order predicates, such as o and t_1 in the predicate $\mathit{mazeOK}(o, t_1)$, then entities are their valuations, e.g. $M0, M1$ for two maze entities and $T0, T1$ for two time point entities.

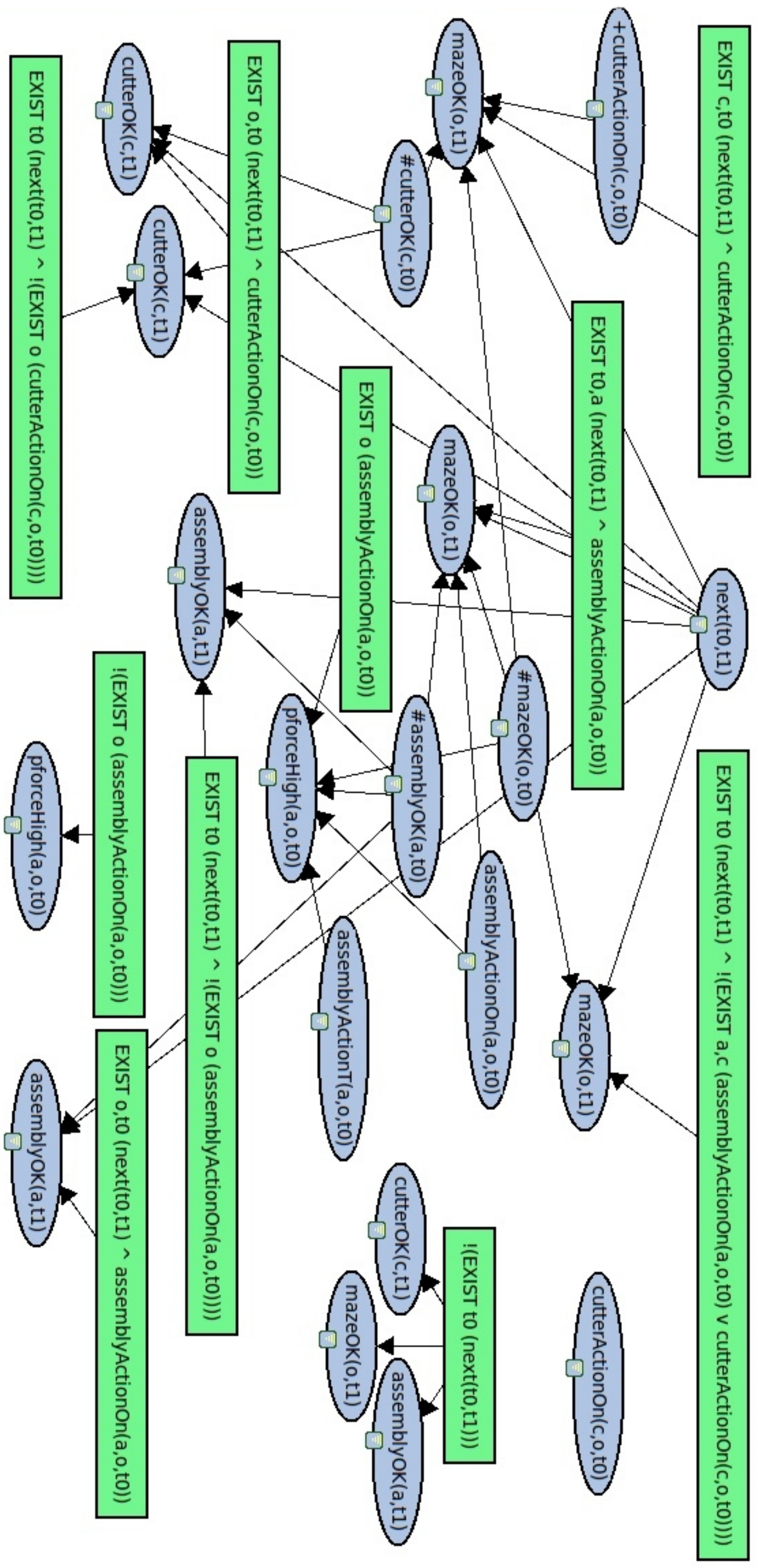
A fragment associates an abstract random variable with a conditional probability table. They encode templates for concrete conditional probability functions. On the graphical level, fragments (ellipses in figure 3.10) correspond to abstract Bayesian nodes, and the arcs between them likewise to abstract conditioning relations. The set of all fragments collectively defines a template for probability distributions. The applicability of a fragment during instantiation may be restricted by (mutually exclusive) first-order logic preconditions (boxes in figure 3.10), which allows to construct more flexible templates. E.g., the conditional probability table chosen for concrete instances of $\mathit{mazeOK}(o, t_1)$ depends on whether there exists a time point instance that precedes $t1$, expressed in the precondition $\exists t_0 : \mathit{next}(t_0, t_1) \wedge \mathit{assemblyActionOn}(a, o, t_0)$ (a, o are implicitly $\forall$-quantified). The second predicate specifies the additional condition that at the previous time point the maze was worked by an assembly station.

First-order logical formulas may be specified in the model to express hard relations among predicates using the standard set of expressive logical operators $\vee, \wedge, \neg, \Rightarrow, \Leftrightarrow$ and quantifiers $\forall, \exists$. The relations are hard as opposed to probabilistic, soft, relations expressed through fragments. Every such formula can be converted into an equivalent set of fragments. However, often global hard relations are more conveniently expressed this way.

Formally, a Bayesian logic net is a triple $\mathcal{B} = (\mathcal{D}, \mathcal{F}, \mathcal{L})$, usually accompanied by a knowledge base DB. $\mathcal{D}$ contains, among other things, the entity types and predicate signatures for abstract random variables. Most of $\mathcal{D}$ can be reused across a wide range of possible BLNs. The code excerpt in listing 3.2 shows definitions occurring in $\mathcal{D}$ for the example in figure 3.10.

$\mathcal{F}$ is the set of fragments, and $\mathcal{L}$ the set of first-order logic formulas. The knowledge base defines existing objects or entities for the first-order logic formulas and fragments as well as known facts about relations among these entities.

It is important to distinguish arguments of abstract random variables and their valuations, entities, from variables X_i and values x as they appear in constraint or Bayesian nets.



EXIST c,t0 (next(t0,t1) ^ cutterActionOn(c,o,t0))
+cutterActionOn(c,o,t0)
mazeOK(o,t1)
#cutterOK(c,t0)
EXIST o,t0 (next(t0,t1) ^ cutterActionOn(c,o,t0))
cutterOK(c,t1)
cutterOK(c,t1)
EXIST t0 (next(t0,t1) ^ !(EXIST o (cutterActionOn(c,o,t0))))
EXIST t0,a (next(t0,t1) ^ assemblyActionOn(a,o,t0))
EXIST o (assemblyActionOn(a,o,t0))
mazeOK(o,t1)
assemblyOK(a,t1)
next(t0,t1)
#mazeOK(o,t0)
#assemblyOK(a,t0)
pforceHigh(a,o,t0)
!(EXIST o (assemblyActionOn(a,o,t0)))
pforceHigh(a,o,t0)
EXIST t0 (next(t0,t1) ^ !(EXIST o (assemblyActionOn(a,o,t0))))
assemblyActionOn(a,o,t0)
assemblyActionT(a,o,t0)
EXIST t0 (next(t0,t1) ^ !(EXIST a,c (assemblyActionOn(a,o,t0) v cutterActionOn(c,o,t0))))
mazeOK(o,t1)
EXIST o,t0 (next(t0,t1) ^ assemblyActionOn(a,o,t0))
assemblyOK(a,t1)
cutterOK(c,t1)
!(EXIST t0 (next(t0,t1)))
mazeOK(o,t1)
assemblyOK(a,t1)
cutterActionOn(c,o,t0)

Figure 3.10.: Manually created Bayesian logic network that models a class of manufacturing scenarios with machining (called cutter) and assembly stations, and their interaction with maze products.

```
Type objType_c;
Type objType_a;
Type objType_o;
Type objType_t;
guaranteed domAction AssemblePins, AssembleCover;
logical Boolean next(objType_t,objType_t);
random Boolean cutterOK(objType_c,objType_t);
random Boolean assemblyOK(objType_a,objType_t);
random Boolean mazeOK(objType_o,objType_t);
logical Boolean cutterActionOn(objType_c,objType_o,objType_t);
logical Boolean assemblyActionOn(objType_a,objType_o,objType_t);
random domAction assemblyActionT(objType_a,objType_o,objType_t);
random Boolean pforceHigh(objType_a,objType_o,objType_t);
```

Listing 3.2: Code excerpt of Bayesian logic network entity types and predicate signatures.

The former exist on the first-order level and can be considered as sort of meta-variables and meta-values. The latter exist on the propositional level of, in particular, Bayesian nets. To illustrate let us look at the predicate $\mathit{mazeOK}(o, t_1)$. It is first-order because it is a function over arguments, or meta-variables, o and t_1. When its arguments are instantiated with, for example, $M0$ and $T1$, we receive what is called a ground predicate $\mathit{mazeOK}(M0, T1)$. This ground predicate is propositional, and directly corresponds to a random boolean variable in a Bayesian net. Ground predicates and the process of grounding will be explained in more detail later.

The difference is reflected in the notation, as arguments are always represented with lower case letters, entities with upper case letters. To avoid confusion with constraint or Bayesian net variables and values, arguments and entities will always appear in the context of first-order predicates and we avoid subscript indices for entities. For the example, the letters a, c, o, t are used to respectively represent arguments for assembly stations, machining stations (cutters for short), mazes (objects/products being worked) and time points.

The manually created example model in figure 3.10 realizes the evolution of a system over time with two abstract arguments t_0 and t_1, representing successive time points, and abstract random variables relating to them for successive actions, states, etc. A time line is enforced through abstract random variable $\mathit{next}(t_0, t_1)$, which encodes that t_0 precedes t_1. When instantiating, successiveness of time points $T0, T1, \ldots$ is ensured by clamping $\mathit{next}(T0, T1)$, $\mathit{next}(T1, T2)$, and so on to true in the knowledge base. Uncertain station evolution is modeled in a simplified way, using two abstract random variables:

$\mathit{assemblyOK}(a,t)$ and $\mathit{cutterOk}(c,t)$ for assembly and machining stations. The failure probabilities 0.03 and 0.01 for machining and assembly stations are encoded in the fragments for these abstract random variables. Product state evolution is modeled in a similar way, i.e. we have the abstract random variable $\mathit{mazeOK}(o,t)$ for the state of the class of maze products. Their fragments are different in that they don't encode any uncertainty. The force alarm observations are encoded as evidence abstract random variable: $\mathit{pforceHigh}(a,t)$ encodes that the force measured at the assembly station a at time t was too high (if `true`). To encode actions, relations encode assembly and machining stations working mazes at a certain time, i.e. abstract random variables $\mathit{assemblyActionOn}(a,o,t)$ and $\mathit{cutterActionOn}(c,o,t)$. The complex relation that a force alarm can be triggered by cutter-damaged holes as well as a misaligned assembly can be expressed as a first-order logic formula:

$$\mathit{assemblyActionOn}(a,o,t) \Rightarrow (\mathit{pforceHigh}(a,t) \Leftrightarrow (\neg \mathit{assemblyOK}(a,t) \vee \neg \mathit{mazeOK}(o,t)))$$

3.7.2. Inference in Grounded Bayesian Logic Networks

As mentioned, inference directly in BLNs is possible with lifted inference [137]. However, for this work we focus on solving problems by instantiating BLNs to Bayesian nets.

A BLN is an abstract model that represents a class of concrete models. Grounding refers to converting the BLN to an instance of this class using information about existing objects and entities from the knowledge base DB. More specifically, entries in DB are of the form $P(E1,\ldots,En) = V$, where P is an arbitrary predicate, $E1,\ldots,En$ entities instantiating its arguments and V is some value (often `true` or `false`). The entities $E1,\ldots,En$ are used to instantiate the according predicates P in the fragments. The assignment of values to grounded evidence predicates, representing evidence $\mathbf{e}$, can be changed after the grounding to account for new evidence. The result of this process is a mixed network, i.e. a Bayesian net with added (propositional) logical constraints. The logical constraints are further converted to random variables and deterministic conditional probability tables, i.e. which evaluate only to 0 and 1. Specifically, for each instance of a logical formula a conditional probability table and an associated boolean random variable is added. These variables are called auxiliary variables, and they are clamped to `true`, thereby enforcing their respective, concrete logical rules. As an example, consider the formula $\forall a : P(a) \Rightarrow Q(a)$, with P, Q being predicates. An instance would be $P(A0) \Rightarrow Q(A0)$ and would yield the following conditional probability table:

$P(A0)$	true		false	
$Q(A0)$	true	false	true	false
true	1	0	1	1
false	0	1	0	0

The resulting Bayesian net is called an auxiliary Bayesian net, due to the auxiliary variables added to encode the logical formulas. The joint distribution $Pr(X_1, \ldots, X_n \mid \mathbf{X}_{\text{aux}})$ over all variables X_i in that Bayesian net (without the auxiliary variables), conditioned on the auxiliary variables is the full joint distribution for this instance ($\mathbf{X}_{\text{aux}}$ being the vector of all auxiliary variables). For further details on grounding BLNs to Bayesian nets we refer to [96].

4. Probabilistic Assessment of Plans for Autonomous Systems

In this section we give a formal definition of the plan assessment problem based on the formalism of probabilistic hierarchical constraint automata (PHCA) [162, 163]. We analyze the implications of the specific context of autonomous manufacturing and how plan assessment can support autonomous decisions. Then we look at how plan assessment relates to similar problems in AI and other fields. Parts of this chapter have been submitted or published earlier: section 4.1 is part of [117] and parts of section 4.2 have been published in [121].

4.1. Problem Definition: Diagnosing System Faults and Estimating Operation Success

Plan assessment extends the maximum probability diagnosis problem [146] towards additionally computing success probabilities for given goals. As we have seen in the introduction the problem arises in scenarios that are characterized by three properties: 1) A rigidly designed system with remaining uncertainties executes 2) pre-planned operations to 3) achieve explicitly defined goals. These aspects are reflected in these three formal elements: 1) A system model, in our case a PHCA M_{PHCA}, 2) an operation sequence or plan $\mathcal{P}$ of operation steps, and 3) a set of goals $\{G_i\}$.

4.1.1. System Model

PHCA models define automata states, called *locations*, and transitions between them. The transitions may be guarded and probabilistic, locations may be composed of sub-locations. We now give the formal definition of PHCA. We modified the definition provided in [129] to fit this thesis' notation and context, and to be somewhat more detailed and clearer.

Definition 3. A PHCA is a tuple $\langle \Sigma, P_\Xi, \Pi, O, \mathit{Cmd}, \mathcal{C}, P_T \rangle$:

- $\Sigma = \Sigma_p \uplus \Sigma_c$ is a set of locations, partitioned into *primitive locations* Σ_p and *composite locations* Σ_c, where a composite location represents a PHCA whose elements are subsets of the elements of the containing PHCA. A location may be marked or unmarked. A marked location represents an active execution branch. A *marking* m^t (at time t) is given as a subset of Σ.
- $P_\Xi(\Xi_i)$ denotes the probability that $\Xi_i \subseteq \Sigma$ is the set of start locations (initial state).
- $\Pi = O \uplus \mathit{Cmd} \uplus \mathit{Dep}$ is a set of variables with finite domains. O is the set of observation variables, Cmd is the set of action or command variables, and Dep is a set of dependent hidden variables which are connected to other variables via constraints.
- $\mathcal{C}$ is the set of finite domain constraints defined over Π, which comprises behavior constraints of locations and guard constraints of transitions. A location's behavior constraint serves to define the observations that are consistent with the location; a transition's guard defines the conditions under which the transition can be taken (usually depending on commands). The constraints are expressed as propositional formulas (using the usual operators $\vee, \wedge, \neg$) over variable assignments.
- $P_T[l]$ (defined for each $l \in \Sigma_p$) is a probability distribution over the subset of the set of transitions $\mathcal{T}$, which contains transitions leading away from l_i whose guards (elements of $\mathcal{C}$) are entailed given the current state. A transition $\tau \in \mathcal{T}$ is defined as function τ: $\mathcal{C} \rightarrow 2^\Sigma$ that maps the transition's guard constraint to the transition's target if the guard is entailed. If it's not entailed, it maps to the empty set, which means that this thread of execution stops. We can construct a global transition function T: $\Sigma_p \times \mathcal{C} \rightarrow 2^\Sigma$ from single transitions randomly chosen from $P_T[l]$ for each location l. All possible transition functions T have an associated global distribution $P_T = \prod_{l \in \Sigma_p} P_T[l]$. For brevity we denote the set of all global transition functions with P_T.

Example PHCAs have been shown in previous sections, for example figure 3.6. Given a PHCA model M_{PHCA} of a technical system, the behaviors of the system over time are estimated by generating sequences of *location markings* $\theta = (m^{t_0}, m^{t_1}, \ldots, m^{t_N})$. We call these sequences *trajectories* and denote by $\mathit{St}(M_{\text{PHCA}})$ the set of all trajectories. Note that in the context of PHCA in the following sections we will abbreviate the notation

t_i, t_{i-1}, t_{i+1} to $t-1, t, t+1$, since time points will be given as natural numbers. For clarity in complex formulas we will denote the set of all trajectories with Θ.

The set $St(M_{\text{PHCA}})$ of all trajectories of a model M_{PHCA} is defined through the PHCA semantic. The PHCA semantic is defined in terms of a probabilistic function STEP [162]. For a given marking m, it probabilistically chooses a transition for each of the locations in m and executes these transitions to yield a marking for the next time step. It executes these transitions, if their guard constraints are entailed, such that the next marking is consistent with the hierarchy of the given PHCA.

Iteratively calling the function STEP generates a single trajectory. To obtain all possible trajectories of a PHCA, we define a function UNROLL, shown as algorithm 4.1, which calls the deterministic version of STEP for all possible transition functions T of the given PHCA. The deterministic STEP function doesn't probabilistically choose transitions but uses a fixed transition function [162]. We provide this transition function as additional parameter. Moreover, we assume a trivial modification such that in case no commands are given, STEP creates follow-up markings for all possible commands.

Together with a sequence of observations $\mathbf{o}^{0:t}$ $(t \leq N)$ a model M_{PHCA} defines a joint distribution $Pr(\theta, \mathbf{O}^{0:t} = \mathbf{o}^{0:t})$ over the trajectories in $St(M_{\text{PHCA}})$:

$$Pr(\theta, \mathbf{O}^{0:t} = \mathbf{o}^{0:t}) = P_{\Xi}(m^0) \prod_{u \in \{0..t\}} Pr(\mathbf{O}^u = \mathbf{o}^u \mid m^u) \prod_{\tau \in \mathcal{T}[\theta]} Pr(\tau) \tag{4.1}$$

where $\mathcal{T}[\theta]$ is the multiset of all transitions as implied by θ, in which a transition τ from location l_i to l_j may occur multiple times; the transition probability is computed as $Pr(\tau) = P_T[l_i](\tau)$. Equation 4.1 reflects that PHCA compactly encode hidden Markov models [162]. It is an adapted version of equation 1 in [129]. The observation model $Pr(\mathbf{O}^u \mid m^u)$ is given as uniform distributions over those assignments to Π that are allowed by behavior constraints.

For a given trajectory θ we refer to its markings with the implicitly defined indexing function, i.e. $\theta(t_0), \ldots, \theta(t_N)$. In correspondence to [129], we also call a location marking a *PHCA state*. The set of markings for a specific time t is denoted $St^t(M_{\text{PHCA}}) = \{\theta(t) \mid \theta \in St(M_{\text{PHCA}})\}$.

Figure 4.1 illustrates PHCA markings and their evolution. We look at a single example transition within the PHCA composite location modeling the machining station in our example scenario from section 2.1. Note that not only the primitive locations need to be marked, but also the composite locations which contain them.

Algorithm 4.1 PHCA semantic that unrolls all possible trajectories for a given PHCA and given commands. For each time step, it generates all follow-up markings for a given trajectory that are consistent with the hierarchy of the PHCA and the given commands (line 6).

```
1:  function UNROLL($M_{\text{PHCA}}$, $\mathbf{c}^{t_0:t_N}$)
2:      $S \leftarrow \{(m^{t_0}) \mid m^{t_0}$ is an initial marking of $M_{\text{PHCA}}\}$
3:      for $i = 1 \,..\, n$ do
4:          $S' \leftarrow \emptyset$
5:          for $(m^{t_0}, \ldots, m^{t_{i-1}}) \in S$ do
6:              $Q \leftarrow \{\text{STEP}(m^{t_{i-1}}, M_{\text{PHCA}}, \mathbf{c}^{t_i}, T) \mid T \in P_T\}$
7:              $S' \leftarrow \{(m^{t_0}, \ldots, m^{t_{i-1}}, m^{t_i}) \mid m^{t_i} \in Q\}$
8:          end for
9:          $S \leftarrow S'$
10:     end for
11:     return $S$
12: end function
```

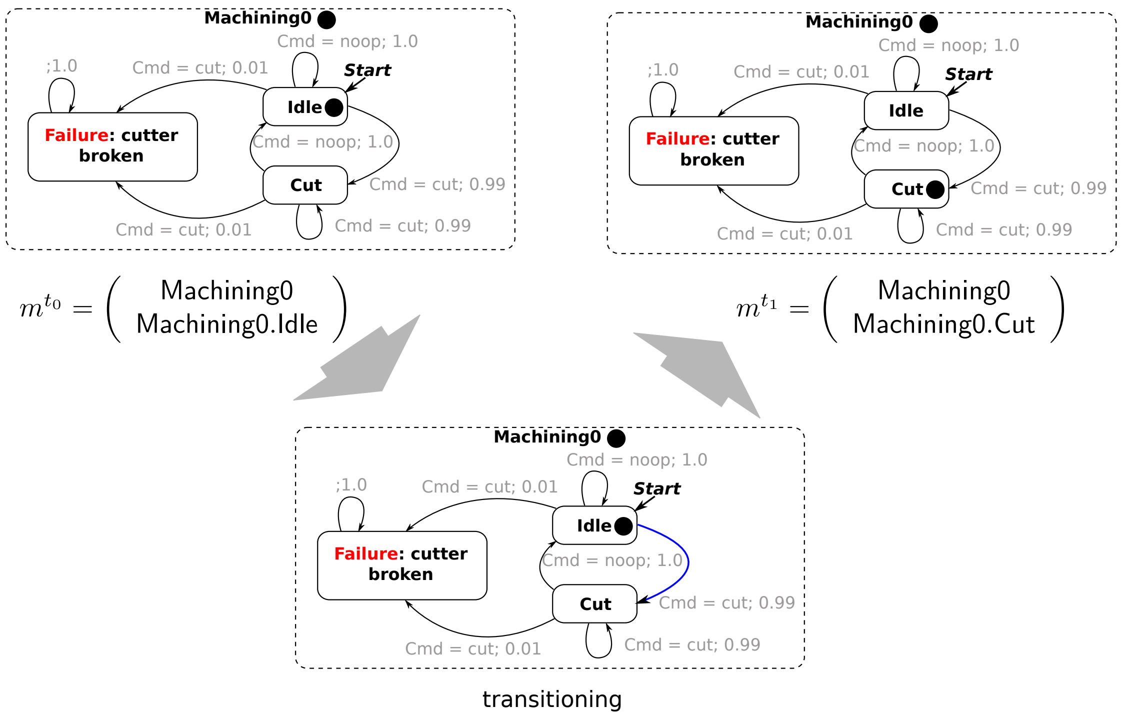

Machining0
Cmd = noop; 1.0
;1.0
Cmd = cut; 0.01
Start
Idle
Failure: cutter broken
Cmd = noop; 1.0
Cut
Cmd = cut; 0.99
Cmd = cut; 0.01
Cmd = cut; 0.99
$m^{t_0} = \begin{pmatrix} \text{Machining0} \\ \text{Machining0.Idle} \end{pmatrix}$
$m^{t_1} = \begin{pmatrix} \text{Machining0} \\ \text{Machining0.Cut} \end{pmatrix}$
transitioning

Figure 4.1.: Illustration of a PHCA location marking and its evolution via transitions.

```
NoProduct , NoComponent , 0: MAZE0 - WORKER = OK
NoProduct , NoComponent , 0: MAZE1 - WORKER = OK
NoProduct , NoComponent , 0: ROBOT0 - WORKER = OK
NoProduct , NoComponent , 0: ASSEMBLY - LINK - HOLES = OK
NoProduct , NoComponent , 0: ASSEMBLY - CMD = NOCOMMAND
Maze0 , Machining0 ,       0: MACHINING0 - CMD = MILL
Maze1 , Machining1 ,       0: MACHINING1 - CMD = MILL

NoProduct , NoComponent , 1: ROBOT0 - WORKER = OK
NoProduct , NoComponent , 1: ASSEMBLY - LINK - HOLES = OK
NoProduct , NoComponent , 1: MACHINING0 - CMD = NOCOMMAND
Maze1 , Machining1 ,       1: MACHINING1 - CMD = MILL
Maze0 , Assembly ,         1: ASSEMBLY - CMD = ASSEMBLE - COVER

NoProduct , NoComponent , 2: ROBOT0 - WORKER = OK
NoProduct , NoComponent , 2: MACHINING0 - CMD = NOCOMMAND
Maze1 , Machining1 ,       2: MACHINING1 - CMD = MILL
Maze0 , Assembly ,         2: ASSEMBLY - CMD = ASSEMBLE - PINS

NoProduct , NoComponent , 3: ROBOT0 - WORKER = OK
Robot0 , Machining0 ,      3: MACHINING0 - CMD = MILL
Maze1 , Machining1 ,       3: MACHINING1 - CMD = MILL
```

Listing 4.1: Encoded operation sequence used for the example scenario in section 2.1.

4.1.2. Operation Sequences

Pre-planned operations are typically given as a sequence $\mathcal{P}$ of operation steps. Steps can be anything from simple (sets of) commands to more complicated operations, such as scheduled allocations of machines, products and actions to perform. For example, listing 4.1 shows the encoded operation steps for the first three time points used in the example scenario in section 2.1.

In plan assessment scenarios it is likely that some sort of flow is important, for example, the flow of products through a factory plant. In abstract terms this means components, for example products and stations, are linked at one time and independent at another. To reflect this we adopted sequences of the form $\langle(p, c, t, a)\rangle_j$ for plans $\mathcal{P}$, where a tuple (p, c, t, a) not only encodes an action a to be performed at time t but also two entities, or components, of the model to be linked at t. We call a single tuple *component link* for short.

We consider scenarios where $\mathcal{P}$ is synthesized automatically (using a planning component), and later executed by the AI controller of the system. This is captured by an *execution adaptation function* $\mathcal{E}_{\mathcal{P}}$, which integrates information given in $\mathcal{P}$ with the PHCA model. This allows to reason about things like the flow of products through a plant.

In detail, $\mathcal{E}_\mathcal{P}$ takes as input a PHCA model M^N_PHCA unfolded for N time steps (reproducing the PHCA's components for each time step t, yielding Σ^t, Π^t, $\mathcal{T}^t$ and $\mathcal{C}^t$). It sets all command variables as given by $\mathcal{P}$, removing the transitions whose guards become unsatisfiable as a result, and modifies the constraints $\mathcal{C}^t$ (for example to encode a connection between a product and a station). The output is a model $M^\mathcal{P}_\text{PHCA} = \mathcal{E}_\mathcal{P}(M^N_\text{PHCA})$ that no longer contains the command variables *Cmd*. The semantic of unfolded models $M^\mathcal{P}_\text{PHCA}$ is given by a slightly modified version of UNROLL, which in each time step considers the linking constraints, which might exclude certain trajectories.

Note that since $\mathcal{E}_\mathcal{P}$ only introduces dependencies within time steps it does not affect a PHCA's Markov property, that is that its current state depends only on its previous state.

4.1.3. Goals

A goal G_i is a tuple (l, t) encoding that a location l should be marked at time t. It induces the set $\mathcal{G}_i = \{\theta \in St(M_\text{PHCA}) \,|\, \theta(t) = m^t_l\}$ of all goal-achieving trajectories that lead to a marking m^t_l that contains location l at time t. Examples are $(\mathsf{Maze0.Ok}, 4)$, $(\mathsf{Maze1.Ok}, 9)$ and $(\mathsf{Robot0.Ok}, 7)$, the goals for the three products in our example scenario from section 2.1. With the notation loc.subloc we refer to sub locations of a PHCA locations or models. Note that the expected finishing time for the products is one step after the last scheduled operation, as the actual execution of a command given in the operation sequence happens in the subsequent time step.

4.1.4. Problem Definition

Problem 7. Let M_PHCA be a PHCA model, $\mathcal{P}$ a sequence of N operation steps with execution adaptation function $\mathcal{E}_\mathcal{P}$ and $\{G_i\}$ a set of goals. Let $M^\mathcal{P}_\text{PHCA} = \mathcal{E}_\mathcal{P}(M^N_\text{PHCA})$. The **plan assessment problem** is, given observations $\mathbf{o}^{0:t}$, to compute the most probable diagnosis as trajectory in $St(M^\mathcal{P}_\text{PHCA})$ as well as $Pr(\mathcal{G}_i \,|\, \mathbf{o}^{0:t})$ for each i.

Note that, strictly speaking, the success probability also depends on the transition probabilities in model M_PHCA. However, these are so-called hyper-parameters, which in general define the shape of a probability distribution. It is common practice to leave these out as conditioning variables.

4.1.5. Discussion of Plan Assessment Complexity

The major input parameters for the plan assessment problem are the PHCA size and the number of time steps to be considered. The PHCA size could be expressed as the number of locations and transitions between them. A somewhat coarser, but more practical measure is the number of components to be modeled and included as sub-automata in a PHCA model. The number of locations and transitions scales linearly with them.

Despite the fact that providing a plan reduces the number of possible trajectories, and therefore the problem size, solving the plan assessment problem in full generality may still be NP-hard.

First of, with model-based diagnosis and probabilistic reasoning the plan assessment problem is derived from two NP-hard problems. Model-based diagnosis has been recast as a constraint optimization problem in [129], which is NP-hard in general [51]. The probabilistic reasoning part concerns computing the success probability $Pr(\mathcal{G}_i \mid \mathbf{o}^{0:t})$, which corresponds to computing marginals in Bayesian networks (see section 3.5.1). This problem has been shown to be #P-complete [138], which in our case means it is at least NP-hard, but probably much harder.

Second, we don't believe that adding information in form of a plan eliminates the exponential dependency that drives the complexity of these two problems. Resource usage grows exponentially with the number of variables defined for these problems. The number of variables in turn scales linearly with the number of time steps and PHCA locations. Adding information means to eliminate variables by assigning values to them, which reduces the problem size. Polynomial complexity could indeed be achieved if variables are eliminated such that all cyclic dependencies among the remaining variables are broken, resulting in a tree of dependencies (as opposed to a graph, in general) [100]. However, We don't believe that *all* possible plans result in this specific elimination scheme for *all* possible PHCA models.

To illustrate, let us reexamine the example from the introduction in section 1.2. Consider a plant with 2 stations that has to manufacture 3 products. The 2 stations are modeled with composite locations with 5 locations and 12 transitions each, the products with 2 locations and 3 transitions each. The naive automaton product of these 5 composite locations has $2 \times 2 \times 2 \times 5 \times 5 = 200$ states and $3 \times 3 \times 3 \times 12 \times 12 = 3888$ transitions, giving $\frac{3888}{200} \approx 19$ possible transitions per state. In general, we would have $s_p^{n_p} \times s_c^{n_c}$ states and $t_p^{n_p} \times t_c^{n_c}$ transitions, where n_p, n_c are the variable number of products and stations, and s_p, s_c, t_p, t_c fixed upper limits on the number of locations and transitions used to model different products and stations, respectively. Putting it all together, we get $\frac{t_p^{n_p} \times t_c^{n_c}}{s_p^{n_p} \times s_c^{n_c}}$ transitions per state, finally yielding $\left(\frac{t_p^{n_p} \times t_c^{n_c}}{s_p^{n_p} \times s_c^{n_c}}\right)^N$ possible behaviors for N time steps. We can clearly see an exponential dependence on the number of products and stations as well as the number of time steps.

An option to remedy this is to exploit the fact that PHCAs are compact encodings of hidden Markov models, employing off-the-shelf implementations of the Viterbi (to compute the most probable trajectory) and the filtering algorithm (to compute the success probability) for these models (both very well explained in [141, Chapter 15]). However, this only eliminates the exponential dependency in time, and has other disadvantages, which we explain later in chapter 5.

4.2. Plan Assessment in Autonomous Manufacturing

The previous section defined the plan assessment problem in general. Now we look at the particularly interesting domain of manufacturing. In manufacturing, systems historically have been very deterministic, operating in strictly controlled environments. Modern requirements such as mass-customized products, however, make them more and more complex, leading to situations where the added complexity of some autonomous control in such systems can be cheaper than trying to remove all uncertainties by design. The result are systems with largely deterministic behavior with some remaining uncertainties: the "breeding ground" for the plan assessment problem.

The PHCA model of a manufacturing system consists of composite locations for stations and for products. The plan $\mathcal{P}$ is a schedule, that is the sequence of tuples $\langle(p, c, t, a)\rangle_j$ defines an action a to perform on a product p at time t on a station or component c. As we said we call these tuples component links. However, in context of manufacturing we denote them more specifically as *product-component links*, since mostly they will specify a link between a product and a factory component, for example a machining station. The elements p and c are names that identify the composite locations modeling product p and station(component) c, respectively. a is an assignment of values to command variables and t an integer representing a discrete time point. In this work we consider planning to be a generalization of scheduling, section 4.4.3 goes into some detail about this. As a consequence, we will generally use the terms "plan" and "planning" when we refer to plans/schedules or planning/scheduling. In the following, we will examine models for products relevant to this work.

4.2.1. Models for Products in Automation and Planning

Before we describe the kind of product models used in this work, we need to look at relevant formal descriptions of products in other fields like enterprise resource planning or automation, and which kind of product models could be used in a planner of an autonomous manufacturing system.

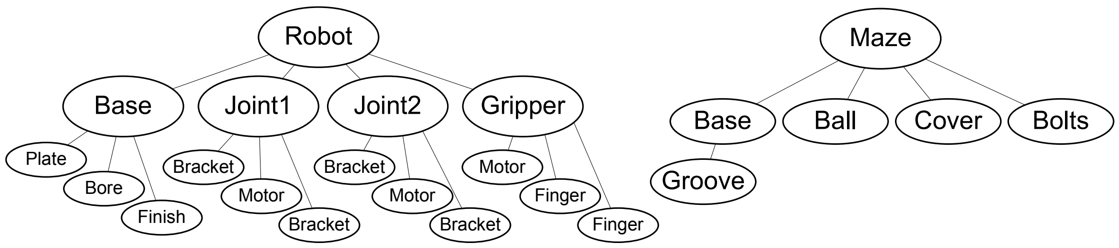


Figure 4.2.: Illustration of how products could be modeled for PDDL planning, following existing approaches to formal descriptions such as the bill of materials. A tree structure describes a product by encoding product parts with nodes and part-of relations with edges. © Thomas Rühr.

A formal or semi-formal description of products used in enterprise resource planning and related fields is the bill of materials (BOM) [28]. It is a structured representation of product parts and how they are to be put together. Typically, the structure is some sort of tree or a directed acyclic graph, often annotated with further information such as part identifiers. The bill of material can come in different flavors, depending on who is using it (engineers, sales persons, etc.). In [156] automated planning of steps for assembly is addressed. This work also uses directed acyclic graphs to represent products. More work on product models can be found in [167, 82] for design and development, in [123] for computer aided process planning (CAPP) and in [61] for issues of mass-customization.

Trees or directed acyclic graphs seem convenient ways of structuring a product model. A plausible way of modeling using the well-known planning domain description language (PDDL) [125] in the domain of manufacturing [57], could be a tree-structure (V, E, C), with vertices V, edges E and a set of first-order logic formulas C. See figure 4.2 for examples (images taken from [121], curtesy of Thomas Rühr). The vertices V represent basic product parts, features and compositions thereof. The edges E are "part-of" relations, while the formulas C encode constraints on ordering of subsets of edges. A product model (V, E, C) could be seen as a bill of materials encoded in PDDL, along with descriptions of stations and products. Given an abstract product description like this, a PDDL planner's task would then be to find a good sequence of actions that realize the "part-of" relations.

4.2.2. PHCA Product Models

In this work we represent a product's production process with a PHCA composite location. The representation follows the basic assumption that a *product is going to be ok as long as no station involved in its production fails.* In other words, the process, and its resulting product, is ok as long as no station or component of the factory plant is faulty while working the product. We use a generic product model, comprised of two locations representing the product being ok or faulty, respectively. The model stays in the ok-location as long as every station working the product is fine, and transitions to the faulty-location if the working station itself was or became faulty. For example, with a small probability the assembly station may become misaligned at any time point. If it does, any product worked by it will become faulty.

An essential part in product modeling is the flow of the products through the plant, represented as product-component links in the schedule $\langle (p, c, t, a) \rangle_j$. The flow influences the model by entangling transitions on stations with transitions of the product model, for the time the product is worked. If the product moves to a different station, the entanglement is adapted accordingly. This entanglement is realized by the execution adaptation function $\mathcal{E}_{\mathcal{P}}$, which applies the necessary changes to the unfolded PHCA model M^N_{PHCA}. A detailed treatment of this process will be given in the context of the solution approaches for plan assessment in chapter 5.

An important question is how these product models are to be incorporated into the PHCA plant model. A mostly automated chain from the design of a product to its autonomous production is hard to implement without some sort of automatic model composition. This composition needs to automatically combine PHCA models given from engineering with product PHCA models derived from given formal product descriptions, such as bills of materials or CAD models. Related work already addresses problems such as deriving manufacturing steps from CAD models [57]; ideas from this work could surely be transferred to a derivation of PHCA product models. Developing such an automatic model composition is, however, a topic in its own right and therefore beyond the scope of this work. Here, we use generic product models as shown in figure 4.3. With these, a manual composition is comparably easy.

Another important point are goals and sub-goals. We consider as goal for producing a product that it should be finished successfully after all scheduled operations. Considering our product model, that means that it should be in the ok-location at that time point. This is easily mapped to a formal goal (Ok, t). t is the time point after the last scheduled operation, which can be read from $\mathcal{P}$. An assumption we make here is that scheduled operations finish on time.

A sub-goal could be finishing sub assemblies for parts of a product. In this work, we do not explicitly regard sub-productions, and goals are assumed to be associated with full products (in the context of manufacturing). However, sub-goals could be regarded by adding corresponding locations to a product composite location, as shown in figure 4.4. Some additional information in plan $\mathcal{P}$ would be required that explicitly hints at

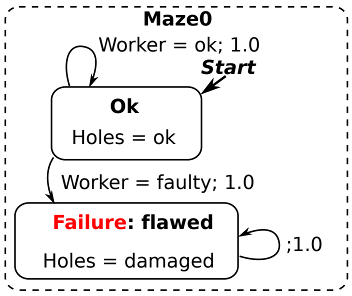


Figure 4.3.: Product model as composite location.

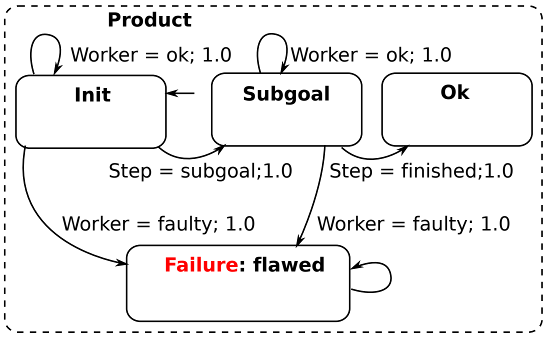


Figure 4.4.: Illustration of how sub-goals for a product, e.g. for sub-assemblies, could be integrated in PHCA product models. An additional requirement for the plan $\mathcal{P}$ is then to provide explicit information when sub-goals should nominally be reached.

when sub-goals should be achieved. This should be an easy task for a planner capable of producing plans $\mathcal{P}$, since it would be precisely its task to figure out how to reach these sub-goals.

4.3. Plan Assessment as a Provider of Decision Criterions

The purpose of a plan assessment component as part of, for example, the AI controller of an autonomous manufacturing plant, is to provide information as basis for automatic decisions. Here is the sketch of a simple automatic decision procedure utilizing the information gained from plan assessment:

1. For each goal G_i:
   a) Compare success probability $Pr(\mathcal{G}_i \,|\, \mathbf{o}^{0:t})$ against thresholds ω_{fail} and ω_{success}.
   b) Proceed for goal G_i if above ω_{success}.
   c) If below ω_{fail}:

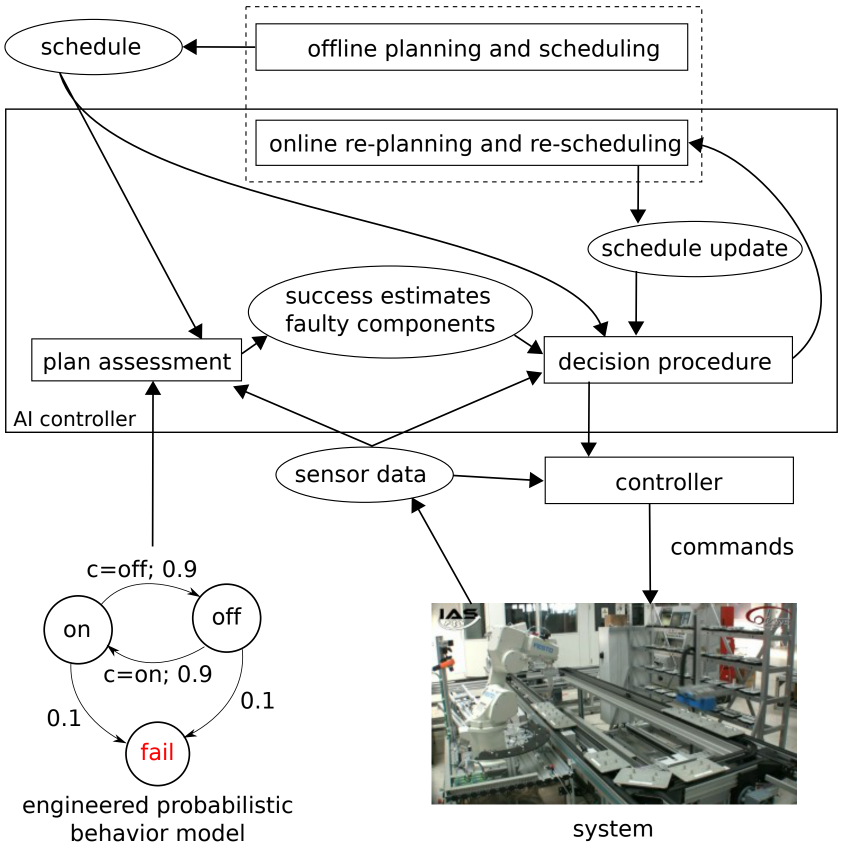


Figure 4.5.: Possible architecture of an AI controller using plan assessment (detailed version of illustration in figure 1.1).

i. Product already finished: Vote for stopping.

ii. Product not yet finished: Request re-planning.

d) If in between ω_{fail} and ω_{success}: Vote for information gathering actions with focus on goal G_i.

2. Combine individual decisions:

   a) Count and weigh (e.g. using expected revenue or loss) votes for stopping and information gathering, then respond appropriately, for example by calling a technician or by re-planning including diagnosis goals [106].

   b) Weigh (e.g. via revenue or loss) re-planning against discarding the goals and if necessary, initiate re-planning focussing on the requesting goals and avoiding faulty components.

3. Repeat.

Together with figure 4.5 the procedure illustrates how plan assessment connects to the existing AI techniques of information gathering and planning.

Step 1 illustrates how the information provided by plan assessment realizes a decision criterion, in the form of success probabilities compared against ω_{fail} and ω_{success}, which can guide decisions in an exploration vs. exploitation manner. Step 2(a) implements in particular the exploration in form of pervasive diagnosis [106] to gain more information on faulty stations. Step 2(b) illustrates how re-planning [25] or re-scheduling [109, 157] can be employed to guide production of jeopardized products away from faulty components.

Furthermore, the simple procedure above hints at a useful extension of plan assessment towards decision theory. In addition to probabilities it also regards the *utility* of the goals and actions, in this case derived from expected revenue or loss. A natural extension of plan assessment as introduced in this work, embeds it in a decision theory framework adopted for AI decision making [140, chapter 16, p. 584–612]. It could use an *expected success utility*, computed as sum over products of the success and failure probabilities with respective utility values for success and failure. Then, the expected success utility could be compared against utility thresholds. A different alternative might be to integrate plan assessment into the framework of partially observable Markov decision processes (POMDP) [142], which deals with making complex decisions under uncertainty. In this work, we focus on computing success probabilities and most probable behaviors, and leave the development of decision-theoretic extensions for future work.

4.4. Plan Assessment in Context of Related Problems

The plan assessment problem lies between model-based diagnosis and probabilistic reasoning. Connections between these two areas have been pointed out and exploited for a long time, for example in [100, 103, 1, 106, 50]. This work follows this tradition as it derives plan assessment from a variant of model-based diagnosis, which we explain hereafter. Then, we detail the relations of plan assessment to state estimation and to planning and scheduling.

4.4.1. Plan Assessment as Extension of Model-Based Diagnosis

First, we recap the classical model-based diagnosis problem [77]. Based on a logical description of a system and "unexpected" observations, the task is to identify faulty components that explain these observations. Formally, the system model is given as a tuple $(SD, COMPS, OBS)$, where SD and OBS are sets of logical (usually propositional) formulas describing the system behavior and available observations, respectively, and $COMPS$ the set of components. The "unexpected" observations are formally expressed as SD and OBS being inconsistent, i.e. $SD \cup OBS \vdash \bot$. The task is thus to restore consistency with a minimal diagnosis $\mathcal{D}(\triangle, COMPS - \triangle)$:

$$SD \cup OBS \cup \mathcal{D}(\triangle, COMPS - \triangle) \nvdash \bot$$

$\mathcal{D}(\triangle, COMPS - \triangle)$ is a logical formula over literals $AB(c)$ if $c \in \triangle$ and $\neg AB(c)$ if $c \notin \triangle$. $\triangle$ is the subset of components assumed to be faulty or abnormal, and $AB()$ the predicate encoding abnormality for a component. A diagnosis is required to be of minimal cardinality, i.e. $\triangle$ must be as small as possible (*all* components being faulty is always a valid diagnosis, but unfortunately not informative at all). In practice, this requirement often turns out to be too weak. *Probability-maximal diagnosis*, which sorts diagnoses according to how probable they are, improves on that. We will explain it in a moment.

The system description SD is constructed using standardized model components, where each component describes the behavior of a standard system component, e.g., an electrical motor, an adder for digital circuits etc. The idea is to construct models from first principles, usually meaning physical laws such as Ohm's law. Theoretically, all information necessary to diagnose any system, apart from observations, is contained in the library of these standard components. Drawing a connection to plan assessment, we see that SD corresponds to M_{PHCA}, although we do not presume that M_{PHCA} is constructed from first principles. We assume that we receive M_{PHCA} from engineers, who might have used such a library when developing the model. OBS obviously corresponds to $\mathbf{o}^{0:t}$, however might be represented differently. The set $COMPS$ is not explicitly represented for plan assessment.

As mentioned, searching for minimal cardinality diagnoses can still produce too many diagnoses to handle. *Probability-maximal diagnosis* [49, 146] addresses this problem by sorting diagnoses according to how probable they are. Also, it can use probabilities as a heuristic to efficiently search for diagnoses [49]. The probability of a diagnosis is the product of individual fault mode probabilities, which are assumed to be independent:

$$\prod_{c \in COMPS} f(c) \text{ , with } f(c) = \begin{cases} p(c) & \text{if } c \in \triangle \\ 1 - p(c) & \text{if } c \notin \triangle \end{cases}$$

The goal is to find the diagnosis with maximal probability.

It is often desirable to consider a system's evolution over time. For example, the effects or symptoms of a fault might not be observable right away. To diagnose faults facing such delayed symptoms requires reasoning over a fixed time horizon [108]. Another advantage is that *intermittent* faults may be regarded, i.e. faults that are not persistent.

To achieve this, model-based diagnosis can be further extended to reason over discrete time steps. Instead of a static mode assignment, a diagnosis is a sequence of mode assignments for all time steps in a given time window of length N. The model SD then contains descriptions of component states and transitions between them. Faults are then modeled as the actual failing of a component between two (or more) time steps. This time-step based diagnosis is typically solved by searching for probability-maximal sequences of mode assignments [108, 129, 3], the *trajectories*.

From a probabilistic reasoning point of view, time-step based diagnosis for $k = 1$ corresponds to the standard problem of finding most probable a posteriori hypotheses (MAP). In case SD is a hidden Markov model (HMM), the most probable diagnosis may be computed using existing implementations of the Viterbi algorithm [21, Chapter 13, p. 629 ff.].

Model-based diagnosis has a global view on the behavior of a system. Given (suspicious) observations it asks, what is broken in my system? One answer could be that most likely the assembly station of a plant is miscalibrated. We can also take a local view point by asking: how likely is it that the assembly station is miscalibrated, given the current observations? Rather than finding an explanation for observations, we are now interested in the influence of observed effects on a specific, local part of the system (the miscalibration fault of the assembly station). Another example are plan goals: How likely

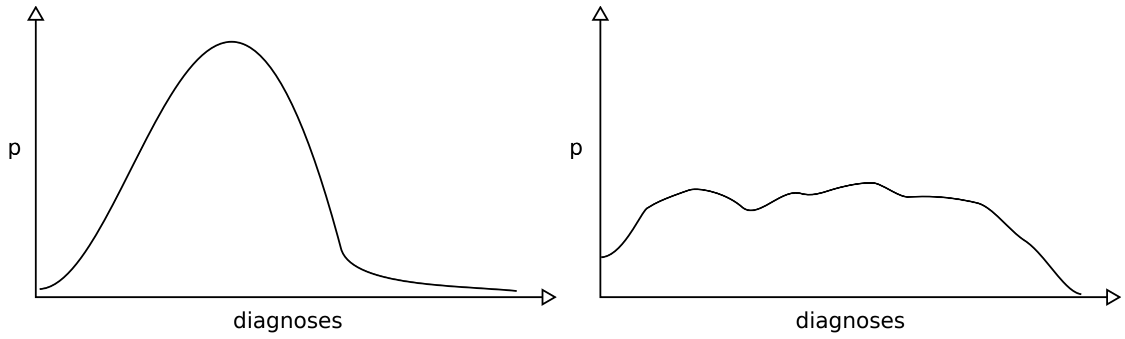


Figure 4.6.: A distribution with a clear maximum probability diagnosis (left) and one that is ambiguous (right), i.e. where many other diagnoses with probabilities close to the maximum exist.

is it that my goal to produce a product is achieved? Here we ask for the influence of observed effects on the (potentially future) manufacturing and assembly process, *locally* for that specific product.

Local queries for probabilities like that can be framed as instances of a core probabilistic reasoning problem: the computation of marginals [135, chapter 4.5, p. 223], which was explained in section 3.5.1. Assume as given a probabilistic model that defines, among other things, a set of random variables, and a set of assignments to designated query variables. These query variables might encode the system parts of interest, for example. The problem is then to compute the effect of the observations, propagated through all other variables, on the probability of these assignments. This is called *marginalizing,* and the query variables are called *marginals.*

The plan assessment problem extends the described model-based diagnosis variant based on trajectories with this local view. In section 2.1 we have seen how both, the global and the local view, can support AI decisions. But putting the two views together can have additional benefits. For example, knowing the most likely system behavior can explain why a certain product has currently low success chances. Vice versa, if a component c was identified to be faulty as part of the most probable diagnosis, asking for the probability of c being faulty (given current observations) can clarify whether this diagnosis is solid or not. It could be not solid if many other diagnoses with lower but close probability values existed (see figure 4.6). This would be an indicator for weak observations and as a reaction one could trigger active information gathering actions.

Plan assessment can be seen as a combination of the probabilistic reasoning problems of computing the most probable a posteriori hypothesis and computing the marginals, but it takes a different point of view. Usually observations and actions/commands are both treated indifferently as evidence in probabilistic reasoning. Plan assessment separates the planned, and thus well known, operations for online execution (such as commands) from the observations that become available only gradually during execution. The planned operations are used to modify the model, focussing it on the known execution of the system. This modification can be done online, but also be taken offline.

4.4.2. Plan Assessment and State Estimation

In practical applications it can quickly become intractable to use a static time window that covers the complete horizon of interest. Typically, a receding horizon approach is used that moves a fixed-size time window along the time line [108, 129]. These and other approaches like [87] are often seen as a tracking of diagnostic hypotheses. A related, classical form of tracking is *belief state update* as it is known from reasoning with partially observable Markov decision processes (POMDP)[142, chapter 17, p. 625 ff]. The idea is to maintain a probability distribution over possible states of a system and/or its environment. In POMDP reasoning, this distribution is the foundation for decision making under uncertainty: based on the current belief state, find the action that optimizes a given reward function. More generally, the POMDP problem is to find a *policy*, i.e. a function mapping belief states to actions, which is optimal with regard to the reward function (i.e. always gives the optimal action for the given belief state). Typically, belief state update is implemented as an iterative process that computes the current distribution from the previous one and from known observations and actions. A standard algorithm available in off-the-shelf implementations is forward filtering for hidden Markov models [140, chapter 15].

In this work, we use probabilistic hierarchical constraint automata (PHCA) to model systems. Therefore, a particularly relevant POMDP implementation is the PHCA executive for embedded systems named Titan [163]. It estimates hidden system states as distribution over PHCA states and then, based on the most probable state, computes least cost action sequences to reach short-term goals. These goals are constantly being derived from the control program that Titan is meant to execute. Plan assessment is different in that it deals with more long-term, static goals and a plan being executed to achieve them, constantly assessing it with respect to these goals.

More generally, POMDP addresses problems that due to high uncertainty require constant online re-planning. In contrast, plan assessment focuses on situations with less uncertainty that warrant long term advance planning by specialized planning components, yet occasionally require dynamic reaction to unplanned events. Furthermore, while it is interesting to investigate adaptations of POMDP algorithms (as described in, e.g., [26]) for the plan assessment problem, this is not in line with the theme of this work to rely on externally implemented, generic algorithms.

The iterative belief-state update approach can be used to realize a receding horizon approach for plan assessment. Chapter 5 describes how plan assessment can be solved by generating trajectories and summing over them. Just like for diagnosis, this becomes quickly intractable for realistically sized time horizons. In this work, we describe an approach that combines searching k most probable fixed-length trajectories with a variant of belief state update. The details are given in section 7.1, here we only sketch the idea. The key question is: how do we incorporate the information about past states that lie outside the time window we are using? The answer is: By computing a start state for the time window, or rather, a *start state distribution*, from the trajectories computed in the previous time step. This is a variant of belief state update, with the subtle but important difference that the new distribution is not, as in belief state update, computed from a prior distribution over states, but over trajectories.

Similar to belief state update is *state observation*, a well-known standard problem in control theory. Given inputs and outputs of some physical system, the task is to continuously refine the internal model state of a *state observer* such that it corresponds with the internal state of the physical system. In other words, the model state is an estimate of the typically unobservable internal state of the physical system, which is continuously updated. The state observer is to state observation what SD is to model-based diagnosis. Standard solutions to this problem iteratively generate new estimates for the current time point from current inputs and outputs and the previous model state. A standard solution for continuous models, e.g. with real-valued state variables and discrete time, is the well known Kalman filter [98]. Implementations of this solution can be easily found on the internet[1]. For purely discrete models again HMM forward filtering can be applied for state observation. When dealing with mixed discrete/continuous models an established class of algorithms is *particle filtering* [55, 105]. Section 7.3.2 covers the reasoning problems such as state observation for hybrid models in greater detail. A good text book reference for state observation with a special focus on diagnosis and fault-tolerant control, and in particular for hybrid models, is [23]. Chapter 3 explains the fundamental modeling of systems or observers, chapters 8 [23, p. 392 ff.] and 9 [23, p. 451] cover state observation for discrete and hybrid continuous/discrete models, respectively.

[1]See, e.g., `http://www.cs.ubc.ca/~murphyk/Software/Kalman/kalman.html` (04.03.2011). The package has however not been tested by the author.

Problems such as belief state update and state observation are typically subsumed under the more general term *state estimation*. However, to the author's knowledge, problems classified as state estimation (and in particular state observation) are often "closer to the hardware" than belief state update or model-based diagnosis. For example, to discover a blunt cutter, state estimation might be used to estimate the current hidden state of the material the cutter is made of (is it weak, about to break?) from the observable cutter temperature and its rotation speed. Model-based diagnosis would typically take a more abstract view of the complete machining station, modeling a blunt or broken cutter as states in a finite state machine framework (such as PHCA).

4.4.3. Plan Assessment, Planning and Scheduling

Plan assessment is most useful when it works in concert with strong planning and/or scheduling components. We now briefly describe the problems of planning and scheduling, how they interact and how they are related to plan assessment. We also support our assumption that re-planning and re-scheduling are, in fact, possible by highlighting some related work in this area.

Planning and Scheduling

Planning is one of the classical AI problems. Following the text book explanation in [143, chapter 11], the planning problem is to find a sequence of valid actions that drive a system from a given initial state to a goal state, which is typically part of a given set of goal states. The sequence of actions is called a *plan*. In principle, general problem solving algorithms such as the search procedures described earlier in section 3.3 could be applied to planning. However, this typically doesn't scale to real-world planning problems. Research in artificial intelligence has been addressing this problem by developing frameworks and modeling languages that are specific with respect to planning, yet try to be general enough such that many planning problems can be addressed. The first such framework was the Stanford Research Institute Planning System, or STRIPS [64]. A more recent, widely used framework is the planning domain description language, PDDL [125]. It uses first-order logic formulas to describe a planning problem and its domain using predefined predicates and actions. Specifically, a PDDL planning problem is a tuple (E, P, A), where E is a set of entities (such as plant stations, components or products), P is a set of first-order logic predicates and A a set of actions. First-order logic is used in particular to describe pre- and postconditions for actions. An example of a planning system designed for planning and executing tasks of household robots is the reactive plan language [13], and its successor, the CRAM plan language [17] (where CRAM stands for cognitive robot abstract machine).

The classical scheduling problem is job shop scheduling [2]: A number of n given tasks/jobs/operations is to be assigned to m resources, while optimizing a function of the accumulated time of achieving all scheduled tasks. Typically, the length of the schedule in time, the makespan, is optimized. For example, to produce products, their single production steps must be assigned to stations in a factory plant. Obviously, the task durations mustn't overlap in the resulting schedule. Additionally, ordering constraints on the tasks might apply, e.g. parts of the maze must be cut before assembly of the complete product.

The relation between planning and scheduling becomes most obvious with this latter point of ordering constraints on tasks. It is precisely the job of a planner to figure out these constraints, i.e. put actions into an order that achieves given goals. This hints on an architecture where planning feeds into scheduling. However, typically many orders of actions are feasible from the point of view of planning, but only a few from a scheduling viewpoint. Hence, we cannot say in general whether it's better to first plan, then schedule, or the other way round. Recent approaches that apply AI planning and scheduling to industrial tasks such as large scale printing [139] combine techniques from both worlds in an interleaved fashion.

Interaction of Planning/Scheduling with Plan Assessment

In the context of plan assessment, we adopt the view that scheduling is a special case of planning. In particular, a plan is the more general structure, simply a sequence of operation steps, while a schedule is a special plan, for example a sequence of tuples $\langle(p, c, t, a)\rangle_j$, specifying that action a is to be performed on (not yet finished) product p at station c at time t. So when speaking generally, we will simply refer to plans, planners and planning components, which however we deem capable of generating schedules.

Plan assessment connects to planning as sort of an online supervisor. We consider the planning task the more expensive task, which is likely to be done offline. A planner would take a more high-level view than a plan assessment component to be able to create plans for large time horizons. It would, for example, ignore the behavior of components such as potential faults and focus on their capabilities. In the online phase, production is supervised by plan assessment. It works with a more detailed model of the system (a PHCA), which, however, is automatically restricted by the given plan and by observations that become available during the online phase.

Plan assessment evaluates plan steps against given goals and generates information for a decision procedure that in turn can trigger local re-planning or re-scheduling if interrupting events such as faults occur. Much research focuses on these problems. For example, in [25] the authors describe a re-planning algorithm for hierarchical planning, whereas in [109, 157] local re-scheduling problems are being addressed.

5. Approaches to Plan Assessment Based on Generic Algorithms

In this chapter we investigate two approaches to solve the plan assessment problem based on a PHCA (probabilistic hierarchical constraint automaton) model. Both use off-the-shelf implementations of generic algorithms. Consequently, both translate a given PHCA into a problem format for the respective tools they employ.

The first is rooted in the area of model-based diagnosis and enumerates most probable trajectories as k best solutions of a constraint optimization problem the given PHCA is being translated to. The second, based on probabilistic reasoning, translates PHCAs to a generalized version of Bayesian nets and applies off-the-shelf solvers that either accept Bayesian nets or their generalization as input. Both approaches can be understood in terms of reasoning about possible system behaviors and their associated probabilities.

Both approaches have their merits and downsides. For the first approach, strong open-source constraint solvers are available that lend themselves to an implementation of k-best versions of their optimization algorithms. This allows to try a combined computation of most probable diagnoses and success probabilities, such that we essentially receive both bits of information simultaneously. Moreover, it has approximation built-in for success probabilities. The caveat: this intertwined computation might force a design decision, as only one of the two, diagnosis or the success probabilities, will be correct for all problem instances, the other might be incorrect in some cases. Our results show that this problem is not purely theoretical.

The second approach relies on strong probabilistic reasoning algorithms for both diagnosis and computing success probabilities. We tested the approach with a state-of-the-art probabilistic inference tool for Bayesian nets. Our translation to generalized Bayesian nets not only allows to use this tool but an even wider range of methods and tools to be applied than for classical Bayesian nets. Also, with sampling an approximation method exists for which we could find well defined stochastic error bounds for success probabilities (see section 7.2). However, so far, sampling was too slow on our translated models to be evaluated. Furthermore, it appears that with existing tools always two runs are necessary to compute both diagnoses and probabilities.

Note that for the probabilistic reasoning approach we focus on computing success probabilities. That is, we didn't implement the computation of diagnoses. However, an algorithm based on the solver we investigate does exist that can compute diagnoses, and we think it is only a matter of time until it becomes available in the same distribution.

5.1. Using Generic Algorithms to Solve Plan Assessment

One theme of this work is to exploit general methods implemented in publicly available tools. This has a number of advantages over hand-crafted algorithms. First of, one avoids reinventing the wheel in form of a special purpose solution. Also, novel general improvements are readily available. Finally, one is less prone to errors when using tried and tested implementations. When considering the plan assessment problem, which lies at the intersection of model-based diagnosis and probabilistic reasoning, we see that in both areas off-the-shelf implementations of generic algorithms are available. Therefore, we choose to translate PHCAs into a representation that these implementations can understand, rather than developing our own, PHCA-specific algorithms for plan assessment.

When choosing the right tools, we face another issue. Model representation always strongly influences problem solving efficiency. Often, Markov modeling is used such that the Markov property may be exploited later. However, compared to more expressive non-Markov modeling this can lead to bigger representations. This size vs. expressiveness trade-off is even more important when automatically generating such representations from high-level models. Originally, PHCA semantics were defined in terms of hidden Markov models (HMM) [162], which allows complex Markov modeling at the cost of potentially larger models. This is problematic for existing off-the-shelf HMM solvers: On the one hand, a naive automated flattening of hierarchical models would generate prohibitively large HMMs. Automatically decomposing PHCAs into explicit, tractable HMM representations, on the other hand, is only possible in a small number of special cases where components are not connected. Our example is not such a case: The assembly and manufacturing stations seem independent of each other; however, they are connected via product models and their planned actions. In addition, automatic compilation typically incurs an overhead compared to custom tailored translation, which further increases the size of explicit HMMs.

These issues do not rule out off-the-shelf HMM tools as potential solving backends for the plan assessment problem. However, strong tools exist that are more general than these solvers, i.e. they do not depend on the Markov property, yet can exploit it (to an extent) if present. Therefore, we choose an approach where we translate, in an offline step, M_{PHCA} problem descriptions that are expressive enough to represent structure and are sufficiently general to allow the use of off-the-shelf solvers that exploit model structure in a general fashion. We denote that translation with Υ.

Remember that we have to apply an execution adaptation function $\mathcal{E}_{\mathcal{P}}$ on a system model to account for operations given in $\mathcal{P}$. Instead of applying $\mathcal{E}_{\mathcal{P}}$ directly to M_{PHCA}, we define execution adaptation functions on the representations resulting from translation. This allows to reuse the translation for different plans $\mathcal{P}$. For easier readability we denote these functions also with $\mathcal{E}_{\mathcal{P}}$.

In the following, we will describe our two approaches, which are both based on such translations. Both roughly follow these steps:

1. Offline, a given PHCA model is translated into a problem description (X, D, C) accepted by external off-the-shelf solvers: $\Upsilon(M^N_{\text{PHCA}})$. In the process, the model gets unfolded over time, explicitly representing all N considered time steps.

2. Online[1], the translation (not the unfolded PHCA) is adapted using special execution adaptation functions $\mathcal{E}_{\mathcal{P}}$, i.e. $\mathcal{E}_{\mathcal{P}}(\Upsilon(M^N_{\text{PHCA}}))$.

3. Online, observations are added.

4. Online, success probabilities and a diagnosis are computed.

This is a good place to pinpoint this thesis' contribution to each of these steps. For step 1), it provides a novel translation of PHCA models to BLNs for the probabilistic reasoning approach. For the model-based diagnosis approach we rely on the existing soft-constraint translation introduced in [129]. For step 2), it contributes for both approaches the adaptation of the translated PHCA model for a given plan. For step 3) we provide implementations for both approaches. The implementation for the model-based diagnosis is adapted from a similar step in [129]. Finally, for step 4) the thesis contributes a novel method to compute success probabilities and a diagnosis based on the k best solutions to a constraint optimization problem. For this, we implemented k-best versions of generic search algorithms in two constraint optimizers. This facilitates the model-based diagnosis approach. For the probabilistic reasoning approach we use the BLN framework developed

[1]This is not a problem for the model-based diagnosis approach, where the effort of execution adaptation is negligible. For the probabilistic reasoning approach this can be a problem, as will be detailed in sections 5.3.3 and 6.5.

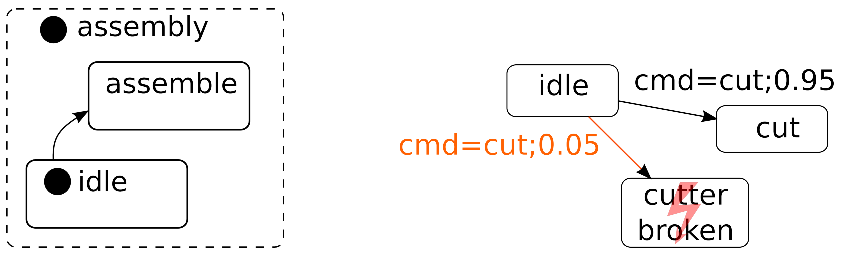


Figure 5.1.: Key translation elements: Translating markings (left) and probabilistic guarded transitions (right).

by the authors of [95] to compute success probabilities, transparently calling an external inference tool.

As the PHCA translation is a major part for both approaches, we roughly describe how it works in general, before we go into the details of either approach. The two key elements of PHCAs that need to be addressed for translation are location markings and probabilistic, guarded transitions (see figure 5.1). Both translations explicitly represent locations (primitive and composite alike) being marked or not marked at certain time points. Linked to these explicit representations are other things, such as a behavior constraint of a location being satisfied if that location is marked. Considering transitions between locations, both translations have to make sure that, for a given location and time, a) the outgoing transitions with inconsistent guards (e.g. if at that time a command is not given) are ruled out and b) among the remaining transitions exactly one is chosen probabilistically. The translations do not ensure that the probability values for transitions add up to one. This has to be addressed by the model designer, who has to define transition probabilities and guards in such a way that for all time points only those transitions are possible, as determined by guard consistency, whose probability values add up to one.

5.2. Model-Based Diagnosis Approach

In this section we introduce an approach to plan assessment that first translates a given PHCA into a constraint net and then solves a k-best constraint optimization problem (COP) for this net, i.e. problem 3 defined in section 3.3.1. The presented work is a revised and extended version of work published in [121].

As described in the beginning of section 4.1, the plan assessment problem extends the maximum probability diagnosis problem. Therefore, an approach suggests itself that builds on methods that compute the maximum probability diagnosis. Such a method is presented in [129], which recasts the maximum probability diagnosis problem as a COP. The optimal solution of this COP is straightforwardly mapped back to the maximum probability diagnosis. Our approach presented hereafter extends this method. Instead of computing only the single best solution, it generates the k best solutions and uses them to compute success probabilities and the most probable diagnoses.

Our approach maps the best solutions of the COP to the most probable trajectories and approximates $Pr(\mathcal{G}_i \mid \mathbf{o}^{0:t})$ by summing over goal-achieving trajectories among these, normalizing over all generated trajectories. We show how this approach invites a combined computation of success probabilities and most probable diagnoses based on the previously generated COP, and how that leads to a problem which forces the already mentioned design decision between potentially incorrect diagnoses or success probabilities. We also present a practical plan evaluation procedure that employs these computations and elaborate a possibility to bound the error of the k-best approximation. Before we begin describing the approach, we first explain the two k-best algorithms that we developed for it.

5.2.1. k-best Constraint Optimization with A* and Branch-and-Bound

To enable approximate approaches for plan assessment, we developed and implemented k-best versions of the well known A* and branch-and-bound algorithms for constraint optimization. The A* implementation was added to Toolbar and the branch-and-bound implementation to Toulbar2.

Algorithm 5.1 shows the k-best version of an A* algorithm. The extension to create k best solutions instead of only one is simple: Instead of terminating once a full assignment is found, search continues until k full assignments have been generated. Like A* this algorithm traverses the search tree of all partial assignments. A more general k-best A* for the search in graphs has recently been proposed in [52].

Let us look at the algorithm in more detail. The function EXPAND() generates, with partial assignment a as basis, a new set of partial assignments by instantiating the next variable in the given ordering with all possible values, and then adds the partial assignments to Q. Functions h and g are, respectively, the heuristic giving optimistic estimates for the cost of expansions yet to be made and the cost function giving the cost of the current partial assignment. Function BEST$(h+g, Q)$ returns the best assignment in Q according to the given evaluation function, which in this case is $h+g$ (point-wise added). Q is typically implemented as priority queue, for example as a heap. S is implemented as an ordered list, i.e. it contains all generated full assignments in the order added.

Note that this algorithm is different from K-best-first search [62], which computes approximately optimal solutions by expanding, at each search node, only the k best candidate solutions in Q. In contrast, both A* and k-best A* expand all candidate solutions and produce optimal solutions.

Algorithm 5.1 k-best A* algorithm for solving k-best weighted constraint satisfaction problems (WCSP).

```
function KBESTASTAR(({$X_1, \ldots, X_n$}, {$D_1, \ldots, D_n$}, $C$, $k$), $\top_{\text{ub}}$, heuristic $h$, cost
function $g$)
    $Q \leftarrow \{\bar{\varepsilon}\}$
    $S \leftarrow \emptyset$
    while $Q \neq \emptyset$ do
        $a \leftarrow$BEST($h + g$, $Q$)
        if $a$ is a full assignment then
            $S \leftarrow S \cup \{a\}$
            if $|S| = k$ then return $S$
            end if
        else
            EXPAND($a, Q, h, g$)
        end if
    end while
    return
end function
```

We developed and implemented a k-best branch-and-bound method shown as algorithm 5.2. The algorithm was inspired by an informal description in [147]. Very recent work also investigated such an algorithm [52], therein called m-BB (short for m-Branch-and-Bound).

Our algorithm doesn't deviate much from the branch-and-bound algorithm 3.2 described in section 3.3.2. The key difference is that during search, instead of the current best full assignment, we keep a list B of full assignments organized as a reverse priority queue (i.e. with the worst assignment on top).

In detail, algorithm 5.1 works as follows: Like branch-and-bound, k-best branch-and-bound keeps generating full assignments in a recursive manner as long as unexpanded nodes are left. Unlike branch-and-bound, it puts full assignments in the list B if they are better than the worst assignment already in B. Every time an assignment is added to B and it then contains more than k elements (lines 5-10), the worst is removed from B. Functions WORST(B) and POPWORST(B) return and remove the worst assignment, respectively. We use a heap data structure for B, giving us efficient implementations of these two functions. In the beginning, set B has to be filled up with consistent assignments. Therefore, as long as B contains less than k assignments, the lower bound $h(\mathbf{x}') + g(\mathbf{x}')$

is compared against $\top_{\text{ub}}$ (by setting a variable ub $= \top_{\text{ub}}$, line 15), which marks the boundary between consistent and inconsistent assignments. As soon as B is filled, it is compared against the worst element in B (line 17). Finally, the function REVERSE() returns the elements of B in reversed order, i.e. best-first.

We may see that this algorithm is correct by ruling out potential incorrect results. First, we may assume correctness for the basic steps of cost computation (lines 15–19 of algorithm 5.2), list sorting (WORST and POPWORST) and variable assignment (line 14). An incorrect result would be a list B^* where at least one true i-th best solution $\mathbf{x}^{*i}$ ($i = 1 \mathinner{..} k$) is not the i-th best assignment *within* B^*, $\mathbf{x}^{*i} \neq B^*_i$. Could $\mathbf{x}^{*i}$ be somewhere else in B^*? Because B^* is sorted, B^*_i and all worse assignments in B^* are worse than the true i-th best solution $\mathbf{x}^{*i}$: $\forall j = i \mathinner{..} k.\, g(\mathbf{x}^{*i}) < g(B^*_j)$. It therefore cannot be at any lower (higher cost) positions $j = i+1 \mathinner{..} k$. It may falsely occur at a higher (cheaper cost) position $j = 1 \mathinner{..} i-1$. But then all true solutions $\mathbf{x}^{*l}, l = j \mathinner{..} i$ cannot be in their respective positions because they are better than $\mathbf{x}^{*i}$ and because B^* is sorted. And just like $\mathbf{x}^{*i}$ these $i - j$ true best solutions cannot occur lower in B^*. They therefore must be either absent from the list or appear at yet higher positions. There they would replace yet another $i - j$ true best solutions, and by induction we see that a set of $i - j$ true best solutions must be absent from the list. In the best case this is one true best solution, which is, with necessary reindexing, $\mathbf{x}^{*i}$.

That is, if we assume an incorrect result B^* as stated above, without loss of generality the true i-th best solution $\mathbf{x}^{*i}$ is missing from the list. This can happen for two reasons only: (1) $\mathbf{x}^{*i}$ was found and put into B (line 7) but later removed as worst assignment within B (lines 8–10), or (2) $\mathbf{x}^{*i}$ was never found because a branch of the search tree was cut off (lines 20–22).

In case (1), since $\mathbf{x}^{*i}$ is only removed from B as the worst assignment, i better assignments must have been found and put into B at this point. But this cannot be the case since by definition only $i - 1$ assignments exist (assignments with equal cost are assumed to be equivalent) that are better than the true i-th best solution $\mathbf{x}^{*i}$, namely the $i - 1$ true best solutions.

In case (2) there must exist a partial assignment $\mathbf{x}'$ that can be expanded to $\mathbf{x}^{*i}$ and an assignment $\mathbf{x}_{\text{ub}}$ that at this point was worst in B for both of which holds: $h(\mathbf{x}') + g(\mathbf{x}') > g(\mathbf{x}_{\text{ub}})$. With an admissible heuristic h, that is one that always underestimates the cost of expanding a given partial assignment $\mathbf{x}'$, we see that all full expansions $\mathbf{x}$ of $\mathbf{x}'$ must have larger cost than $\mathbf{x}_{\text{ub}}$: $\forall \mathbf{x}$ expansion of $\mathbf{x}'.\, g(\mathbf{x}) \geq h(\mathbf{x}') + g(\mathbf{x}') > g(\mathbf{x}_{\text{ub}})$. In particular, the cut-off solution $\mathbf{x}^{*i}$ is more expensive than $\mathbf{x}_{\text{ub}}$. This means $\mathbf{x}_{\text{ub}}$ must be one of the $i - 1$ true best solutions. However, this cannot be the case since $\mathbf{x}_{\text{ub}}$ is the worst of

k assignments in B. It could therefore only be the k-th true best solution, which is a contradiction ($i-1<k$ by our assumptions).

We conclude that algorithm 5.2 cannot produce an incorrect result and is therefore correct.

Algorithm 5.2 k-best branch-and-bound algorithm to solve k-best weighted constraint satisfaction problems (WCSP).

```
function kBestBranchAndBound$((\{X_1,\dots,X_n\},\{D_1,\dots,D_n\},C,k)$, $\top_{\text{ub}}$, heuristic $h$, cost function $g$)
    $B \leftarrow$ kbnbRecurse$((\{X_1,\dots,X_n\},\{D_1,\dots,D_n\},C,k)$, 0, $\bar{\varepsilon}$, $\emptyset$, $h$, $g$)
    return reverse$(B)$
end function
function kbnbRecurse$((X,D,C,k)$, $i$, $\mathbf{x}$, $B$, $h$, $g$)
    if $i > |X|$ then
        $B \leftarrow B \cup \{\mathbf{x}\}$
        if $|B| > k$ then
            popWorst$(B)$
        end if
        return $B$
    end if
    for $v \in D_i$ do
        $\mathbf{x}' \leftarrow \mathsf{assign}(\mathbf{x}, X_i, v)$
        if $|B| < k$ then
            ub $\leftarrow \top_{\text{ub}}$
        else
            ub $\leftarrow g(\text{worst}(B))$
        end if
        if $h(\mathbf{x}') + g(\mathbf{x}') \leq \mathsf{ub}$ then
            $B \leftarrow$kbnbRecurse$((X,D,C), i+1, \mathbf{x}', B, h, g)$
        end if
    end for
    return $B$
end function
```

5.2.2. Encoding System Behavior over Time with Soft Constraints

This work exploits an automatic translation of a PHCA to a constraint net $\mathcal{R}=(X,D,C)$ defined in [129]. We term this translation Υ_{COP}, which maps a model M_{PHCA} to variables X, their finite domains D and local objective functions $c \in C$, called soft constraints. Remember from section 3.3.1 that constraints map partial variable assignments to $[0,1]$

and thus offer a good way of handling the probabilistic parts of PHCA. Next, we explain the translation of PHCA to constraint nets, recapping the work in [129]. After that, we introduce the necessary extensions for plan assessment.

Recap: Translating PHCA to Constraint Optimization Problems

We will focus our recap on the most important parts of the translation. Note that we modified some of the formal elements used in [129] to fit the notation of this thesis, but also to make things more understandable in context of this thesis.

The translation function Υ_{COP} receives as input a PHCA and creates as output the constraint net $\mathcal{R} = (X, D, C)$, consisting of the following variables and constraints (Variables belonging to time step t are marked by superscript ^{t}) [129, p. 330]:

- A set of variables $X^t_\Sigma \cup \Pi^t \cup X^t_{Exec}$ for $t = 0..N$, where $X^t_\Sigma = \{X^t_{l_1}, ..., X^t_{l_{|\Sigma|}}\}$ is a set of variables that correspond to PHCA locations $l_i \in \Sigma$, Π^t is the set of PHCA variables at time t, and $X^t_{\text{Exec}} = \{X^t_{E_1}, ..., X^t_{E_{|\text{Exec}|}}\}$ is a set of auxiliary variables used to encode the PHCA structure and its transition semantic over N time steps. We call the set $\bigcup_t X^t_\Sigma$ the set of location or marking variables.

- A set of finite, discrete-valued domains $D_{X_\Sigma} \cup D_\Pi \cup D_{X_{\text{Exec}}}$, where $D_{X_\Sigma} = \{\{\textsf{marked}, \textsf{unmarked}\}\}$ contains the single domain for variables in X_Σ, D_Π is the set of domains for PHCA variables Π, and D_{Exec} is a set of domains for variables X_{Exec}.

- A set of logical (hard) constraints $R \subseteq C$ that include the behavioral constraints associated with locations within the PHCA and the guard constraints associated with transitions, as well as constraints that encode the structure and the transition semantic of PHCAs.

- A set of soft constraints which encode all probabilistic features, such as the probability distribution P_Ξ of PHCA start states and probabilities associated with PHCA transitions P_T.

Hard constraints such as behavioral PHCA constraints are represented by a soft constraint function mapping (partial) variable assignments disallowed by the constraint to 0 and allowed assignments to 1. To avoid confusion, we refer to the behavioral and guard constraints of a PHCA as PHCA constraints, and constraint net (soft and hard) constraints simply as constraints.

To transition a PHCA between time steps, starting with a marking m^t, possible target locations to be marked at $t+1$ have to be identified, transitions have to be probabilistically chosen and consistency of commands with transition guards and observations with the behavior of the targets needs to be checked. In fact, it involves checking whether all behavior PHCA constraints are consistent with given commands and observations, as they might encode dependencies between locations and composite locations. Later we will see a special kind of dependency between composite locations that can be used to model, for example, the connection between products and stations in manufacturing scenarios. Finally, targets have to be marked correctly regarding, among other things, the hierarchical structure of a PHCA and initial marking.

These semantics are encoded as constraints for single time points, consisting of consistency and marking constraints, and for transitions between time points. The constraint net consists of N copies of these constraints, corresponding to the N time steps of the time window. Marking constraints encode the correct marking of locations to be initially marked, locations that are transition targets, and they take care of propagating markings through the PHCA hierarchy. The size of the constraint net resulting from the translation has $O(N(|\mathcal{T}| + |\Sigma| + |\Pi|))$ variables and $O(N(|\Sigma| + |\mathcal{T}|))$ constraints.

Marking constraints are less interesting here, therefore we focus on consistency and transition constraints and refer to [129] for further details on marking constraints. PHCA constraints are local to locations (behavior) or transitions (guards), i.e., if inconsistent, they render a specific location or transition impossible. In contrast, constraints of the constraint net always globally refer to the complete model. If inconsistent, no solution to the associated COP and therefore no PHCA trajectory exists. This means PHCA constraints cannot be mapped directly to constraints. This is resolved with so-called *consistency constraints*: they explicitly encode consistency of behavior and guards by connecting the PHCA constraints with auxiliary variables $\mathsf{Behavior}_l^t, \mathsf{Guard}_\tau^t \in X_{Exec}$ for locations l and transitions τ at time t. The constraints are constructed, by (the implementation of) Υ_{COP}, as instances of the following higher order rules [129, p. 330]:

Behavioral consistency:

$$\forall t \in \{0..N\}, \forall l \in \Sigma : \mathsf{Behavior}_l^t = \mathsf{consistent} \Leftrightarrow \mathit{behavior}(l)^t$$

Transition guard consistency:

$$\forall t \in \{0..N-1\}, \forall \tau \in \mathcal{T} : \mathsf{Guard}_\tau^t = \mathsf{consistent} \Leftrightarrow \mathit{guard}(\tau)^t$$

The functions *behavior*() and *guard*() evaluate to true iff their associated behavior/guard constraints hold for time t, respectively.

Transition choice constraints encode, for a given location, that a single outgoing transition may be probabilistically enabled at time t. All transitions τ are assigned auxiliary variables $\{T_\tau^t | t \in \{0..N\}\}$ with domain $\{\mathsf{enabled}, \mathsf{disabled}\}$, encoding whether a transition τ is possible in between t and $t+1$, regardless of guard satisfaction. The translation has to do two things: first, ensuring that only exactly one outgoing transition of some location is enabled while all others are disabled (because no two transitions may be taken at the same time), and second, this transition must be chosen probabilistically.

To do the first, the translation follows the following higher-order rule, which formally describes the "enablings" with only a single transition enabled at once (modified version of the same rule in [129, p. 331]).

Probabilistic transition choice:[2]

$$\forall t \in \{0..N-1\}, \forall l_p \in \Sigma_p : \quad (\exists \tau \in \{T | \mathit{source}(T) = l_p\} \Rightarrow \mathfrak{f}_1 \bigwedge \mathfrak{f}_2)$$

where $\mathfrak{f}_1, \mathfrak{f}_2$ are defined as follows:

$$\begin{aligned} \mathfrak{f}_1 \quad &\equiv \quad X_{l_p}^t = \mathsf{marked} \Leftrightarrow (\exists \tau_1 \in \{T | \mathit{source}(T) = l_p\} : \\ &\qquad T_{\tau_1}^t = \mathsf{enabled} \wedge (\forall \tau_2 \in (\{T | \mathit{source}(T) = l_p\} \setminus \{\tau_1\}) : T_{\tau_2}^t = \mathsf{disabled})) \\ \mathfrak{f}_2 \quad &\equiv \quad X_{l_p}^t = \mathsf{unmarked} \Leftrightarrow (\forall \tau_1 \in \{T | \mathit{source}(T) = l_p\} : T_{\tau_1}^t = \mathsf{disabled}) \end{aligned}$$

The first formula encodes the various possibilities of enabled transitions for a given location l_p that is marked. The second encodes situations where this location is unmarked, which forces that all its outgoing transitions have to be disabled.

For the second aspect, the probabilistic choice, the translation encodes the probability distribution over all possible transitions with the following soft constraint function F_T with scope $\{X_{l_p}^t\} \cup \{T_\tau^t | \mathit{source}(\tau) = l_p\}$, which maps each logical model M of the transition choice rule to probability values:

$$F_T(M) = \begin{cases} P_T[l](\tau) & \text{if } (\exists T_\tau^t : T_\tau^t = \mathsf{enabled}) \\ 1.0 & \text{otherwise} \end{cases}$$

The logical model M determines, among other things, the location l, transition τ and time t in the above formula. It can be understood as sort of a propositional instantiation of the transition choice rule that removes the quantification and any higher order elements.

As an example, consider the Idle location of machining station in figure 5.2. It has three outgoing transitions. If one of them should be probabilistically chosen, say $\tau_{\mathsf{Idle} \to \mathsf{Cut}}$,

[2]Where $\{T | \mathit{source}(T) = P\}$ is short for $\{T \in \mathcal{T} | \mathit{source}(T) = P\}$.

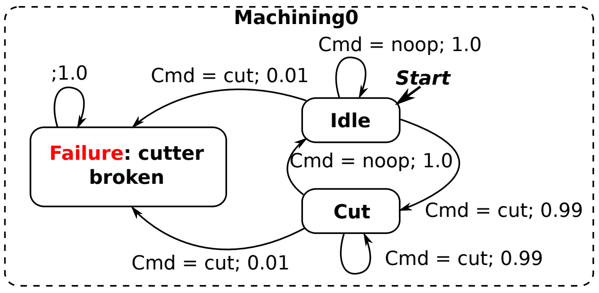


Figure 5.2.: Machining station sub-automaton.

the location must be marked (due to $\mathfrak{f}_1$). The probability of $\tau_{\mathsf{Idle}\to\mathsf{Cut}}$ is obtained as $F_T(M_{\mathsf{Idle},\tau_{\mathsf{Idle}\to\mathsf{Cut}},t}) = 0.99$ for each time point t ($M_{\mathsf{Idle},\tau_{\mathsf{Idle}\to\mathsf{Cut}},t}$ being the model that determines the location, transition and time chosen).

If a transition is enabled with some probability > 0, its guard must be satisfied. This is encoded through transition consistency constraints, which essentially encode $\forall t \in \{0..N\} : \mathsf{Guard}^t_\tau \neq \mathsf{consistent} \Rightarrow T^t_\tau = \mathsf{disabled}$, i.e. that a transition τ cannot be enabled if its guard constraint does not hold.

Other aspects not detailed here are the marking of primitive and composite locations and the initial probabilistic marking of the PHCA. For this, and in general for an in depth discussion of the translation of PHCAs to constraint nets we refer to [129].

Mapping COP Solutions to PHCA Trajectories

Full assignments to variables of the created constraint net $\mathcal{R} = (X, D, C)$ can be projected to assignments to solution variables X^t_Σ for $t \in \{0, \dots, N\}$. These variables with domain $\{\mathsf{marked}, \mathsf{unmarked}\}$ encode the location markings for all time points t, and thus represent PHCA state sequences, i.e. system trajectories. In other words, each COP solution corresponds to a PHCA trajectory. During the translation, a mapping of variables to time points must be stored, such that for each constraint variable X_i its associated time point is known.

Execution Adaptation for Plan Assessment

A plan $\mathcal{P}$ is a sequence $\langle(p, c, t, a)\rangle_j$, each defining for time t two entities p and c to be connected and an action a to perform. In manufacturing, the physical connection of products being worked by particular stations can be mapped to the logical connection of p and c. In that case, p and c are identifiers for products and factory components (usually stations). Particular commands being executed are mapped to action a, which then takes the form of a variable assignment. $\mathcal{P}$ may contain arbitrary many tuples for a single time point to define multiple connections to form and actions to take.

The tuples (p, c, t, a) are realized by execution adaptation functions $\mathcal{E}_{\mathcal{P}}$ that modify the constraint net $\mathcal{R} = (X, D, C)$ resulting from translating a PHCA (instead of first modifying the time-unfolded PHCA and then translating to a constraint net). More precisely, $\mathcal{E}_{\mathcal{P}}$ does the following:

1. It adds constraints to C that encode the link between p and c with equality of variables $\{X_p^t\}$ and $\{X_c^t\}$, specifically introduced as dependent variables into the PHCA for this purpose. These variables are added manually, following a naming scheme that allows $\mathcal{E}_{\mathcal{P}}$ to automatically identify them.

2. It adds unary constraints to encode variable assignments specified in action a. These are mostly assignments to command variables.

The common domain of two variables X_p^t and X_c^t encodes influences, e.g. faulty for station c inflicting damage on product p, or a faulty product p causing unusual observations in c. The value ok encodes that no (harmful) influence is present. To illustrate, consider the machining station and maze product PHCA models shown in figure 5.3. If a plan $\mathcal{P} = \langle \ldots, (\mathsf{Maze0}, \mathsf{Machining1}, t, \mathsf{Cmd} = \mathsf{cut}), \ldots \rangle$ determines that the maze is to be cut by the machining station at time t, then a constraint is added to C that encodes

$$\mathsf{Product}^t_{\mathsf{Machining1}} = \mathsf{Worker}^t_{\mathsf{Maze0}}.$$

The identifiers Product and Worker represent products and stations each from the respective opposite point of view: For stations, Product represents the product the station is working on, and for products, Worker represents the working station. This is one way to identify the link variables (apart from the product and station identifiers p and c) in a PHCA. We implemented a second, more general way where a link variable is identified by adding the code word "LINK" to its name, followed by an arbitrary string identifying the specific link (see the plans and model codes in the appendix, for example listing B.3).

5.2.3. Computing Success Probabilities and Most Probable Diagnoses from K-best Solutions of Constraint Optimization

The approach to plan assessment for a translated and adapted model $\mathcal{E}_{\mathcal{P}}(\Upsilon_{\mathrm{COP}}(M_{\mathrm{PHCA}})) = (X, D, C)$ extends the COP-based diagnosis approach presented in [129]. To compute a diagnosis, the most probable trajectory θ for the given PHCA and observations $\mathbf{o}^{0:t}$ must be computed. In [129] this is done by projecting the best COP solution, i.e. the best

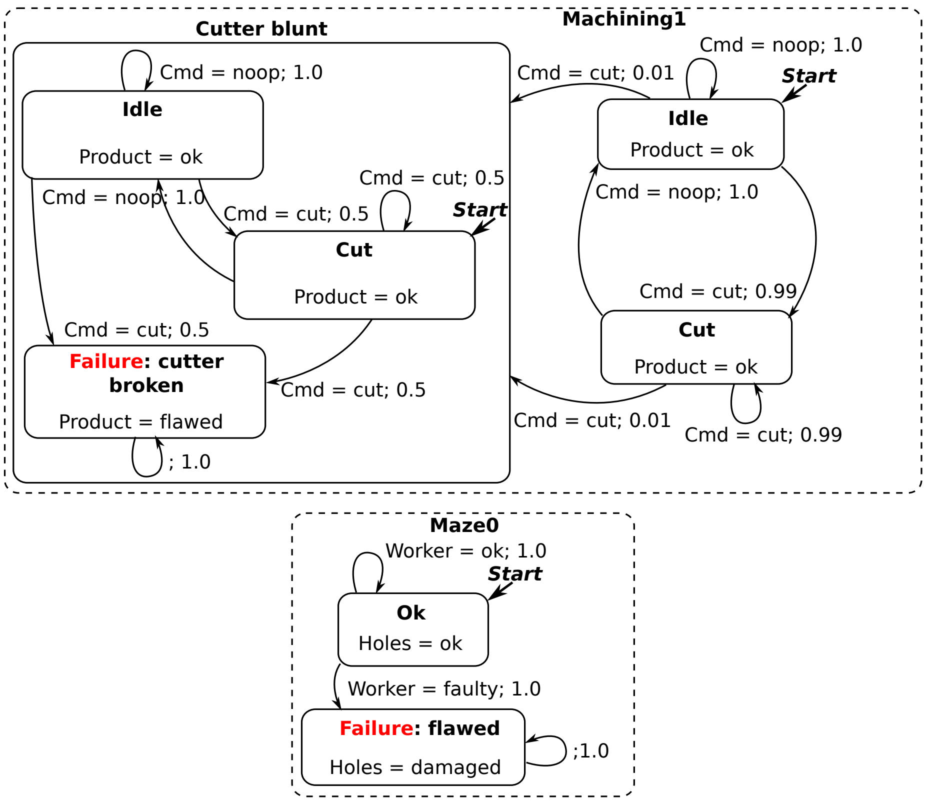


Figure 5.3.: PHCA composite locations modeling a machining station and a maze.

full assignment $\overset{*}{\mathbf{x}}$ to variables X, to the marking variables X_Σ^t for all time points, which yields the most probable trajectory θ.

Success probabilities $Pr(\mathcal{G}_i \,|\, \mathbf{o}^{0:t})$ can be computed based on probabilities $Pr(\theta, \mathbf{o}^{0:t}, \mathbf{o}^{t+1:N})$, which correspond to objective values of the COP solutions:

$$\begin{aligned} Pr(\mathcal{G}_i \,|\, \mathbf{o}^{0:t}) = \sum_{\theta \in \mathcal{G}_i} Pr(\theta \,|\, \mathbf{o}^{0:t}) &= \\ \sum_{\theta \in \mathcal{G}_i} \frac{Pr(\theta, \mathbf{o}^{0:t})}{Pr(\mathbf{o}^{0:t})} &= \\ \frac{\sum_{\theta \in \mathcal{G}_i} Pr(\theta, \mathbf{o}^{0:t})}{\sum_{\theta \in \Theta} Pr(\theta, \mathbf{o}^{0:t})} &= \\ \frac{\sum_{\theta \in \mathcal{G}_i} \sum_{\mathbf{o}^{t+1:N}} Pr(\theta, \mathbf{o}^{0:t}, \mathbf{o}^{t+1:N})}{\sum_{\theta \in \Theta} \sum_{\mathbf{o}^{t+1:N}} Pr(\theta, \mathbf{o}^{0:t}, \mathbf{o}^{t+1:N})} & \end{aligned} \qquad (5.1)$$

The observations $\mathbf{o}^{t+1:N}$ are the potential future observations. Each COP solution of the translation corresponds to a trajectory θ of M_{PHCA} conjoined with some possible future observations $\mathbf{o}^{t+1:N}$. The idea is now to enumerate more than just the best COP solution by solving a k-best COP instead of a COP, i.e. problem 3, then sum over the probabilities of these solutions. The best solution can be picked for diagnosis, all generated solutions to compute $Pr(\mathcal{G}_i \,|\, \mathbf{o}^{0:t})$. At this point, note that with k we refer to the number of trajectories, not solutions for a COP (unless stated otherwise). Their number is typically larger than k by a small factor, depending on the number of free variables. Often trajectories entail certain observations, which means there are no free variables and the factor is thus 1. However, if there are free variables this can lead to a problem, which we describe now.

The solvers we use, Toolbar and Toulbar2, are award-winning open source implementations and lend themselves to implement the k-best versions of A* and branch-and-bound described in section 3.4. However, in order to compute an exact diagnosis, variables for potential future observations have to be summed out. That is, strictly speaking we are interested in computing

$$\arg\max_\theta Pr(\theta, \mathbf{o}^{0:t}) = \arg\max_\theta \sum_{\mathbf{o}^{t+1:N}} Pr(\theta, \mathbf{o}^{0:t}, \mathbf{o}^{t+1:N}). \qquad (5.2)$$

Unfortunately, to our knowledge, no constraint solvers exist that can deal with the three operations maximization, sum (needed to sum out $\mathbf{o}^{t+1:N}$) and product (needed to compute $Pr(\theta, \mathbf{o}^{0:t}, \mathbf{o}^{t+1:N})$) at the same time. While algorithms exist to solve the above

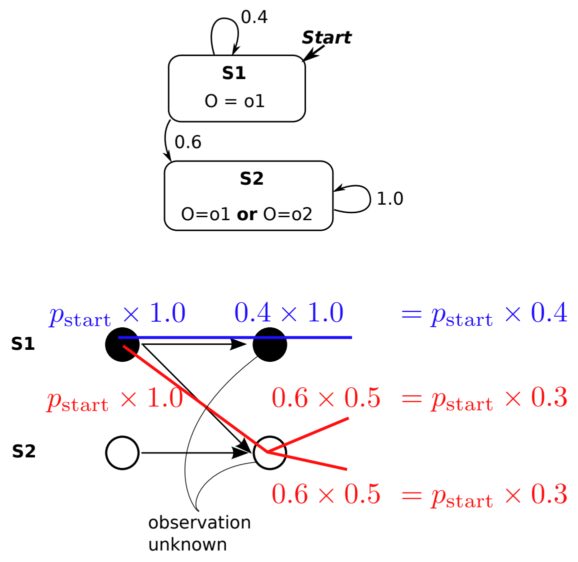

0.4
Start
S1
O = o1
0.6
S2
O=o1 or O=o2
1.0
$p_{\text{start}} \times 1.0$
0.4×1.0
$= p_{\text{start}} \times 0.4$
S1
$p_{\text{start}} \times 1.0$
0.6×0.5
$= p_{\text{start}} \times 0.3$
S2
0.6×0.5
$= p_{\text{start}} \times 0.3$
observation unknown

Figure 5.4.: Example PHCA (shown above, its unfolding over one time step shown below) for which computing diagnosis using a constraint solver would fail.

problem (which corresponds to the probabilistic reasoning problem MAP, problem 4), it seems a framework that allows general constraint solvers has yet to be developed.

Constraint solvers instead compute

$$\arg\max_{\theta, \mathbf{o}^{t+1:N}} Pr(\theta, \mathbf{o}^{0:t}, \mathbf{o}^{t+1:N}), \tag{5.3}$$

which is fine in many situations. However, in some situations, solving 5.3 and simply projecting to θ instead of solving 5.2 can lead to wrong most probable diagnoses. Consider the example PHCA in figure 5.4. For simplicity we assume that we have a single COP variable representing the marking of S1 and S2, and a single observation variable O for the second time step. We have the two potential trajectories $(\mathtt{s1}, \mathtt{s1})$ and $(\mathtt{s1}, \mathtt{s2})$ and the three COP solutions $(\mathtt{s1}, \mathtt{s1}, \mathtt{o1})$, $(\mathtt{s1}, \mathtt{s2}, \mathtt{o1})$, $(\mathtt{s1}, \mathtt{s2}, \mathtt{o2})$. The three full assignments are marked as red and blue lines, their probabilities are $0.4, 0.3, 0.3$, respectively. The colors represent the trajectories. The most probable trajectory is the one corresponding to the red lines, since first the unknown observation would be summed out, yielding probability value $p_{\text{start}} \times 0.6$ for this trajectory. However, a constraint solver, since it maximizes over full assignments instead of trajectories, would wrongly choose the trajectory associated with the blue assignment, which has the largest probability value $p_{\text{start}} \times 0.4$.

A way to remedy this situation is to "simulate" summing over potential future observations by replacing their observation probabilities with 1 during translation. Using the

PHCA definition, we know that

$$\arg\max_{\theta,\mathbf{o}^{t+1:N}} Pr(\theta, \mathbf{o}^{0:t}, \mathbf{o}^{t+1:N}) =$$
$$\arg\max_{\theta,\mathbf{o}^{t+1:N}} P_{\Xi}(m^0) \prod_{u\in\{0..t\}} Pr(\mathbf{o}^u \mid m^u) \prod_{\tau\in\mathcal{T}[\theta]} Pr(\tau) \prod_{u\in\{t+1..N\}} Pr(\mathbf{o}^u \mid m^u)$$

We can see that if we simply remove $\prod_{u\in\{t+1..N\}} Pr(\mathbf{o}^u \mid m^u)$, we basically receive what we wanted. This is achieved by assuming 1 for the factors $Pr(\mathbf{o}^u \mid m^u)$ in this product, which could be done by replacing them during translation. In fact, the soft constraint translation in [129] already does something more subtle, for all $Pr(\mathbf{o}^u \mid m^u)$ it uses the over-approximation

$$f(\mathbf{o}^u \mid m^u) = \begin{cases} 1 & \mathbf{o}^u \text{ is consistent with } m^u \\ 0 & \text{otherwise} \end{cases}$$

to avoid the effort of computing the correct probabilities by counting possible observations for each location. Remember that the PHCA observation model assumes uniform distributions over multiple possible observations. The observation distributions are still uniform, only missing normalization.

As it turns out, this has the same effect as the above described assumption, because we can ignore the cases where $f(\mathbf{o}^u \mid m^u) = 0$ for potential future observations. Since we can assume that there are potential future observations with which a trajectory θ is consistent (unless the model itself is inconsistent), θ will never disappear (receive probability 0). Only the special pairings $(\theta, \mathbf{o}^{t+1:N})$ of θ with those potential future observations it is inconsistent with will be removed.

Since probability values are not needed for diagnosis, the over-approximations are no problem here. However, in some situations they can lead to an error in the exact computation of success probabilities, which can be seen from the results in table 6.2. This leaves us with a design choice for this approach:

1. Correct diagnoses, potentially erroneous success probabilities (even if not approximated).

2. Correct success probabilities, potentially erroneous diagnoses.

3. Compute success probabilities and diagnosis separately, using two specially adapted constraint net translations.

In cases 2 and 3 we have to compute the correct probabilities $Pr(\mathbf{o}^u \,|\, m^u)$ by counting the number of observations $\#\mathbf{o}$ possible for marking m

$$Pr(\mathbf{o}^u \,|\, m^u) := \begin{cases} \frac{1}{\#\mathbf{o}} & \mathbf{o}^u \text{ is consistent with } m^u \\ 0 & \text{otherwise} \end{cases}$$

This is an additional effort, which however is done offline during the translation.

Approximation

Computing $Pr(\mathcal{G}_i \,|\, \mathbf{o}^{0:t})$ exactly requires to generate all trajectories with non-zero probability $\Theta \subseteq St(M_{\text{PHCA}})$. Since this can become intractable quickly, we approximate $Pr(\mathcal{G}_i \,|\, \mathbf{o}^{0:t})$ using only a subset $\Theta(k) \subseteq \Theta$ of the k most probable trajectories:

$$Pr(\mathcal{G}_i \,|\, \mathbf{o}^{0:t}) \approx Pr^k(\mathcal{G}_i(k) \,|\, \mathbf{o}^{0:t}) = \frac{\sum_{\theta \in \mathcal{G}_i(k)} \sum_{\mathbf{o}^{t+1:N}} Pr(\theta, \mathbf{o}^{0:t}, \mathbf{o}^{t+1:N})}{\sum_{\theta \in \Theta(k)} \sum_{\mathbf{o}^{t+1:N}} Pr(\theta, \mathbf{o}^{0:t}, \mathbf{o}^{t+1:N})}.$$

$\mathcal{G}_i(k) \subseteq \Theta(k)$ denotes the set of goal-achieving trajectories among the k most probable. The set $\Theta(k)$ is readily constructed from the solutions enumerated by the modified constraint solvers.

This approach is based on using the k most probable trajectories as an approximation of the distribution over trajectories. This is a sensible approach for our problem domains because it is reasonable to assume that the distribution over trajectories under given observations is peaked, such that a few k trajectories carry most of the probability mass. Since the probabilities involved are failure probabilities, they usually fall into classes of failures which are orders of magnitudes apart in their probability of occurrence, leading to peaked distributions for most observations. Our empirical results in section 6.4 concerning the approximation error support this view.

Generating Sets $\mathcal{G}_i(k)$ and $\mathcal{G}_i$ from Trajectories

Let $G_i = (l, t)$ be a goal, represented as location l being marked at time t. The set of goal achieving trajectories $\mathcal{G}_i(k)$ ($\mathcal{G}_i$ if all trajectories are enumerated, i.e. $k = |\Theta|$) has to be filtered from the sorted list of k most probable trajectories retrieved from a solver. Remember that this set is defined to contain all trajectories that lead to the marking m_l^t, which contains location l at time t: $\mathcal{G}_i = \{\theta \in St(M_{\text{PHCA}}) \,|\, \theta(t) = m_l^t\}$. When translated to a constraint net, marked locations are represented by assignments of the form $X_l^t = \mathsf{marked}$. The set can therefore be created by iteratively putting trajectories

θ into a container if their corresponding constraint solution contains the assignment $X_l^t = \mathsf{marked}$.

Using Constraint Solvers to Compute Most Probable Trajectories

Before trajectories of a translated PHCA model $\mathcal{E}_{\mathcal{P}}(\Upsilon_{\text{COP}}(M_{\text{PHCA}})) = (X, D, C)$ can be computed, known observations have to be added. This is done the same way commands are set, by adding unary constraints (i.e. constraints over a single variable) to C that force assignments of known observation values to respective variables. Note that $\mathcal{E}_{\mathcal{P}}$ merely adds a small number of constraints to the constraint net. This is a small effort compared to solving the COP and thus allows online adaptation. As a final, technical step, the constraint net must be converted into a WCSP (see section 3.3.1), the format, or formalism, accepted as input by the solvers we have chosen.

Those solvers are the algorithm suites known as Toolbar[3] (described further in [27]) and Toulbar2[4]. They have been chosen for their off-the-shelf implementations of strong constraint optimization algorithms and the fact that they are open-source, which allows to easily modify the algorithms. The latter is indeed a more recent version of the former[5].

With Toolbar we followed an approach that was first described in [99]. Given a constraint optimization problem, this approach first generates a heuristic using an approximate version of the cluster tree elimination algorithm [100], called mini-bucket elimination. As mentioned in section 3.3.2, cluster tree elimination can exploit the hidden tree structure of models if they are made available beforehand (in an offline step). Toolbar implements this mini-bucket elimination, and we added an implementation of the k-best A* search (algorithm 5.1) described in section 5.2.1.

In Toulbar2 we implemented the k-best Branch-and-Bound algorithm (algorithm 5.2) shown in section 5.2.1.

5.2.4. Procedure to Evaluate a Plan Against its Goals

Algorithm 5.3 implements a simple evaluation procedure that employs an external constraint solver, such as Toolbar or Toulbar2, to solve the plan assessment problem and evaluate a plan goal based on the result. The function SOLVE() calls the external solver, automatically adapts k to produce enough solutions such that k trajectories are generated

[3]`https://mulcyber.toulouse.inra.fr/projects/toolbar/` (03.2011)
[4]`https://mulcyber.toulouse.inra.fr/projects/toulbar2` (03.2011)
[5]Note that the spelling stems from the fact that these tools were developed by groups in **Tou**louse and **Bar**celona. It seems the developers were more consistent in the second version than in the first with respect to spelling.

and projects COP solutions to trajectories. The function $va(\theta)$ returns the COP variable assignments associated with the trajectory θ, which are needed to separate goal-achieving and violating trajectories. The two functions VoteReplan() and VoteInformation-Gathering() implement a voting for either re-planning if a goal is likely to be not achieved, or information gathering if the success probability is inconclusive. Information gathering could be pervasive diagnosis [106], for example.

Algorithm 5.3 Procedure that uses the success probability of plan $\mathcal{P}_i$ to decide whether to a) continue with it, b) stop it because it probably won't succeed or c) gather more information.

```
1:  procedure EvaluatePlan($\mathcal{R} = (X, D, C)$, $\mathbf{o}^{0:t}$, $\mathcal{P}$, $G_i = (l_i, t_i)$)
2:      $\mathcal{R}' \leftarrow$ add constraints encoding $\mathbf{o}^{0:t}$ to $\mathcal{R}$
3:      $\Theta(k) \leftarrow$ solve($\mathcal{R}'$, $k$)
4:      $\mathcal{G}_i(k) \leftarrow \{\theta \in \Theta(k) \mid (X_{l_i}^{t_i}, \mathsf{marked}) \in va(\theta)\}$
5:      $\theta_{\text{diagnosis}} \leftarrow \Theta(1)$
6:      $p \leftarrow Pr^k(\mathcal{G}_i(k)|\mathbf{o}^{0:t})$
7:      if $p > \omega_{\text{success}}$ then return
8:      else if $p < \omega_{\text{fail}}$ then
9:          VoteReplan($\mathcal{P}$, $G_i$, $\theta_{\text{diagnosis}}$)
10:     else
11:         VoteInformationGathering($\mathcal{P}$,$\Theta(k)$)
12:     end if
13: end procedure
```

5.2.5. K-best Approximation Error

Approximation means error. How does the approximation error depend on k? Can we estimate bounds on the error, or the success probability, depending on k? How can we choose k to minimize the error? We define two error functions that we use to measure the error, the absolute error

$$\epsilon_i(k) = |Pr(\mathcal{G}_i \mid \mathbf{o}^{0:t}) - Pr^k(\mathcal{G}_i(k) \mid \mathbf{o}^{0:t})|$$

and the error relative to the maximum error over all possible values for k,

$$\tilde{\epsilon_i}(k) = \frac{\epsilon_i(k)}{\max\limits_k \epsilon_i(k)}.$$

Empirical results of those measurements are shown in the evaluation in chapter 6.

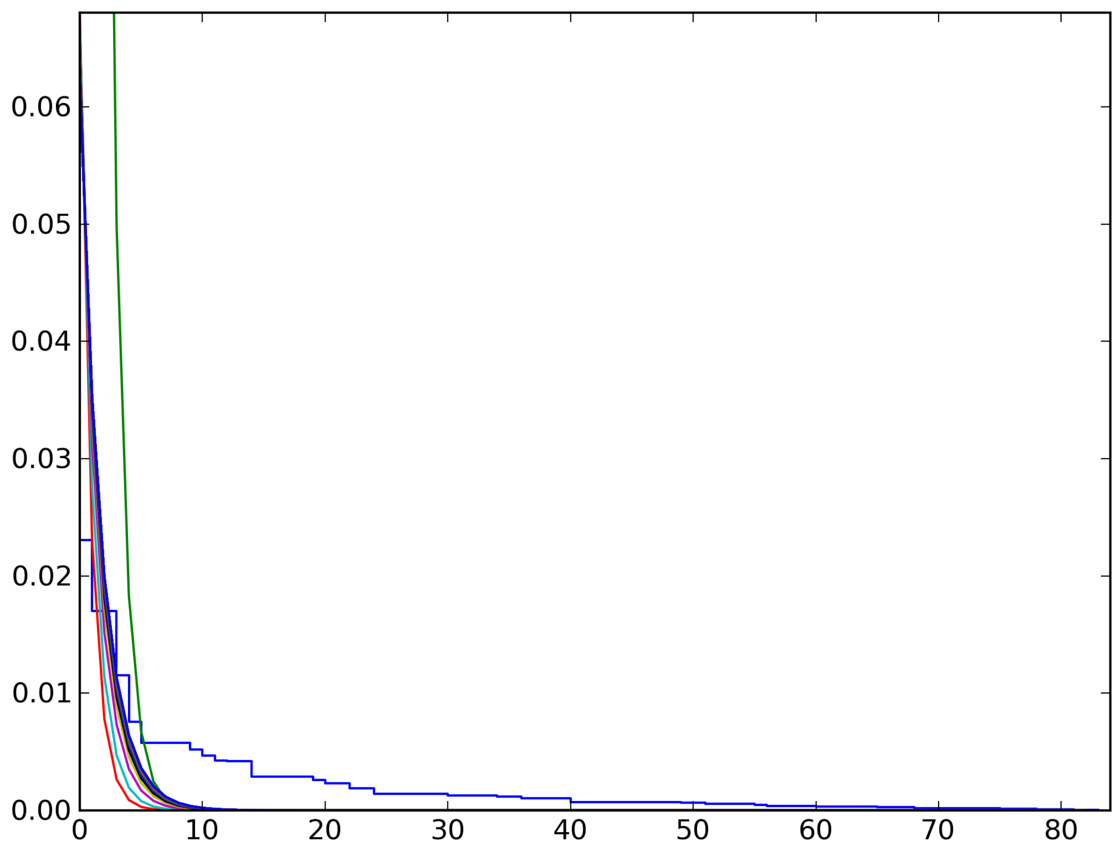


Figure 5.5.: Plot of the function f (step-like plot) and all functions f^* for all enumerated trajectories. The X-axis is the index of a trajectory (sorted by probability value), the Y-axis its probability value. A fitted function f^* was generated for each k.

The most practical question is probably whether we can compute, enumerating only k trajectories, bounds $p_l \leq Pr(\mathcal{G}_i \,|\, \mathbf{o}^{0:t}) \leq p_u$ that guarantee that the success probability lies within the implicitly defined interval. We now sketch an idea on how bounds could be developed based on fitting a function to the distributions over trajectories using the first few enumerated trajectories.

Sketch of How to Estimate the Error Using Function Fitting

We don't know when to stop k-best enumeration, i.e. we need a stop criterion. A valid criterion is "stop when $p_u - p_l \leq \eta$ for some given threshold η", which requires computing the said bounds. First, consider these basic bounds:

Proposition 1. ***Basic bounds for*** $\boldsymbol{Pr}(\mathcal{G}_i \,|\, \mathbf{o}^{0:t})$ *Let* M_{PHCA} *be a PHCA and* G_i *a goal of a plan* $\mathcal{P}$*. Let* $\overline{\mathcal{G}}_i(k) = \Theta(k) \setminus \mathcal{G}_i(k)$ *be those trajectories that violate goal* G*. Then*

$$p_l = \frac{\sum_{\theta \in \mathcal{G}_i(k)} Pr(\theta, \mathbf{o}^{0:t})}{1} \leq \frac{\sum_{\theta \in \mathcal{G}_i} Pr(\theta, \mathbf{o}^{0:t})}{\sum_{\theta \in \Theta} Pr(\theta, \mathbf{o}^{0:t})} = Pr(\mathcal{G}_i \,|\, \mathbf{o}^{0:t})$$

is a lower bound on $Pr(\mathcal{G}_i \,|\, \mathbf{o}^{0:t})$ *and*

$$p_u = 1 - \frac{\sum_{\theta \in \overline{\mathcal{G}}_i(k)} Pr(\theta, \mathbf{o}^{0:t})}{1} \geq 1 - \frac{\sum_{\theta \in \overline{\mathcal{G}}_i} Pr(\theta, \mathbf{o}^{0:t})}{\sum_{\theta \in \Theta} Pr(\theta, \mathbf{o}^{0:t})} = 1 - Pr(\overline{\mathcal{G}}_i \,|\, \mathbf{o}^{0:t}) = Pr(\mathcal{G}_i \,|\, \mathbf{o}^{0:t})$$

an upper bound on $Pr(\mathcal{G}_i \,|\, \mathbf{o}^{0:t})$*.*

These bounds get monotonically closer to $Pr(\mathcal{G}_i \,|\, \mathbf{o}^{0:t})$ the more solutions/trajectories are enumerated, however are still arbitrarily bad as 1 usually vastly over-approximates $Pr(\mathbf{o}^{0:t})$. Our idea is to overestimate $Pr(\mathbf{o}^{0:t})$ as closely as possible to improve the bounds. A possibility to compute an estimation for $Pr(\mathbf{o}^{0:t})$ is to predict the development of $f(i) = Pr(\theta_i, \mathbf{o}^{0:t})$ with increasing index i. Note that indexed trajectories θ_i are sorted by decreasing probability, just as they are enumerated. The prediction can be made by assuming that the probability values lie on the curve of some function $f^* : \mathbb{R}_0^+ \to \mathbb{R}_0^+$. Interesting candidates are exponential functions $f^*_{\alpha,\gamma}(x) = \gamma * e^{-(x-1)*\alpha}$, the Weibull distribution or functions that model heavy tailed distributions. The functions are then fitted to the probability values of the generated trajectories.

We illustrate the idea using the exponential function class $f^*_{\alpha,\gamma}(x) = \gamma * e^{-(x-1)*\alpha}$. The estimate for $Pr(\mathbf{o}^{0:t})$ can be computed as

$$p_o^* = \sum_{i=0}^{k} f(i) + \int_{k+1}^{\infty} f^*_{\alpha,\gamma}(\tau) d\tau.$$

As said, the estimate must be conservative, i.e.

$$\int_{k+1}^{\infty} f^*_{\alpha,\gamma}(\tau) d\tau \geq \sum_{i=k+1}^{k_{\max}} f(i).$$

We assume that the probability values lie on the curve of f^*, which means we can compute its parameters α, γ such that the error $\varsigma(i)$ in $f(i) = f^*_{\alpha,\gamma}(i) + \varsigma(i)$ is minimized. The error is assumed to be gaussian with mean 0. The parameters could be estimated using standard approaches such as linear regression. Depending on how quick the fitting is it could be done online, using the first few enumerated trajectories to estimate the parameters such as α, γ, or one could use typical problem instances to generate a general fitted function for all instances of interest.

However, since we cannot compute $\sum_{i=k+1}^{k_{\max}} f(i)$, it remains unclear how the curve fitting could be guaranteed not to violate the required conservativeness. We assume that choosing the right model for fitting would minimize potential violations. In our first tests, however, it seemed that exponential functions are the wrong candidate class. Figure 5.5 shows the result of fitting an exponential function to f using the Levenberg-Marquart non-linear least-squares algorithm [114][122].

```
bConsistent(t,Cutter_broken) <=> var_PRODUCT(t,Faulty).
bConsistent(t,Cutterblunt_cut) <=> var_PRODUCT(t,Ok).
...
gConsistent(t,Cutter_broken_to_cutter_broken).
gConsistent(t,Cutterblunt_idle_to_cutterblunt_idle) <=>
var_CMD(t,Nocommand).
...
gConsistent(t,Idle_to_cut) <=> var_CMD(t,Cut).
...
locMarked(t,l) => bConsistent(t,l).
...
var_MAZE0_WORKER(T1,x) <=> var_MACHINING1_PRODUCT(T1,x).
var_MAZE0_WORKER(T2,x) <=> var_MACHINING1_PRODUCT(T2,x).
```

Listing 5.1: Excerpt of logical formulas $\mathcal{L}$ from the translation of the machining station shown in figure 5.6b. Abbrev.: bConsistent stands for behaviorIsConsistent, gConsistent stands for guardIsConsistent.

5.3. Probabilistic Reasoning Approach

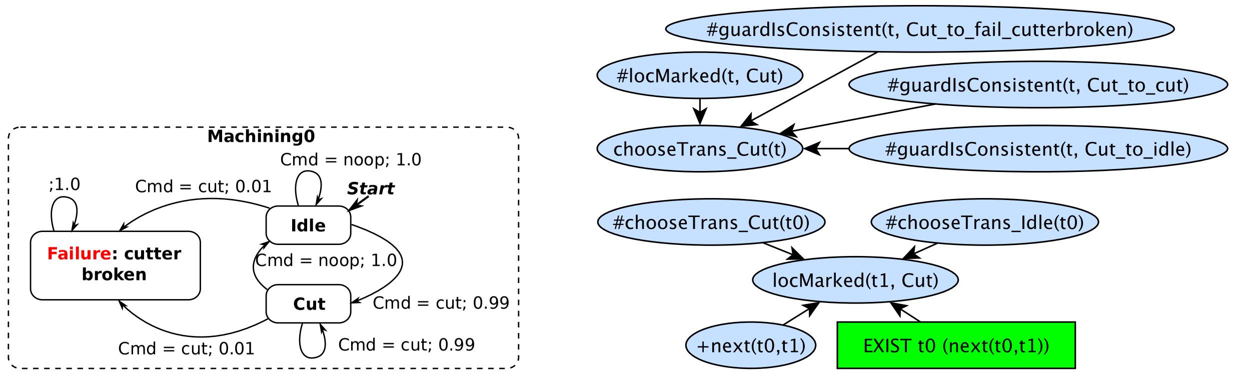
From a probabilistic reasoning point of view, the diagnosis part of plan assessment corresponds to computing most probable a posteriori hypotheses (MAP) and the success probability part to computing marginals. MAP and the computation of marginals are long time standard problems in probabilistic reasoning with existing off-the-shelf solutions. Those solutions often accept probabilistic models represented as Bayesian networks (BN) as input. We now describe an approach to plan assessment that is based on a new translation of PHCA models to abstract, generalized Bayesian networks, which can in turn be automatically instantiated to BNs. This opens up the possibility of comparing the COP-based approach from the previous section with methods and tools from the probabilistic reasoning community. We translate PHCAs to Bayesian logic networks [95], see section 3.7. In particular, we use the BLN toolbox[6] developed by Dominik Jain.

5.3.1. Translating PHCAs to Bayesian Logic Networks

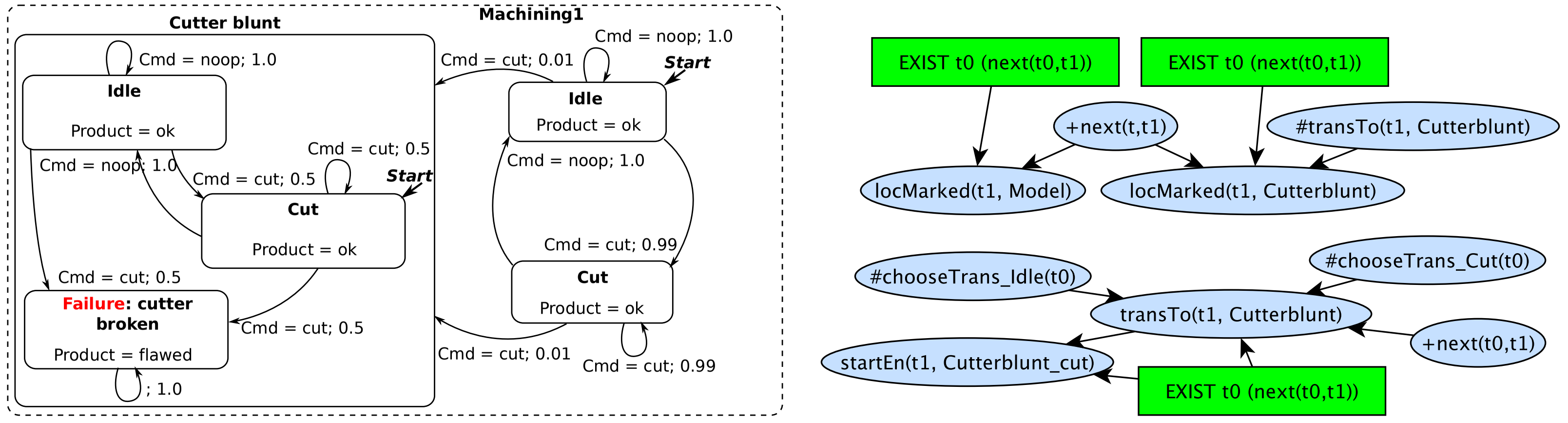
Our novel translation is conceptually similar to the COP encoding Υ_{COP} of PHCAs. This encoding is defined in terms of formal higher-order rules for structure, probabilistic behavior and consistency with observations and commands in [129], some of which have been shown in section 5.2.2.

We adapt these rules for the translation to BLN. To a large extent, we keep the structure of the translation described in [129], which means we have roughly the same number of

[6] `http://www9-old.cs.tum.edu/people/jain/dl.php?get=probcog` (06.2011)



(a)



(b)

Figure 5.6.: Showing excerpts of the BLN fragments from translating the machining station model with simple (5.6a) and hierarchical fault (5.6b). In 5.6a, excerpts of location marking and probabilistic transition choice are shown, for 5.6b excerpts of composite and full target marking.

rules for the same purposes. The structure of the text below describing the rules reflects this. The contribution of this thesis here is, besides an implementation of the translator, to modify or reformulate the rules from [129] such that they are closer to the first-order logical language of the BLN framework. For example, we introduce logical predicates that can be easily translated into the BLN language. Often, these predicates can be transferred with practically no changes. As a side effect, in our view some of the rules are now easier to understand, e.g. the rule for probabilistic transition choice.

The translation function Υ_{BLN} takes as input a model M_{PHCA} and creates a BLN $\mathcal{B} = (\mathcal{D}, \mathcal{F}, \mathcal{L})$ and a knowledge base DB. As explained in section 3.7, $\mathcal{B}$ consists of the declarations $\mathcal{D}$, the set of fragments $\mathcal{F}$ and the set of first-order logic formulas $\mathcal{L}$. The knowledge base defines existing objects or entities for the first-order logic formulas and fragments as well as known facts about relations among these entities. When the BLN is grounded, DB is extended with further evidence. Execution adaptation functions $\mathcal{E}_{\mathcal{P}}$ for BLNs add formulas to $\mathcal{L}$ and facts to DB.

We know from definition 3 in section 4.1 that PHCAs define composite and primitive locations and guarded, probabilistic transitions between them. Together, they determine the evolution of location markings. Consequently, the BLN encoding revolves around a predicate $\mathit{locMarked}(t, l)$, which evaluates to true if location l is marked at time t. When grounded, this predicate is instantiated to boolean variables L_i^t, which in turn encode location markings at specific time points. Additional predicates encode, e.g., probabilistic transition choice, hierarchical structure and step-wise transitioning. Those will be introduced together with the rules in which they appear. The predicates are declared in $\mathcal{D}$, while the rules are encoded as logical formulas in $\mathcal{L}$ or, to a larger extent, as fragments in $\mathcal{F}$.

PHCA Constraint Consistency

Like for the translation to soft constraints we have to handle the consistency of PHCA behavior and guard constraints explicitly. For the BLN translation we define, as formulas in $\mathcal{L}$, behavior and transition guard consistency predicates in terms of formulas over assignments of PHCA variables O and Cmd. The formal rules that define these predicates are similar to the behavioral and transition guard consistency rules in the soft constraint encoding.

Behavior consistency predicate:

$$\forall t \in \{0..N\}, \forall l \in \Sigma.\ \mathit{behaviorIsConsistent}(t, l) \Leftrightarrow \mathit{behavior}(l, t)$$

Transition guard consistency predicate:

$$\forall t \in \{0..N-1\}, \forall \tau \in \mathcal{T}.\ guardIsConsistent(t,\tau) \Leftrightarrow guard(\tau,t)$$

The functions *behavior*() and *guard*() map to time dependent versions of the behavior and guard constraints of location l and transition τ, respectively. When specifying the above formulas in $\mathcal{L}$, functions *behavior*() and *guard*() need to be evaluated, while the rest can be represented directly. As an example, consider line 1 in listing 5.1: *behavior*() evaluates to a concrete behavior constraint that specifies that the behavior of location "broken" (which is part of composite location "cutter") is consistent if and only if the product being processed will be broken in the next time step. An example for a concrete rule that determines the consistency of a particular transition guard (i.e. the evaluation of *guard*() for a particular transition) is line 8 in listing 5.1, which requires that the transition from location Idle to location Cut can only be taken if and only if the command for the machining station is cut.

In addition, $\mathcal{L}$ contains the general rule (line 10 in listing 5.1) that, for all points in time, a location's behavior must be consistent if it is marked.

Marked location behavior:

$$\forall t \in \{0..N\}.\ \forall l \in \Sigma.\ locMarked(t,l) \Rightarrow behaviorIsConsistent(t,l)$$

A similar rule holds for guard constraints. However, it is more convenient to combine this rule with the rule for probabilistic transition choice.

Location Marking

Now we look at how locations are marked: Because they are initially marked, because they are the target of transitions or indirectly, because of the PHCA hierarchy. In general, the predicate $locMarked(t,l)$ encodes a location l being marked at time t. Certainly a location becomes marked if it is the destination of some transition, i.e. transitioned to the time step before. This is reflected in the *transTo*() predicate. However, some locations are designated start locations. They can also become marked as a result of being chosen as a starting point for a new (sub-)trajectory. In that case the start location in question is being enabled, which is reflected in the *startEnabled*() predicate, and can become marked as a result of that. In both cases, the location in question is considered the *target* of a potential marking, hence it is called target marking. The initial marking of locations is a special case, since no incoming transitions are possible. This is reflected in some special rules. Finally, rules are needed to cope with the hierarchical structure of a PHCA model.

We start with the initial marking of so-called top-level locations of a PHCA. If M_{PHCA} is a PHCA, then top-level composite locations are those that have M_{PHCA} itself as parent. In our examples, composite locations modeling stations and products are always top-level locations. These locations are always initially marked.

Initial top level marking:

$$\forall l \in \Sigma_{\mathrm{top}}.\ \mathit{locMarked}(T0, l)$$

$T0$ is the entity representing the first time step, the set $\Sigma_{\mathrm{top}} \subseteq \Sigma$ contains all top-level locations. The above rule is encoded in DB, e.g. for the machining station 1 the following code is added: `locMarked(T0, Machining1) = True`.

The initial marking of other locations results from propagating the marking down the PHCA hierarchy, starting from the explicitly marked top-level locations. The interplay of two rules is responsible for this propagation. One rule is concerned with enabling start locations of composite locations that are marked, the other with in turn marking enabled start locations. We explain the first rule momentarily when we address hierarchy in general. The second rule, named “initial marking/unmarking” is shown below. Our version of this rule is deterministic and therefore a special case of the PHCA probabilistic initial marking defined as distribution P_Ξ.

Initial marking/unmarking:

$$\forall l \in \Sigma \setminus \Sigma_{\mathrm{top}}.\ \mathit{startEnabled}(T0, l) \Leftrightarrow \mathit{locMarked}(T0, l)$$

This translates to the following deterministic conditional probability function:

$$Pr(L_l^0 = \mathsf{marked} \mid \mathsf{Start}_l^0) = \begin{cases} 1 & \mathit{startEnabled}(T0, l) \\ 0 & \neg\mathit{startEnabled}(T0, l) \end{cases}$$

This brings up the question whether predicate *startEnabled*() is actually needed. After all, if every start location first becomes enabled and then immediately marked, this predicate seems unnecessary. The translation to constraint nets developed in [129] had an analogous construct in form of a higher-order variable, which we think was introduced to allow probabilistic initial marking. We didn’t adopt the probabilistic marking developed in [129] for reasons we explain shortly. Yet we stuck with the *startEnabled*() construct since it is used in a number of rules and it is not clear at all whether removing it would in the end yield a more complicated logical description of PHCA semantics. This is a spot where our translation could possibly be improved.

The probabilistic marking from [129] was implemented with the following function (adapted to our notation and our *startEnabled*() predicate):

$$Pr(L_l^0 = \mathsf{marked} \mid \mathsf{Start}_l^0) = \begin{cases} p_l & \mathit{startEnabled}(T0, l) \\ 0 & \neg \mathit{startEnabled}(T0, l) \end{cases}$$

This means start location l is not marked deterministically if it is enabled, but with probability p_l. This marking is of course more general than ours, but still a special case of the PHCA probabilistic choice P_Ξ. It allows to model uncertainty about the initial conditions of a technical system by defining, for each start location, a distribution over its domain $\{\mathsf{marked}, \mathsf{unmarked}\}$. However, this leads to two problems. First, these distributions allow the unlikely yet not negligible case of every start location being chosen as unmarked. In other words, in this case no location would be marked initially, a situation that doesn't make sense. Second, this probabilistic initial marking doesn't clearly differentiate the cases where, on the one hand, we want to probabilistically choose between two start locations, and on the other hand, where we want two start locations to be marked *simultaneously*. For these reasons we decided to not use this initial marking and restrict our translation to deterministic initial conditions.

Next, we look at target marking, that is the marking of locations that are enabled start locations or that are being transitioned to (at some time point other than the initial one). Primitive locations as targets are handled by the primitive target marking rule. It marks primitive locations if they are either transitioned to or if they are enabled starting locations.

Primitive target marking:

$$\forall t_0 \in \{0..N-1\}.\ \forall t_1 \in \{1..N\}.\ \forall l_p \in \Sigma_p.\ \exists l \in \mathit{parents}(l_p).\ \mathit{next}(t_0, t_1) \quad \Rightarrow$$
$$(\mathit{target}(\mathit{chooseTrans}(t_0, l)) = l_p \vee \mathit{startEnabled}(t_1, l_p) \Leftrightarrow \mathit{locMarked}(t_1, l_p))$$

The function *target* maps transitions to their target locations and *parents* maps a location to the set of locations connected to it via transitions (in a similar fashion as *parents*() defined for BNs maps a BN node to those connected to it via incoming arcs). The function *chooseTrans*() is responsible for probabilistically choosing transitions. It will be explained shortly in the context of the probabilistic transition choice rule.

The composite target marking rule, just like the rule for primitive targets, marks a composite location if it is transitioned to or if it is an enabled start location. The difference is that we now introduce the explicit *transTo*() predicate. While we follow the assumption that fewer predicates will lead to a more compact and readable translation (as is the case for the primitive target marking), composite locations, because they are more complex to handle, turned out to be an exception: We found that introducing this extra predicate made the translation less complicated.

Composite target marking:

$$\forall t \in \{1..N\}.\ \forall l_c \in \Sigma_c.\ \mathit{transTo}(t, l_c) \vee \mathit{startEnabled}(t, l_c) \Rightarrow \mathit{locMarked}(t, l_c)$$

The *transTo* predicate is defined as follows:

$$\begin{aligned}\forall t_0 \in \{0..N-1\}.\ \forall t_1 \in \{1..N\}.\ \forall l_c \in \Sigma_c.\ \mathit{next}(t_0, t_1) \quad \Rightarrow \\ (\mathit{transTo}(t_1, l_c) \Leftrightarrow \exists l \in \mathit{parents}(l_c).\ \mathit{target}(\mathit{chooseTrans}(t_0, l)) = l_c)\end{aligned}$$

To properly reflect the PHCA hierarchy we have to handle the marking of start locations that are sub-locations and we have to ensure that sub-locations may only be marked if and only if their parent location is marked. This leads to the following two rules.

The full target marking rule ensures that all start sub-locations (given by function *subStart*()) of a composite location are enabled (and thereby ready to be marked) if and only if this composite location is the target of a chosen transition or is itself enabled.

Full target marking:

$$\begin{aligned}\forall t \in \{1..N\}.\ \forall l_c \in \Sigma_c.\ (\mathit{transTo}(t, l_c) \vee \mathit{startEnabled}(t, l_c) \quad \Leftrightarrow \\ \forall l \in \mathit{subStart}(l_c).\ \mathit{startEnabled}(t, l))\end{aligned}$$

We treat the initial time point $t = 0$ separately with the following rule, which ensures that the start locations of initially marked locations are enabled.

Initial full marking:

$$\forall l_c \in \Sigma.\ \forall l \in \mathit{subStart}(l_c).\ \mathit{locMarked}(T0, l_c) \Rightarrow \mathit{startEnabled}(T0, l)$$

The hierarchical marking/unmarking rule ensures that a composite location is marked if and only if at least one of its sub-locations (which are given by function *sub*) is marked.

Hierarchical marking/unmarking:

$$\forall t \in \{0..N\}.\ \forall l_c \in \Sigma_c.\ \mathit{locMarked}(t, l_c) \Leftrightarrow \exists l \in \mathit{sub}(l_c).\ \mathit{locMarked}(t, l)$$

To implement this rule, our choice was to have the translation create fragments and add them to $\mathcal{F}$. However, a more elegant way is probably to create partially instantiations of this rule in form of BLN logical formulas that are added to $\mathcal{L}$. For example, the following formula could be added for the composite location Cutter blunt:

```
  locMarked(t,Cutterblunt) <=> locMarked(t, Cutterblunt_idle) v locMarked(t,
Cutterblunt_cut) v locMarked(t, Cutterblunt_broken).
```

The difference between these two encodings mostly lies in the readability of the translation result. When grounded, the above formula will be converted to a Bayesian net very similar to the one that will result from the fragments that are the result of our current translation.

Probabilistic Transition Choice and Guard Consistency

The central rule for probabilistic behavior is probabilistic transition choice. Given a primitive location, exactly one of its outgoing transitions may be chosen (according to transition probabilities defined in the model), and that if and only if the location is marked and the chosen transition's guard is consistent.

Probabilistic transition choice:

$$\forall t \in \{0..N\}.\ \forall l_p \in \Sigma_p.\ \exists \tau \in \mathit{outgoing}(l_p) \cup \{\tau_\varepsilon\}.$$
$$\mathit{locMarked}(t, l_p) \wedge \mathit{guardIsConsistent}(t, \tau) \Leftrightarrow \mathit{chooseTrans}(t, l_p) = \tau \wedge \tau \neq \tau_\varepsilon$$

In the formula, function $\mathit{chooseTrans}(t, l_p)$ maps time and location to an admissible outgoing transition. The function $\mathit{outgoing}()$ maps primitive locations to their set of outgoing transitions. τ_ε denotes the empty transition, which indicates that no transition has been chosen. The translation eliminates quantification over l_p and creates $\mathit{chooseTrans}()$ functions for each location separately (see figures 5.6a and 5.6b for examples). Their CPTs define the following probability function:

$$Pr(T^t_{l_p} = \tau \mid L^t_p, \mathbf{G}^t) = \begin{cases} P_T[l_p](\tau) & \text{if (a)} \\ 1 & \text{if (b)} \\ 0 & \text{otherwise} \end{cases} \tag{5.4}$$

where $T^t_{l_p}$ is a random variable for choosing among l_p's outgoing transitions, L^t_p encodes $\mathit{locMarked}(t, l_p)$ and $\mathbf{G}^t$ is a vector of random variables encoding $\mathit{guardIsConsistent}(t, \tau)$ for each outgoing transition. Condition (a) is $\mathit{locMarked}(t, l_p) \wedge \mathit{guardIsConsistent}(t, \tau)$ and (b) is $(\neg \mathit{locMarked}(t, l_p) \vee (\neg \exists \tau' \in \mathit{outgoing}(l_p).\ \mathit{guardIsConsistent}(t, \tau'))) \wedge \tau = \tau_\varepsilon$.

The latter condition means “don't care" if the location isn't marked or all outgoing transitions are inconsistent, and expect *chooseTrans*() to return τ_ε. The remaining cases, e.g. *chooseTrans*() returning an inconsistent transition, are ill-conditioned and hence yield probability 0. Here we see that this fragment also encodes the consistency with guards, which is more compact than having a separate logical rule as for marked location behavior.

Execution Adaptation for Plan Assessment

We now address BLN translation related to adaptation functions $\mathcal{E}_\mathcal{P}$. Remember that a plan $\mathcal{P}$ is a sequence $\langle(p, c, t, a)\rangle_j$, each defining an action a to perform and two entities p and c to connect at time t. The actions, commands typically, are encoded as facts in DB. The logical connection between p and c is realized with logical formulas in $\mathcal{L}$ that enforce equality of variables $\{X_p^t\}$ and $\{X_c^t\}$, according to the following rule:

$$\forall(p, c, t, a) \in \mathcal{P}.\ \forall(X_p, X_c) \in \Pi \times \Pi.\ X_p^t = X_c^t$$

Just like in the case of the soft constraint translation, multiple pairs (X_p, X_c) of variables may encode multiple different types of influences for the same component link. For each such pair a logical formula is added to $\mathcal{L}$. The common domain of such a pair encodes the specific influence, e.g. faulty for damage and ok for no (harmful) influence. The variables are defined in the PHCA model, as part of the PHCA variables Π. As an example, formulas 12 and 13 in listing 5.1 encode a maze (p) being worked by a machining station (c) for the first two time steps. The predicates `var_MAZE0_WORKER` and `var_MACHINING1_PRODUCT` encode assignments of $\{\mathsf{ok}, \mathsf{faulty}\}$ to two variables X^t_{Maze0} and $X^t_{\mathsf{Machining1}}$, respectively. More precisely, the variables are, again, $\mathsf{Worker}^t_{\mathsf{Maze0}}$ and $\mathsf{Product}^t_{\mathsf{Machining1}}$, according to the identification scheme mentioned for the soft constraint encoding of component links in section 5.2.2.

This might not be the most efficient encoding, as it does not fully exploit the expressivity of BLNs. One can imagine a translation that creates for each link (p, c, t, a) only one formula in $\mathcal{L}$, which then quantifies over all pairs (X_p, X_c).

Creating Fragments for Rules

Using the location Cutter blunt and the composite marking rule as an example, we show how the rules not encoded in $\mathcal{L}$ are translated into fragments in $\mathcal{F}$. Generally, one fragment is created for each predicate occurring in a rule, except if the translator can determine, e.g. from the model structure, that a predicate is always true or false. The implementation of Υ_{BLN} partially instantiates the predicates, i.e. it removes all quantification except over time. In case of Cutter blunt, fragments for partially instantiated predicates *transTo*(t, Cutter blunt) and *locMarked*(t, Cutter blunt) are created. No fragment is created for *startEnabled*(t, Cutter blunt) because Cutter blunt is no start location and the predicate thus always false. The following table shows the CPT template for *locMarked*(t, Cutter blunt):

transTo(t, Cutter blunt)	T	F
locMarked(t, Cutter blunt) $= T$	1	0.5
locMarked(t, Cutter blunt) $= F$	0	0.5

The CPT encodes that Cutter blunt is marked if it is being transitioned to. If not, the CPT doesn't influence the marking. See figure 5.6b for the partial fragment network corresponding to the composite target marking rule for this location. In the graphical BLN notation, not all elliptical nodes define their own fragments. The node `+next(t,t1)` works as a precondition: its children are only valid if it evaluates to true. In this example, it means there must be a previous time point for the fragments to be valid. This corresponds to the composite target marking rule not being valid for the initial time point $t = 0$ (which is treated separately). A small remark at this point about notation: Occasionally, we will refer directly to BLN code using `typewriter` font, e.g. to formulas such as `+next(t,t1)`.

5.3.2. Translation Correctness

Remember the PHCA joint distribution over trajectories and observations given in 4.1. We show that, assuming that our translation rules correctly describe PHCA marking evolutions, this distribution is actually encoded by Bayesian nets (BN) resulting from our translation. Formally, we would like to show:

Theorem 1. *Let (X, D, G, P) be the Bayesian net grounding of a Bayesian logic net retrieved from translating a PHCA model M_{PHCA}. Let $\theta = (m^{t_0}, m^{t_1}, \ldots, m^{t_N})$ be an arbitrary marking sequence of M_{PHCA}, valid according to possible evolutions of M_{PHCA}. Then*

$$Pr(\theta, \mathbf{O}^{0:t} = \mathbf{o}^{0:t}) = \sum_{\mathbf{O}_{\text{BN}}^{t+1:N}} Pr(\mathbf{l}^{t_0}, \ldots, \mathbf{l}^{t_N}, \mathbf{O}_{\text{BN}}^{0:t} = \mathbf{o}^{0:t}, \mathbf{O}_{\text{BN}}^{t+1:N} \mid \mathbf{X}_{\text{aux}} = \mathsf{true})$$

The proof idea is to map each of the relevant factors of the distribution of the BN to corresponding factors of the PHCA distribution, defined in equation 4.1. In the following, $\mathbf{L}^{t_j}$ are vectors of BN location marking variables $L_i^{t_j} \in X$ for each time point t_j, and $\mathbf{l}^{t_j}$

are according vectors of binary values. An assignment $\mathbf{L}^{t_j} = \mathbf{l}^{t_j}$ is thus a BN encoding of a PHCA marking. We abbreviate the assignments $\mathbf{L}^{t_j} = \mathbf{l}^{t_j}$ with $\mathbf{l}^{t_j}$. $\mathbf{O}_{\text{BN}}$ is a vector of BN observation variables $O_l^{t_j} \in X$ for each time point t_j (l ranges over indices of observation variables for a given time point). Similarly to the soft constraint encoding of PHCA, $X_{\text{exec}} = X \setminus \{L_i^{t_j}\} \cup \{O_l^{t_j}\} \cup X_{\text{aux}}$ is the set of helper variables that ease the translation. They result from the grounding of predicates such as *guardIsConsistent*() or *startEnabled*(). X_{aux} is the set of auxiliary variables added to the BN for each concrete logical rule instantiated from abstract rules in $\mathcal{L}$. In the formula, vectors of the variables in X_{exec} and X_{aux} are used.

The BN distribution for a given assignment to the $\mathbf{L}^{t_j}$-variables is

$$Pr(\mathbf{l}^{t_0}, \ldots, \mathbf{l}^{t_N}, \mathbf{O}_{\text{BN}}^{0:t} = \mathbf{o}^{0:t}, \mathbf{O}_{\text{BN}}^{t+1:N}, \mathbf{X}_{\text{exec}} \,|\, \mathbf{X}_{\text{aux}} = \mathsf{true}).$$

The variables $\mathbf{X}_{\text{exec}}$ encode the deterministic PHCA structure that underlies marking sequences. Therefore, they are completely determined by $\mathbf{l}^{t_0}, \ldots, \mathbf{l}^{t_N}$, and can be left out. This leaves us with the simpler term

$$Pr(\mathbf{l}^{t_0}, \ldots, \mathbf{l}^{t_N}, \mathbf{O}_{\text{BN}}^{0:t} = \mathbf{o}^{0:t}, \mathbf{O}_{\text{BN}}^{t+1:N} \,|\, \mathbf{X}_{\text{aux}} = \mathsf{true}).$$

Furthermore, we regard only full assignments to the variables of the BN that are structurally consistent. If this is not the case for some full assignment, for example if this assignment violates the PHCA hierarchy, at least one conditional probability table of a helper variable X_{exec} evaluates to 0. This means that structurally inconsistent assignments don't contribute to the distribution and thus can be ignored.

Before we continue with the formal proof of theorem 1, we show that translations actually exist and that distributions by BNs resulting from our translation take a specific factorial form.

Lemma 1. *For any given PHCA M_{PHCA}, at least one BLN exists which defines, via ground BN (X, D, G, P), the same distributions over trajectories as M_{PHCA}.*

Proof. This lemma follows from the facts that PHCA represent HMM [162] and that BLN are a generalization of HMM [95]. □

The factorization of the BN distribution is determined by conditional dependency relations among the variables, which in turn are determined by the PHCA semantics captured in the translation rules. We conjecture that the rules themselves correctly capture the PHCA semantics, since we assume the original rules described in [129] to be correct:

Conjecture 1. *The rules defined in section 5.3.1 correctly describe the evolution of PHCA markings over time as defined in [162].*

The BN factorizes then as follows:

Lemma 2. *Let (X, D, G, P) be the BN resulting from translating and grounding a PHCA M_{PHCA}. Then the following factorization holds:*

$$\begin{gathered} Pr(\mathbf{L}^{t_0}, \ldots, \mathbf{L}^{t_N}, \mathbf{O}_{\text{BN}}^{0:t}, \mathbf{O}_{\text{BN}}^{t+1:N} \mid \mathbf{X}_{\text{aux}}) = \\ Pr(\mathbf{L}^{t_0} \mid \mathbf{X}_{\text{aux}}) \cdot \prod_{u \in \{1..N\}} Pr(\mathbf{L}^u \mid \mathbf{L}^{u-1}, \mathbf{X}_{\text{aux}}) \cdot \\ \prod_{u \in \{0..t\}} Pr(\mathbf{O}_{\text{BN}}^u \mid \mathbf{L}^u, \mathbf{X}_{\text{aux}}) \cdot \prod_{u \in \{t+1..N\}} Pr(\mathbf{O}_{\text{BN}}^u \mid \mathbf{L}^u, \mathbf{X}_{\text{aux}}) \end{gathered}$$

Proof. The lemma holds if each of the factors is actually present and if conditional independencies that hold for PHCA models are not changed by the translation rules.

Each of the above factors results from a subset of the translation rules. The factors are composed of multiple conditional probability tables, which stem from the fragments of the predicates that occur in the rules.

1. The factor $Pr(\mathbf{L}^{t_0} \mid \mathbf{X}_{\text{aux}})$ corresponds to the initial distribution at the first time point. The respective rules are "initial top level marking", "initial marking/unmarking", "initial full marking" and "hierarchical marking/unmarking".

2. The factors $\prod_{u \in \{1..N\}} Pr(\mathbf{L}^u \mid \mathbf{L}^{u-1}, \mathbf{X}_{\text{aux}})$ correspond to transitions that may occur in the model. The respective rules are "probabilistic transition choice", "primitive target marking", "composite target marking", "full target marking", "hierarchical marking/unmarking" and "transition guard consistency".

3. The factors $\prod_{u \in \{0..t\}} Pr(\mathbf{O}_{\text{BN}}^u \mid \mathbf{L}^u, \mathbf{X}_{\text{aux}})$ and $\prod_{u \in \{t+1..N\}} Pr(\mathbf{O}_{\text{BN}}^u \mid \mathbf{L}^u, \mathbf{X}_{\text{aux}})$ correspond to observations. The respective rule is "marked location behavior".

Conditional independencies could be violated if connections between BLN fragments were added, resulting in wrong connections between random variables in a ground BN. Since we conjecture (with 1) the translation rules to correctly describe the PHCA structure, which in turn determines the mentioned independencies, we can conclude that no connections are added that violate these independencies. □

Now we develop the proof of theorem 1 following our proof idea.

Proof. With lemma 2 we have

$$\sum_{\mathbf{O}_{\mathrm{BN}}^{t+1:N}} Pr(\mathbf{l}^{t_0}, \ldots, \mathbf{l}^{t_N}, \mathbf{O}_{\mathrm{BN}}^{0:t} = \mathbf{o}^{0:t}, \mathbf{O}_{\mathrm{BN}}^{t+1:N}) \quad = \\ Pr(\mathbf{l}^{t_0}) \cdot \prod_{u \in \{1..N\}} Pr(\mathbf{l}^u \,|\, \mathbf{l}^{u-1}) \cdot \prod_{u \in \{0..t\}} Pr(\mathbf{O}_{\mathrm{BN}}^u \,|\, \mathbf{l}^u)$$

We map these factors to the factors of equation 4.1. For brevity, we leave out the conditioning on the auxiliary variables $\mathbf{X}_{\mathrm{aux}}$, which doesn't affect the calculations. Observe that $\prod_{u \in \{0..t\}} Pr(\mathbf{O}_{\mathrm{BN}}^u \,|\, \mathbf{l}^u)$ corresponds to $\prod_{u \in \{0..t\}} Pr(\mathbf{O}^u = \mathbf{o}^u \,|\, m^u)$ and $Pr(\mathbf{l}^{t_0})$ to $P_\Xi(m^0)$, because markings m^t correspond to assignments $\mathbf{L}^t = \mathbf{l}^t$, and thus to BN value vectors $\mathbf{l}^t$. This allows to replace these factors, yielding

$$P_\Xi(m^0) \cdot \prod_{u \in \{1..N\}} Pr(\mathbf{l}^u \,|\, \mathbf{l}^{u-1}) \cdot \prod_{u \in \{0..t\}} Pr(\mathbf{O}^u = \mathbf{o}^u \,|\, m^u).$$

The remaining factors $Pr(\mathbf{l}^u \,|\, \mathbf{l}^{u-1})$ are a product of probabilities given in the CPTs of *chooseTrans*(), *locMarked*() and *guardIsConsistent*(), see figure 5.7 for an example. We can ignore the factors that stem from the conditional probability tables of *locMarked*() and *guardIsConsistent*(), since they are deterministic. If they evaluate to 1, they don't influence the result. If they evaluate to 0, then it means the associated full assignment is structurally inconsistent and thus doesn't contribute to the distribution. Considering this, we have

$$Pr(\mathbf{l}^u \,|\, \mathbf{l}^{u-1}) = \prod_{l_1 \to \tau \to l_2, \mathbf{l}^{u-1} \vdash l_1 \wedge \mathbf{l}^u \vdash l_2} Pr(T_{l_1}^{u-1} = \tau \,|\, L_{l_1}^{u-1} = \mathsf{true}, \mathbf{G}^t).$$

The notation $\mathbf{l}^u \vdash l$ means that in assignment $\mathbf{l}^u$ location l is marked, i.e. $L_l^u = \mathsf{true}$. $l_1 \to \tau \to l_2$ is a triple of a location l_1 that has outgoing transition τ, which leads to location l_2. For $Pr(T_{l_1}^{u-1} = \tau \,|\, L_{l_1}^{u-1} = \mathsf{true}, \mathbf{G}^t)$ we know that it is $P_T[l_1](\tau)$ if condition a) holds (see equation 5.4 of the probabilistic transition choice rule), i.e. if the source location of τ is marked the guard of τ is consistent. We can ignore the full assignments where the probability function evaluates to 1 (has no influence) or to 0 (structurally inconsistent). Therefore, we have $Pr(\mathbf{l}^u \,|\, \mathbf{l}^{u-1}) = \prod_{l_1 \to \tau \to l_2, \mathbf{l}^{u-1} \vdash l_1 \wedge \mathbf{l}^u \vdash l_2} P_T[l_1](\tau)$. If we additionally form the product ranging over all time points, we get

$$\prod_{u \in \{1..N\}} \prod_{l_1 \to \tau \to l_2, \mathbf{l}^{u-1} \vdash l_1 \wedge \mathbf{l}^u \vdash l_2} P_T[l_1](\tau),$$

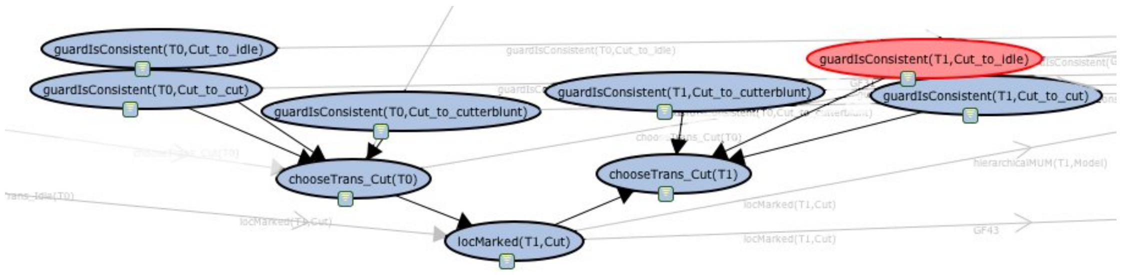

guardIsConsistent(T0,Cut_to_idle)
guardIsConsistent(T0,Cut_to_cut)
guardIsConsistent(T0,Cut_to_cutterblunt)
guardIsConsistent(T1,Cut_to_cutterblunt)
guardIsConsistent(T1,Cut_to_idle)
guardIsConsistent(T1,Cut_to_cut)
chooseTrans_Cut(T0)
chooseTrans_Cut(T1)
locMarked(T1,Cut)

Figure 5.7.: Excerpt of the Bayesian net retrieved from translating and grounding the machining station PHCA (adapted for one time step) shown in figure 5.6b. The excerpt illustrates, using the transition from location Cut to itself, how the nodes that correspond to the predicates *chooseTrans*(), *locMarked*() and *guardIsConsistent*() are connected. The image has been edited to highlight the relevant arcs.

which corresponds to factor $\prod_{\tau \in \mathcal{T}[\theta]} Pr(\tau)$ in the PHCA trajectory probability. The multi-set $\mathcal{T}[\theta]$ is then given by $\mathcal{T}[\theta] = \biguplus_{u \in \{1..N\}} \{\tau \mid \forall l_1, l_2 \in \Sigma.\ l_1 \to \tau \to l_2 \wedge \mathbf{l}^{u-1} \vdash l_1 \wedge \mathbf{l}^{u} \vdash l_2\}$[7]. We can now replace the remaining factor $\prod_{u \in \{1..N\}} Pr(\mathbf{l}^u \mid \mathbf{l}^{u-1})$, which gives us

$$P_{\Xi}(m^0) \prod_{\tau \in \mathcal{T}[\theta]} Pr(\tau) \prod_{u \in \{0..t\}} Pr(\mathbf{O}^u = \mathbf{o}^u \mid m^u).$$

We can reorder the factors to

$$P_{\Xi}(m^0) \prod_{u \in \{0..t\}} Pr(\mathbf{O}^u = \mathbf{o}^u \mid m^u) \prod_{\tau \in \mathcal{T}[\theta]} Pr(\tau),$$

which is the distribution of the original PHCA as given in equation 4.1. □

Conjecture 1 indicates a missing link for a rigorous theoretical correctness guarantee. The original rules for the COP translation given in [129] have not been shown in a rigorous way to correctly encode PHCA marking evolutions. In this work, we assume these rules to be correct and thus that our own translation rules correctly describe the evolution of PHCA markings.

[7]The operator $\biguplus$ is a multi-set join. If two (multi-) sets being joined with this operator contain the same element, the joint set is then a multi-set that contains the said element twice.

5.3.3. Computing Success Probabilities and Model-Based Diagnoses as Marginals and Most Probable A Posteriori Hypotheses

Translating and adapting a given PHCA model, $\mathcal{E}_{\mathcal{P}}(\Upsilon_{\text{BLN}}(M_{\text{PHCA}}))$, produces a BLN that can be used as input for probabilistic reasoning approaches for plan assessment. In this work, we ground BLNs to BNs using the BLN tools that come with the BLN framework described in [95]. We then feed the resulting BN into the state-of-the-art inference tool Ace 2.0[8], which computes marginal probabilities for all random variables in the BN. As described in section 3.5.2, Ace draws its strength from representing the BN as arithmetic circuit, generated in an offline step.

An arithmetic circuit is an efficient tree representation of the probability distribution over all full assignments in form of a sum, or polynomial, of the single joint probabilities. The arithmetic circuit may reflect local model structure in the form of deterministic conditional probability tables and global tree structure if BNs can be transformed in a join tree with low tree width, i.e. only few variables per tree node. Ace 2.0 is implemented in Java and C++.

Ace cannot solve MAP (i.e. problem 4) out of the box, needed to compute most probable diagnoses. Therefore we focus on computing success probabilities with Ace in this work. However, it is fair to assume that the recently developed AceMAP algorithm for this problem [37] will find its way into this toolbox sooner or later.

Now let's look in more detail at how Ace computes success probabilities, for example the success probability for maze Maze0 in our main example described in section 2.1. Marginals, and thus in particular success probabilities $Pr(\mathcal{G}_i \,|\, \mathbf{o}^{0:t})$ are computed as differentials of the above mentioned polynomial. Remember the small example from section 3.5.2:

$$Pr(A = a \,|\, b) = \frac{1}{\lambda_a\theta_a\theta_{b\,|\,a} + \lambda_{\overline{a}}\theta_{\overline{a}}\theta_{b\,|\,\overline{a}}} \frac{\partial f}{\partial \lambda_a}(b) = \frac{\theta_a\theta_{b\,|\,a}}{\lambda_a\theta_a\theta_{b\,|\,a} + \lambda_{\overline{a}}\theta_{\overline{a}}\theta_{b\,|\,\overline{a}}}.$$

As explained in this section, the products $\lambda_a\lambda_b\theta_a\theta_{b\,|\,a}$, $\lambda_a\lambda_{\overline{b}}\theta_a\theta_{\overline{b}\,|\,a}$, etc. are the probabilities of full assignments, which in case of plan assessment represent possible system trajectories. Multiple full assignments might of course represent the same trajectory, but the necessary summing is done automatically by Ace.

Remember that goals in plan assessment are represented as a tuple (l, t) of a PHCA location l that should be marked at time t. In our example, the location is $l = \mathsf{Maze0.Ok}$ of the maze PHCA, and the time is $t = 3$. In BLN terms, the goal $(\mathsf{Maze0.Ok}, 3)$ translates

[8]`http://reasoning.cs.ucla.edu/ace/` (03.2011)

to *locMarked*$(3, \mathsf{Maze0.Ok}) = \mathsf{true}$, and this in turn to the assignment $L^3_{\mathsf{Maze0.Ok}} = \mathsf{true}$ for the boolean random variable encoding the marking in the BN resulting from grounding. This means, to compute the success probability for the maze, we must compute $Pr(L^3_{\mathsf{Maze0.Ok}} = \mathsf{true} \,|\, \mathbf{o}^{0:t})$ for the BN. In terms of the polynomials encoded in the Ace arithmetic circuits, this roughly looks like this:

$$Pr(L^3_{\mathsf{Maze0.Ok}} = \mathsf{true} \,|\, \mathbf{o}^{0:t}) =$$

$$\frac{1}{\ldots + \lambda_{O^0_{ForceAlarm} = \mathsf{nominal}} \cdot \ldots \cdot P_T[\mathsf{Maze0.Ok}](\tau_{\mathsf{Maze0.Ok} \rightarrow \mathsf{Maze0.Ok}}) \cdot \ldots \cdot \frac{1}{\#\mathrm{o}_l} \cdot \ldots \cdot \prod_{p_{\mathrm{det}}} p_{\mathrm{det}} + \ldots} \frac{\partial f}{\partial \lambda_{L^3_{\mathsf{Maze0.Ok}}}}(\mathbf{o}^{0:t}) \quad .$$

That is, for full assignments to BN variables we have products of the form

$$\lambda_{O^0_{ForceAlarm} = \mathsf{nominal}} \cdot \ldots \cdot P_T[\mathsf{Maze0.Ok}](\tau_{\mathsf{Maze0.Ok} \rightarrow \mathsf{Maze0.Ok}}) \cdot \ldots \cdot \prod_{p_{\mathrm{det}}} p_{\mathrm{det}} \cdot$$

The transition probabilities $P_T[l_i](\tau_j)$, such as $P_T[\mathsf{Maze0.Ok}](\tau_{\mathsf{Maze0.Ok} \rightarrow \mathsf{Maze0.Ok}})$, and factors $\frac{1}{\#\mathrm{o}_l}$ from the uniform observation model take the place of the $\theta_{b\,|\,a}$ factors. Given observations yield evidence factors such as $\lambda_{O^0_{ForceAlarm} = \mathsf{nominal}}$. The factors for the observation model result from the uniform distribution over observations allowed by the behavior of some location l. Their number is denoted as $\#\mathrm{o}_l$. Finally, the factors $\prod_{p_{\mathrm{det}}} p_{\mathrm{det}}$ result from conditional probability tables of deterministic conditions and rules.

The Ace compilation of a BN to an arithmetic circuit is considered a (potentially expensive) offline step, the computation of success probabilities the (quick) online step. Evidence can be added during this online phase, i.e. the computation can be done for different sets of observations without the need for recompilation.

Ace compilation can only be done offline if the execution adaptation is done offline. The reason is that execution adaptation adds component links, and this operation is not defined for arithmetic circuits. As long as we perform computations over the complete plan length $N_{\mathcal{P}}$, this is not a problem. However, in real-world applications the size of $\mathcal{P}$ can quickly become too large to allow reasoning over the complete time horizon, requiring receding horizon schemes that translate the model for a fixed number of N time steps and then iteratively move it along the time line to cover the complete $\mathcal{P}$. In section 7.1 we introduce such a scheme for the model-based diagnosis approach. Within such a scheme, the current versions of $\mathcal{E}_{\mathcal{P}}$ must be applied online for every iteration step. If we were to combine the receding horizon scheme with probabilistic reasoning using Ace, we thus would have to perform the *Ace* compilation online, too. In future work, the problem could be remedied by, for example, defining an execution adaptation function for arithmetic circuits, developing a way to add component links to it.

6. Evaluation and Comparison of the Presented Approaches

In this chapter we present experimental results for both the model-based diagnosis approach and the probabilistic reasoning approach. We first describe the implementations we built for this work. Then we explain the problem instances, describe in detail our experiments and finally conclude with a discussion of the results.

6.1. Implementation of Model-Based Diagnosis Approach

For this thesis, a prototypical library was implemented that offers the necessary methods and components to compute success probabilities $Pr(\mathcal{G}_i \,|\, \mathbf{o}^{0:t})$, following equation 5.1, and output sorted lists of trajectories for diagnosis. As a key feature the library allows to transparently call different WCSP solvers, i.e. optimization tools for weighted constraint satisfaction problems. It also provides methods to parse plans $\mathcal{P}$ and observations $\mathbf{o}^{0:t}$ and integrate them with the separately stored COPs generated from PHCA models. The WCSP for the solvers is automatically generated on the fly. As programming language we used Python. For experiments, Python scripts use this library to implement experimental runs. The k-best A* algorithm has been implemented in C within the solver Toolbar, while the k-best branch-and-bound algorithm has been implemented in C++ within the solver Toulbar2, version 0.8.

Figure 6.1 shows a schematic of the plan assessment component (shown in figure 4.5 depicting a potential architecture for an AI controller) being implemented with the elements provided by our library. Most prominent is the estimator (sub-)component, which is worth looking at in detail. This component reads streams of observations and plan steps and outputs the mentioned sorted list of trajectories. The streams in turn are generated by the depicted stream component, which reads the data from files. Internally, the estimator component on the fly modifies the COP to integrate observations and plan steps. This implements the execution adaption with $\mathcal{E}_{\mathcal{P}}$. It then converts the COP to a WCSP and sends it to an external solver, for example Toulbar2. This is done via a generic interface in form of a Python class representing generic WCSP solvers. The solver returns

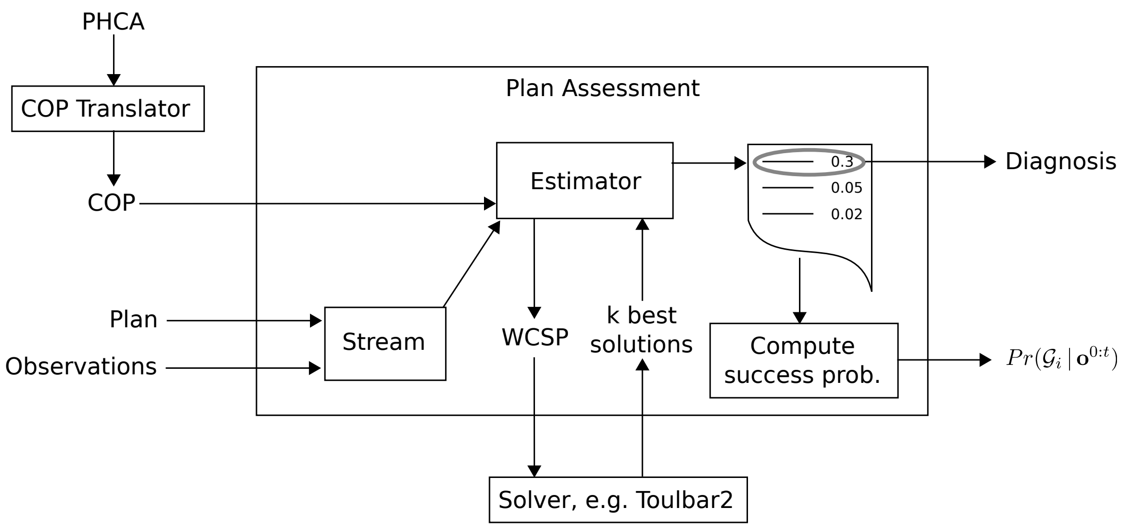


Figure 6.1.: Schematic of a plan assessment component constructed from the elements of our prototypical library. Inputs to the component are the plan $\mathcal{P}$ and observations $\mathbf{o}^{0:t}$, outputs are the diagnosis and the success probabilities.

a list of k best solutions, which the estimator then maps to the list of trajectories. Other methods and functions use this list to compute the success probabilities $Pr(\mathcal{G}_i \mid \mathbf{o}^{0:t})$ and, in case a receding horizon scheme is used, to compute the start distribution for the next time window. This approach is explained in detail in section 7.1. The plan assessment component can simply pick the top element of said list as diagnosis, and output it together with the computed success probabilities. Figure 6.2 shows the Python classes realizing the estimator component, the interface for WCSP solvers and the streaming component that reads observations and plan steps and provides them to the estimator component. Instructions about how to use our implementation can be found in the appendix in A.2.

6.2. Implementation of Probabilistic Reasoning Approach

For the probabilistic reasoning approach, we implemented a component to parse and translate a PHCA to a BLN. That component is part of our library mentioned in the previous section. Success probabilities are computed by first translating a PHCA model to a BLN using this component, and then feeding the BLN to the BLN toolbox developed by Dominik Jain[1] (the relevant publications are [95, 96]). This toolbox reads the BLN and

[1]`http://www9-old.cs.tum.edu/people/jain/dl.php?get=probcog` (06.2011)

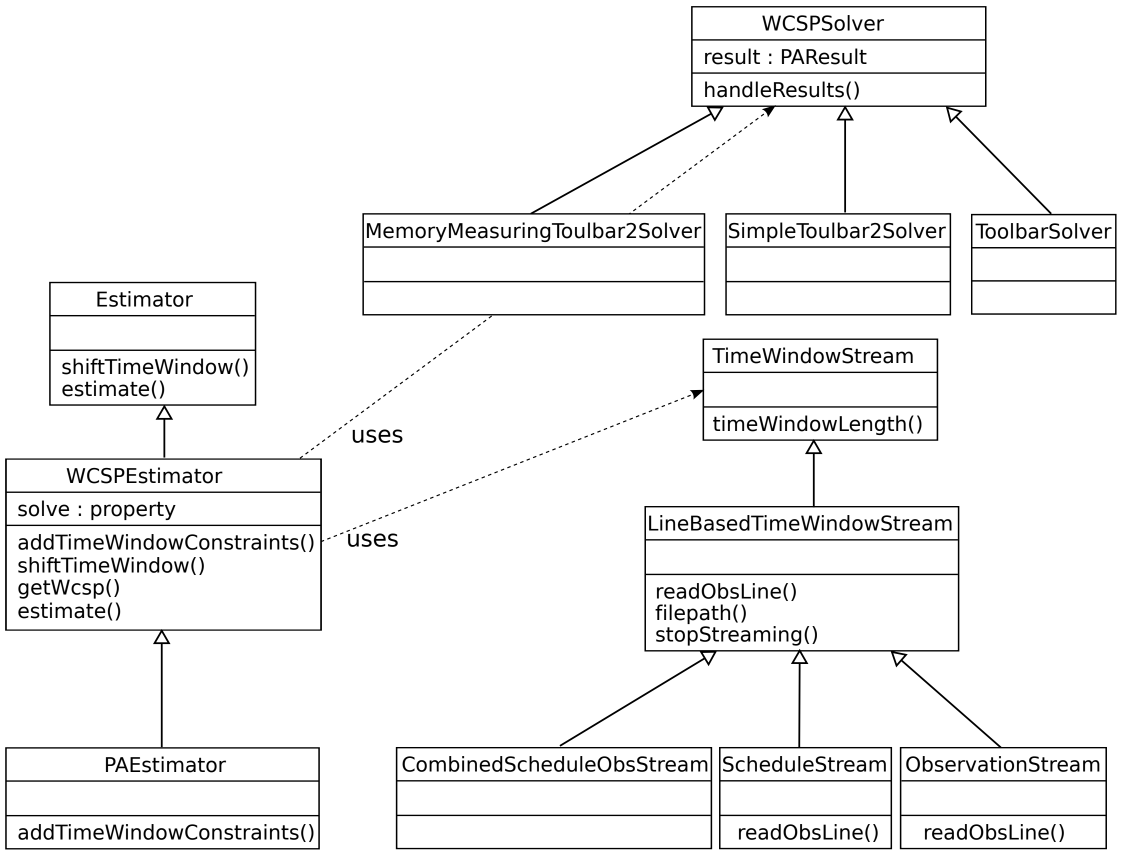


Figure 6.2.: UML diagram depicting the key classes of our plan assessment library.

transparently uses tools such as Ace to compute success probabilities. While we didn't prototype the necessary elements for a plan assessment component for this approach, future implementations could exploit the generality of our library. One could derive a specialized estimator class that encapsulates the BLN related computations, especially calling tools such as Ace, and integrate it into the schema shown in the previous section in figure 6.1.

For the translation component, we did not use the PHCA parser that comes with the COP translation developed by the authors of [129]. That would have required to implement the BLN translation within their (C++) framework, which would have made the integration with our library much harder. By writing our own parser directly as additional Python module, it is easier to connect to the Java-based BLN framework via Jython[2] and to formulate experiment scripts in Python. Existing Python packages ease this task.

[2] `http://www.jython.org/` (10.2011)

We built the parser by "implementing" the grammar of the PHCA description language. We used the Python package "pyparsing" by Paul McGuire[3], which allows to construct the parser by writing down the grammar rules in a simple Python-based syntax. A separate Python class, tailored to the translation to BLNs, uses this parser to retrieve an object that encapsulates the parsed PHCA description as abstract syntax tree. It provides methods to generate the different pieces of BLN code that, put together, result in declarations $\mathcal{D}$, fragments $\mathcal{F}$ and logical formulas $\mathcal{L}$ as well as the entries for the knowledge base DB. A separate Python script uses this component to read a PHCA model along with a plan and observations from model/plan/observation files and to create and output the resulting BLN $\mathcal{B}$ and DB.

The BLN $\mathcal{B}$ and knowledge base DB are read by the BLN toolbox. Since we want to compute results with the external solver Ace, the toolbox grounds $\mathcal{B}$ and DB to a Bayesian net and separate evidence and creates two files in the format that Ace accepts. The evidence file contains the observations $\mathbf{o}^{0:t}$. Then, the Ace compilation is applied to the Bayesian net (without the evidence) to create an arithmetic circuit. Finally, Ace is called with the arithmetic circuit and the evidence to obtain $Pr(\mathcal{G}_i \mid \mathbf{o}^{0:t})$. The toolbox then reads the results from Ace's output and presents them in (more) readable format.

This implementation was meant to demonstrate the feasibility of the probabilistic reasoning approach and is therefore rather crude. For example, it doesn't provide a separate translation path from observations in our own format to evidence (in DB) and then to the Ace format. However, this only requires to implement said path, such that we don't have to rerun the translation if only the observations change.

6.3. Problem Instances

We used eight different instances for our experiments. Seven of them use models of factory plants, comprising assembly and machining stations as well as toy mazes and toy robot arms as products. One instance uses a satellite model for a diagnosis task taken from [129], which we extended towards a plan assessment scenario. Of the factory models, one part comprises variations of the model used in the example scenario described in section 2.1. The other part follows a second scenario, taken from our work in [121]. We denote the instances with the abbreviations fm$x$, sm and em, where "fm" stands for "factory model", "sm" for "satellite model" and "em" for "example model". x is the number of the factory model. In some cases we add something to further differentiate problem instances.

[3] `http://pyparsing.wikispaces.com/` (10.2011)

Table 6.1.: The size of PHCAs, COP and BN translations.

problem instance	N	phca size	COP		BN
			# var	# con	# nodes
fm1 [121]	6	11 / 6 / 27	643	670	1106
fm2	9	15 / 8 / 33	1202	1251	2122
fm3	9	17 / 8 / 33	1305	1311	2292
fm2(long $\mathcal{P}$)	19	15 / 8 / 33	2482	2601	4444
fm3(long $\mathcal{P}$), no $\mathbf{o}$	33	18 / 8 / 35	4748	4892	8878
fm3(long $\mathcal{P}$)	33	18 / 8 / 35	4748	4892	8878
sm [129]	8	8 / 4 / 22	640	661	1080
em	9	18 / 8 / 38	1394	1418	2422

In the mentioned second scenario, the cutter can go blunt during operation and is then more likely to break. Instead of a force sensor, we have a vibration sensor that picks up increased vibrations of blunt cutters. However, the sensor signals are ambiguous: some components generate random vibrations, and thus not every vibration means that a component is faulty. We assume that, with some probability, vibrations in the assembly can trigger signals in sensors of machining stations. Here, a vibration is detected at $t_{\mathsf{vibration}}$, while the machining station is cutting a part for the robot arm and the maze is being assembled (see figure 7.2). Is the vibration an indicator for a blunt cutter, and how does this possibility affect the plans? This scenario comprises one machining and one assembly station, with one maze and one robot being scheduled for manufacturing.

All factory models contain one assembly station, one or two machining stations and one, two or three product models. According to the two scenarios, sub-PHCA models for machining and assembly stations come in two different flavors:

1. In the first scenario with the force alarm, like in the example in section 2.1, machining stations may break products if their cutter breaks, and assembly stations have a force sensor that causes an alarm if either a worked maze product has improper holes (due to the broken cutter), or if the station is misaligned. The latter fault is modeled as a composite location within the assembly station model, see figure 3.6.

2. In the second scenario with the vibration sensors, the cutter going blunt is modeled with an extra composite location that recreates the nominal behavior of the station, only with a now blunt cutter (see figure 3.6). A composite location in the assembly station models occasional vibrations. An additional helper location introduces

a behavior constraint that links the vibrations caused by the stations with the observation variable (shown in figure 7.3).

The instances with factory models comprise three basic instances, fm1, fm2 and fm3, two extended instances fm2(long $\mathcal{P}$) and fm3(long $\mathcal{P}$), and an instance em based on the example factory model used in section 2.1. The instances em and fm3 both belong to the first scenario. They both use models with one assembly station, two machining stations and three product models. Instances fm1 and fm2 belong to the second scenario. Their models contain one assembly station and one machining station. Instance fm1 has one, fm2 two products. fm1 is a simplified prototype scenario for which plan assessment was first tested in [121]. It is simplified in that the product is represented as a single binary variable. Specifically, no explicit product flow is modeled, that is there are no product-component links. Also, it uses a very simple plan $\mathcal{P} = ((\mathsf{cut}, 0), (\mathsf{assemble}, 1), (\mathsf{cut}, 2), (\mathsf{cut}, 3), (\mathsf{cut}, 4), (\mathsf{cut}, 5))$ with six operation steps.

Instances fm2(long $\mathcal{P}$) and fm3(long $\mathcal{P}$) extend fm2 and fm3, respectively. The most important difference is in their plans, which have a higher resolution in time, i.e. more time points. These were introduced to test the approaches on bigger plans. Also, the model of fm3(long $\mathcal{P}$) is fit to deal with robot products, while fm3 only handles maze products. To deal with a robot product we added another location to the assembly sub-model.

We added another instance with the same model and plan as fm3(long $\mathcal{P}$), however leaving out all observations. This instance illustrates that, as a consequence of the design choice for the constraint optimization approach, errors in diagnosis can occur. Finally, the diagnosis instance captures a scenario that simulates diagnosing hardware or software faults in a satellite camera module [129]. We added this instance to our set of instances for comparison. We defined a plan assessment scenario based on the diagnosis scenario described in [129]. The plan assessment scenario defines a goal $(\mathsf{ProcessingImage}, 5)$ within a simplified plan $\mathcal{P}$ with 8 time steps, which operates the components of the camera module.

Table 6.1 lists all the problem instances along with the PHCA size (number of primitive locations, composite locations and transitions), the number of variables and constraints in the generated constraint net and the number of nodes (random variables) in the BN obtained from translating the PHCA to a BLN and then grounding this BLN.

To illustrate what is used as input and what results as output for a single instance, we take another look at our main example and its factory model instance em. Figure 6.3 shows again the plan $\mathcal{P}$ for this instance. Since it is an example from the manufacturing domain, $\mathcal{P}$ is a schedule. Figure 6.4 shows the PHCA M_{PHCA} for this instance, and in

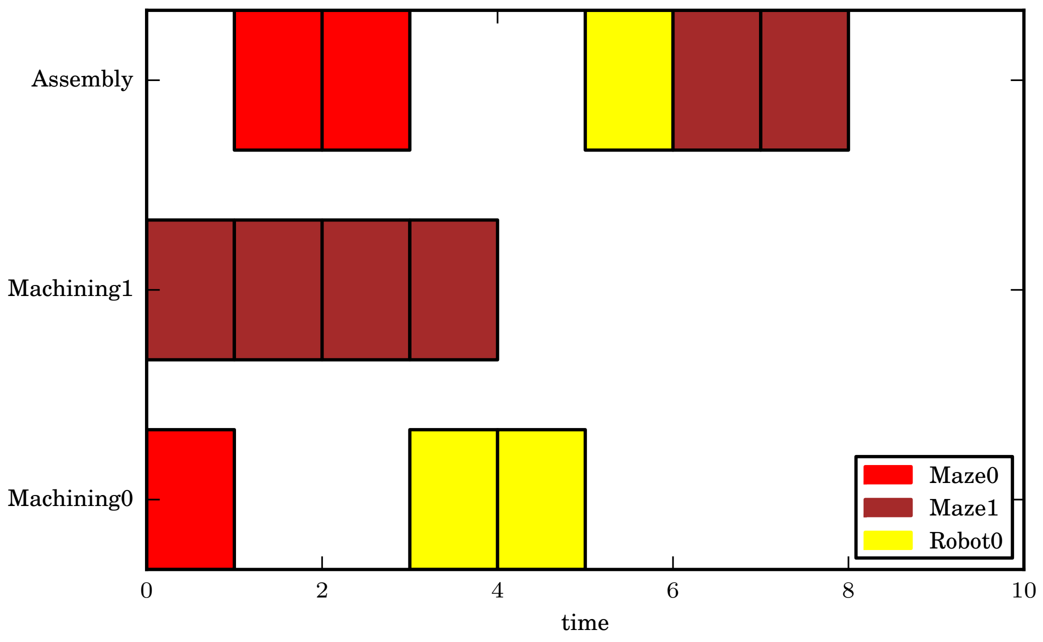


Figure 6.3.: Plan $\mathcal{P}$ for problem instance em, which represents our factory example described in section 2.1.3. It is shown which product is being worked when by which station. For example, Maze0 (light red) is worked by Machining0 from $t = 0$ to $t = 1$, then by Assembly from $t = 1$ to $t = 3$. Not shown are the commands that drive the execution of the factory stations.

listing 6.1 the available observations are shown. Together, these elements are the input to both our solution approaches. We refer to the appendix for the actual encoded input data (see appendix B).

The output in general consists of a number of pairs of probability values and goals as well as a sorted list of marking sequences. The former are the success probabilities for the specified goals and the latter the most probable system trajectories for diagnosis. At this point remember that our implementation of the model-based diagnosis approach can compute both of these outputs, whereas our implementation of the probabilistic reasoning approach currently only computes success probabilities. Probabilities for the em instance can be found in tables 6.2 and 6.3, the most probable trajectory is shown in figure 6.5. Note that, since observations are only available up to time point 3, the behavior of the system is being predicted beyond that time point.

```
PFORCE__0=NONE
PFORCE__1=NONE
PFORCE__2=NONE
PFORCE__3=HIGH
```

Listing 6.1: Observations for instance em, encoded as assignments to an observation variable for the first four time points.

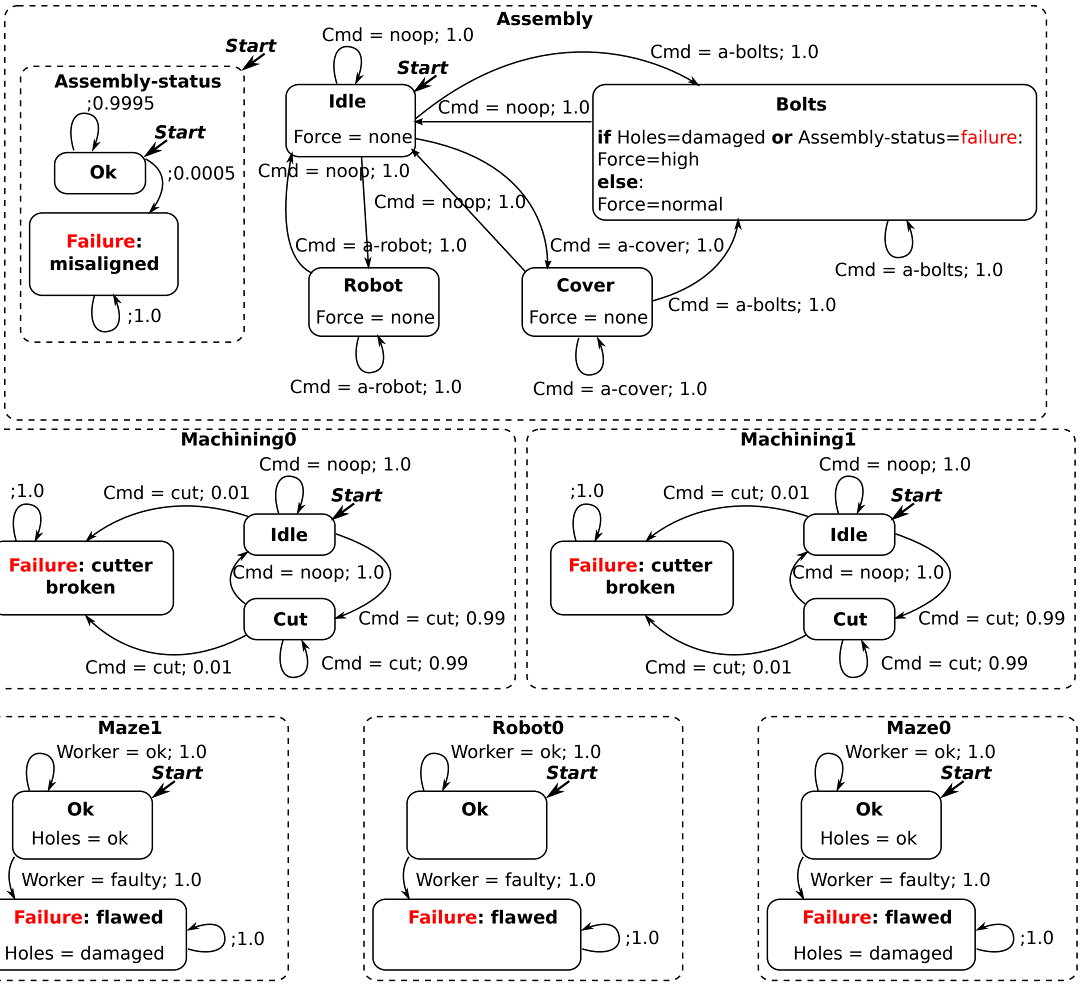


Figure 6.4.: The complete PHCA model of problem instance em, the example instance from section 2.1.3. This model, encoded in the PHCA description language, was used to produce the results for em seen in tables 6.2 and 6.3. To keep the graphic in a comprehensible format we changed variable names to a more readable form and also left out some details, such as the helper variables needed to create component links. The code for this model is shown in the appendix, see B.

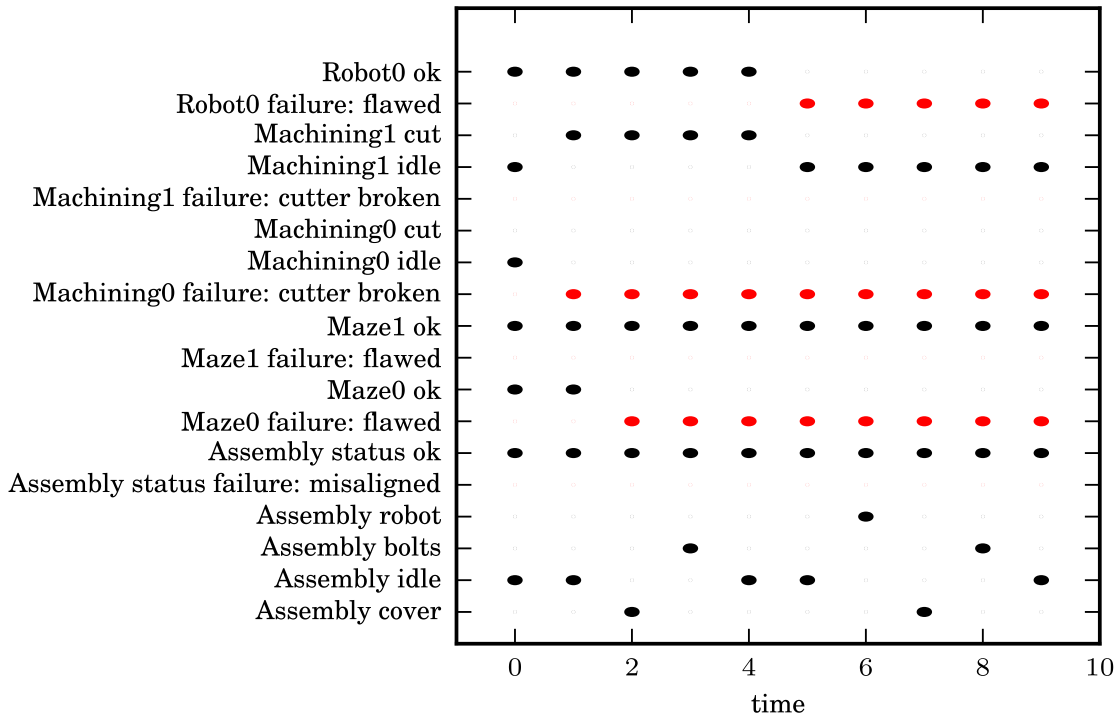


Figure 6.5.: Most probable trajectory of instance em, given plan $\mathcal{P}$ (figure 6.3), observations (figure 6.1) and the model M_{PHCA} (figure 6.4). The graph shows the marking of locations (y-axis) for each time point (x-axis). A marked location is represented with a black ellipse if it's a normal location and with a red ellipse if it's a fault location.

Table 6.2.: Results for computing most probable diagnoses and success probabilities using the model-based diagnosis approach with constraint solver Toulbar2. Results on the left were computed using the constraint net adapted for diagnosis, results on the right adapted for success probabilities. Results were computed exactly, that is all trajectories with probability > 0 where enumerated.

instances	adapted for diagnosis		adapted for success prob.	
	$Pr(\mathcal{G}_i \mid \mathrm{o}^{0:t})$	error	$Pr(\mathcal{G}_i \mid \mathrm{o}^{0:t})$	diagnosis error
fm1	PRODUCT: 0.54	0.1231	PRODUCT: 0.42	1 (21)
fm2	MAZE0: 1.00 ROBOT: 0.54	0.1231	MAZE0: 1.00 ROBOT: 0.42	1 (84)
fm3	MAZE2: 0.00 MAZE0: 0.00 MAZE1: 0.36	0.0000	MAZE2: 0.00 MAZE0: 0.00 MAZE1: 0.36	1 (110)
fm2(long $\mathcal{P}$)	MAZE0: 1.00 ROBOT: 0.13	0.0472	MAZE0: 1.00 ROBOT: 0.08	1 (648)
fm3(long $\mathcal{P}$), no **o**	MAZE0: 0.59 MAZE1: 0.99 ROBOT0: 0.17	0.0000	MAZE0: 0.59 MAZE1: 0.99 ROBOT0: 0.17	64 (963)
fm3(long $\mathcal{P}$)	MAZE0: 0.00 MAZE1: 0.99 ROBOT0: 0.00	0.0000	MAZE0: 0.00 MAZE1: 0.99 ROBOT0: 0.00	1 (255)
sm	PROCESSING: 0.06	0.0036	PROCESSING: 0.06	1 (64)
em	MAZE0: 0.00 MAZE1: 0.83 ROBOT0: 0.00	0.0000	MAZE0: 0.00 MAZE1: 0.83 ROBOT0: 0.00	1 (190)

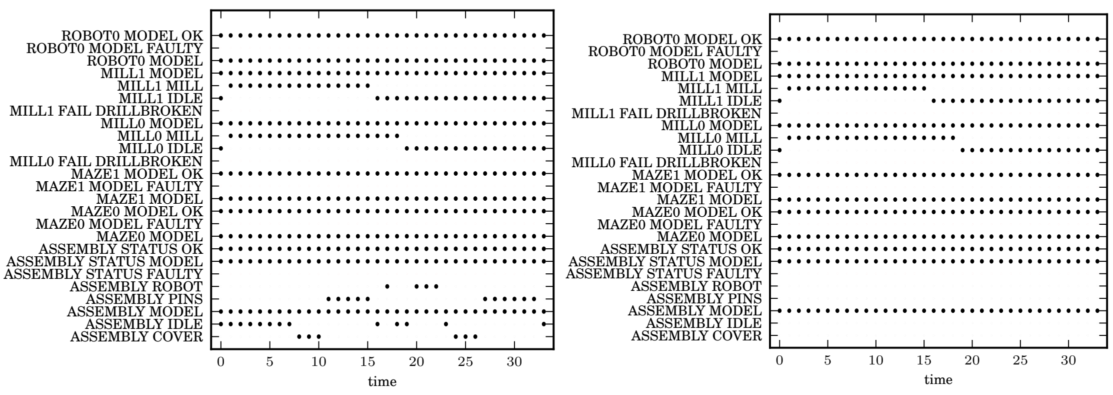


Figure 6.6.: The design choice to adapt the model-based diagnosis approach for correct success probabilities leads to an incorrect diagnosis (right side) for problem instance fm3(long $\mathcal{P}$), where we left out all observations. The diagnosis is then a highly improbable trajectory wherein the assembly station is never being executed: it doesn't contain any markings for the primitive locations ending with ...ROBOT, ...PINS,...IDLE,...COVER (lower part of the plots). The correct diagnosis, where assembly execution follows the given commands, is shown on the left side. It was generated by this approach adapted for correct diagnosis.

6.4. Experiments and Results

In this sub-section we describe our experiments and what is shown in the result plots and tables. The next section then discusses the results.

In general, measurements of resource consumption (runtime and memory) where obtained within a virtual machine with 2GB of memory, which used a single core of an Intel core2duo (2.53 Ghz) and ran Ubuntu Linux as operating system (the host system was Mac OS X). An exception are the (older) results in table 6.4, which were obtained on an earlier Linux computer with an Intel core2duo 2.2 Ghz CPU with 2 GB RAM (not virtualized). Unless stated otherwise, we used default options for the involved solvers. Memory measurements were done for Toolbar and Toulbar2 using Valgrind[4], and for Ace using the measurements it offered, namely the "allocated megabytes" (which is being output on the console). For runtime, we measured the sum of the time to create a heuristic and the search time for Toolbar, the search time for Toulbar2 and the inference time for Ace. All these times are measured and output by the tools themselves. For the sake of completeness note that, in the following, the parameter k will refer to the number

[4]`http://valgrind.org/` (07.2011)

of enumerated COP solutions, not trajectories (unless stated otherwise). However, the difference is not relevant for the results related to k, as it involves only a small factor.

In table 6.2 results are shown from applying the constraint optimization approach to the plan assessment instances, using the solver Toulbar2. The left side shows results for the case that the approach is adapted for diagnosis. For each instance, success probabilities for all goals are shown, along with the maximum error among them. As reference for the error results from the probabilistic reasoning approach were used, except for fm2(long $\mathcal{P}$), which could not be solved by this approach. Here, the results from the right side served as reference. The right side shows results in case the approach is adapted to computing success probabilities. Also shown on this side is the error of diagnosis, which has been determined as follows: We computed the position of the most probable diagnosis, obtained with the diagnosis-adapted approach, among solutions generated with the approach adapted for success probabilities. The further down the diagnosis appears in the list (the bigger the number), the worse the error. If the diagnosis appears first (1), no error occurred. In parentheses the number of all COP solutions is shown. The additional instance fm3(long $\mathcal{P}$) where we left out all observations (termed “no $\mathbf{o}$”) shows that the error can occur and can be quite significant. Figure 6.6 shows the diagnoses for this instance for both adaptations of the model-based approach.

Table 6.3 shows success probability results obtained with the probabilistic reasoning approach, using the solver Ace. By default, we used the publicly available version 2.0. The three bigger instances with long $\mathcal{P}$ failed due to precision loss. With the not yet publicly available Ace 3.0[5], results for fm3(long $\mathcal{P}$) with and without observations could be obtained, but not for fm2(long $\mathcal{P}$) within the 2GB memory limit. Therefore, this instance is omitted from the table. We didn't apply Ace 3.0 to all instances since one requirement for this comparison was the public availability of the off-the-shelf solvers.

In table 6.4 we can see results from measuring runtime and memory consumption for the constraint optimization approach using our A* implementation in Toolbar. The results where obtained on variations of the fm1 instance. We varied the number of cut actions after $t = 2$ for this scenario, yielding different $\mathcal{P}$. The time window size N accordingly ranges from 2 to 6. We also varied the number of trajectories k (not COP solutions in this case) and the parameter i that controls the accuracy of the heuristic generated for the A* search. Higher values for this parameter result in higher cpu/memory investment to generate better search heuristics (both steps, heuristic generation and search, are performed online). The results in table 6.4 where previously published in [121].

[5]This version of Ace was provided by Mark Chavira and Adnan Darwiche.

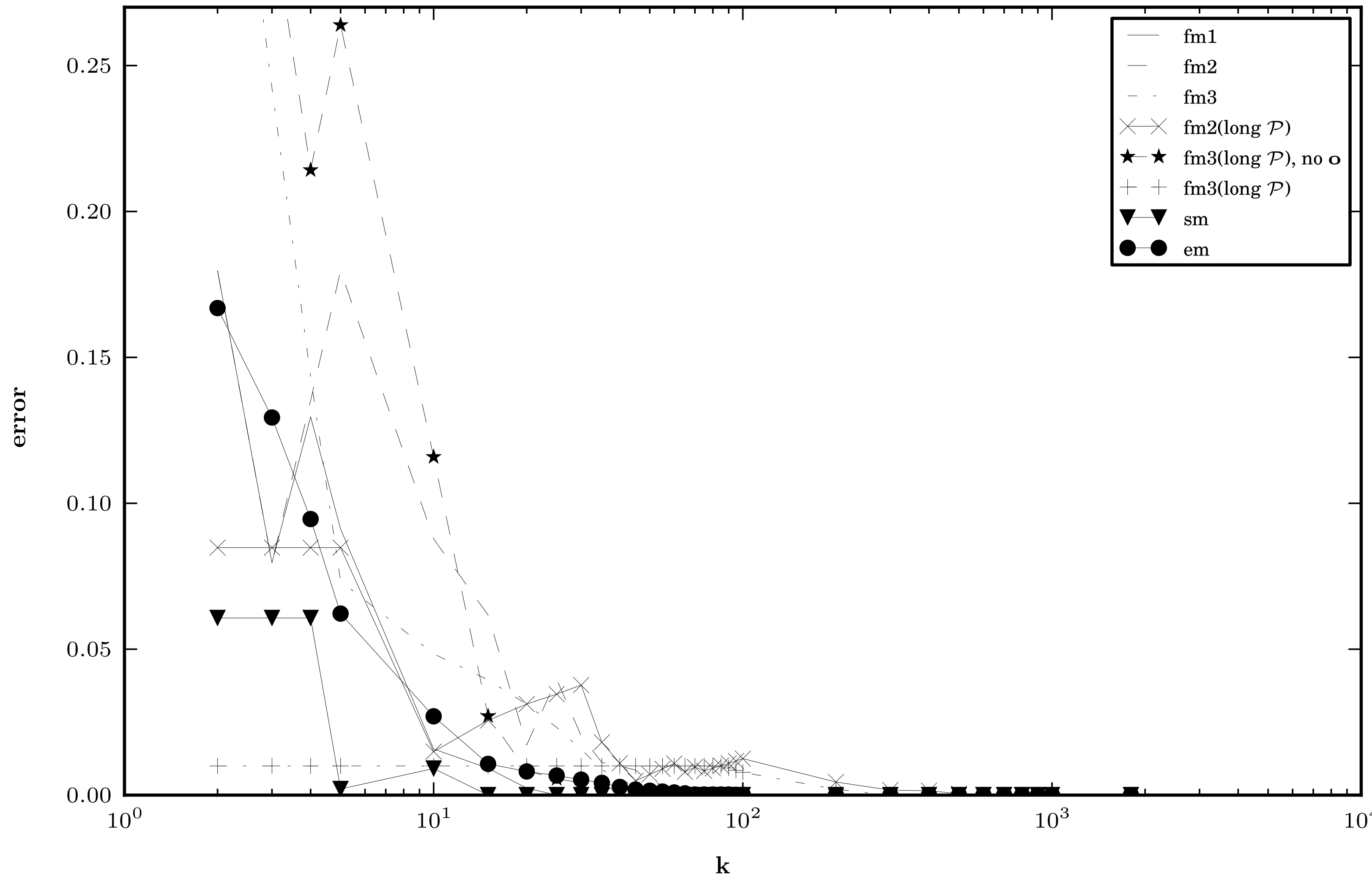


Figure 6.7.: Effect of increasing Toulbar2 approximation parameter k (X-axis, log scale) on the absolute approximation error (Y-axis) for our instances. Plotted with the help of Dominik Jain's plotting class.

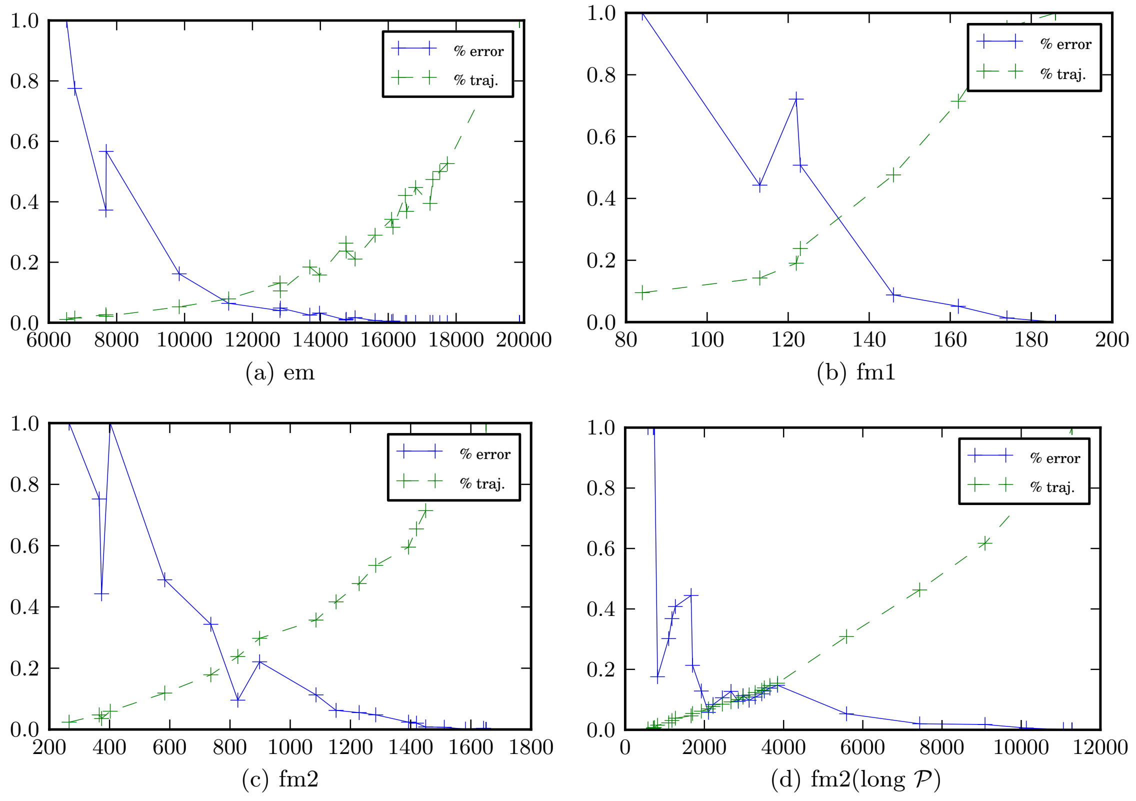
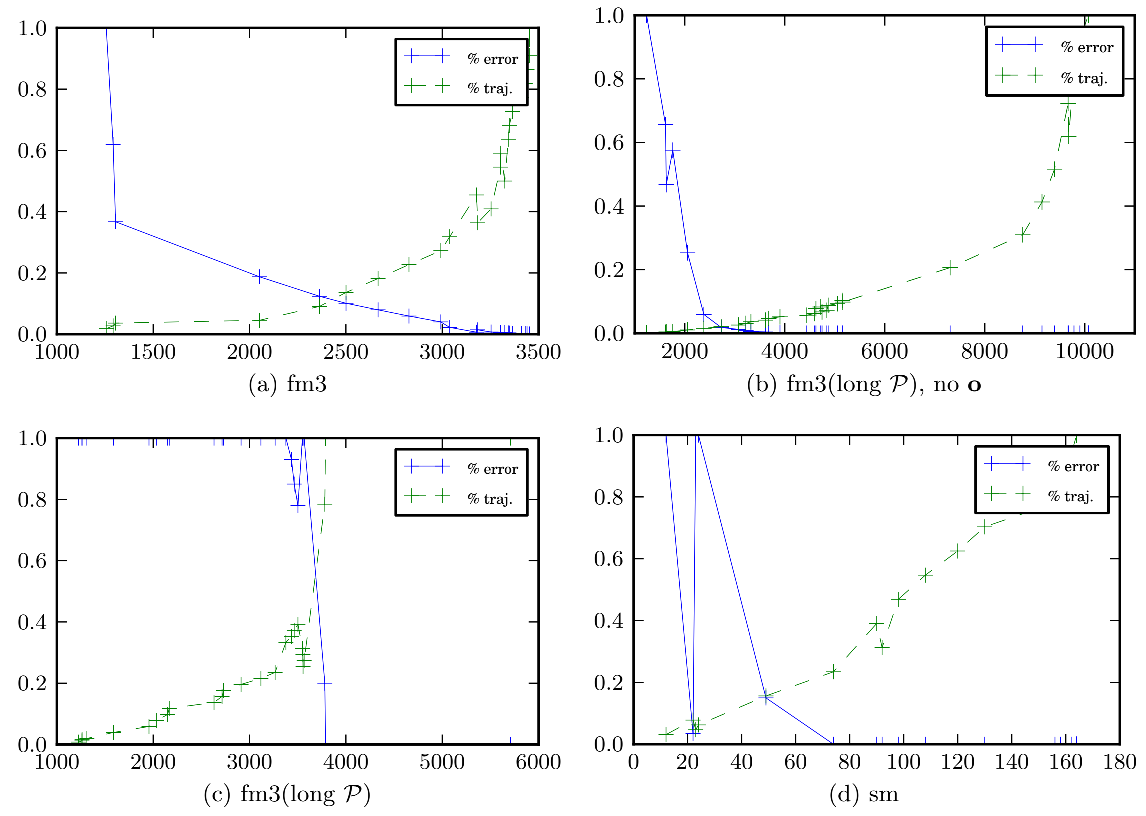


Figure 6.8.: *(Page 1)* These graphs show the effect of resource consumption on approximation error of Toulbar2. Each graph plots the following: **X-axis:** Expanded nodes. **Y-axis:** Relative error (blue, solid line) and relative number of enumerated COP solutions $\frac{k}{k_{\max}}$ (green, dashed line). Plotted with the help of Dominik Jain's plotting class.



(a) fm3 (b) fm3(long $\mathcal{P}$), no $\mathbf{o}$ (c) fm3(long $\mathcal{P}$) (d) sm

Figure 6.8: *(Page 2)*

Table 6.3.: Results for computing success probabilities with Ace 2.0 , except for those marked with *, which where computed with Ace 3.0.

instances	$Pr(\mathcal{G}_i \vert o^{0:t})$
fm1	PRODUCT: 0.42
fm2	MAZE0: 1.00 ROBOT: 0.42
fm3	MAZE2: 0.00 MAZE0: 0.00 MAZE1: 0.36
fm3 (long $\mathcal{P}$), no $\mathbf{o}$	MAZE0: 0.59* MAZE1: 0.99* ROBOT0: 0.17*
fm3 (long $\mathcal{P}$)	MAZE0: 0.00* MAZE1: 0.99* ROBOT0: 0.00*
sm	PROCESSING: 0.06
em	MAZE1: 0.83 MAZE0: 0.00 ROBOT0: 0.00

The remaining two experiments are meant to 1) explore the approximation of the constraint optimization approach and 2) to compare this approach with the probabilistic reasoning approach. In these two experiments we used default options for both tools except for Ace compilation of fm2(long $\mathcal{P}$). When compiling a BN to an arithmetic circuit, Ace by default tries to automatically determine the more suited of two basic compilation algorithms, depending on the complexity of the BN. One is a variant of variable elimination [38], which in turn is similar to the junction tree algorithms described in section 3.5.2. It uses tabular representations of the involved probability factors. The other is based on converting BNs to propositional logic representations and counting full assignments in these representations, called logical model counting [46]. For fm2(long $\mathcal{P}$), manually forcing the logical model counting yielded much better results. The accordant command line parameter (Ace is a command line tool) is `-noTabular` (for Ace 2.0).

Figures 6.7 and 6.8 show results on measuring the impact of approximation on the accuracy of success probabilities computed with the constraint optimization approach. Results in figure 6.7 allow comparison of the absolute error $\epsilon_i(k) = |Pr(\mathcal{G}_i \,|\, \mathbf{o}^{0:t}) - Pr^k(\mathcal{G}_i(k) \,|\, \mathbf{o}^{0:t})|$ among all instances. Specifically, one can see at which value for k the absolute error for an instance drops and stays below a certain threshold. We see the

Table 6.4.: Runtime and memory results for computing success probabilities using the older Toolbar solver. Runtime in seconds / peak memory consumption in megabytes. (e) indicates that all feasible trajectories were computed with this configuration. (mem) indicates that A* ran out of memory (artificial cutoff at > 1 GB).

		No. times machining used in $\mathcal{P}$ (window size N, #Variables, #Constraints)				
k	i	0 (2,239,242)	1 (3,340,349)	2 (4,441,456)	3 (5,542,563)	4 (6,643,670)
1	10	< 0.1 / 1.8	0.1 / 6.8	0.1 / 19.0	(mem)	(mem)
	15	0.1 / 1.9	0.3 / 4.2	0.5 / 7.8	0.5 / 16.6	0.8 / 32.0
	20	0.1 / 1.9	0.5 / 5.2	3.7 / 20.1	6.5 / 34.5	9.5 / 50.7
2	10	< 0.1 / 2.1	0.1 / 11.9	0.2 / 38.5	(mem)	(mem)
	15	0.1 / 2.2	0.3 / 5.4	0.5 / 9.7	0.6 / 28.0	0.8 / 52.0
	20	0.1 / 2.2	0.5 / 6.4	3.7 / 21.8	6.5 / 37.2	9.5 / 55.8
3	10	< 0.1 / 2.3 (e)	0.1 / 11.9	0.2 / 40.1	(mem)	(mem)
	15	0.1 / 2.4 (e)	0.3 / 5.4	0.5 / 11.4	0.6 / 29.9	0.9 / 55.5
	20	0.1 / 2.4 (e)	0.5 / 6.4	3.7 / 23.5	6.6 / 38.3	9.5 / 57.4
4	10	(e)	0.1 / 12.5	0.2 / 40.1	(mem)	(mem)
	15	(e)	0.3 / 5.9	0.5 / 11.4	0.6 / 30.9	0.9 / 57.2
	20	(e)	0.5 / 6.9	3.7 / 23.5	6.6 / 39.3	9.5 / 59.1
5	10	(e)	0.1 / 13.1	0.2 / 40.7	(mem)	(mem)
	15	(e)	0.3 / 6.6	0.5 / 12.0	0.6 / 33.6	0.9 / 59.5
	20	(e)	0.5 / 7.6	3.7 / 24.0	6.6 / 42.8	9.5 / 63.9
10	10	(e)	0.1 / 14.0 (e)	0.2 / 43.4 (e)	(mem)	(mem)
	15	(e)	0.3 / 6.7 (e)	0.5 / 14.7 (e)	0.6 / 36.2	0.9 / 64.8
	20	(e)	0.6 / 7.7 (e)	3.8 / 26.6 (e)	6.6 / 45.8	9.6 / 68.9

absolute error being plotted against k , where $k \in \{2, \ldots, k_{\max}\}$ is plotted on a log scale. $k_{\max}$ is the number of feasible COP solutions, i.e. which have probability > 0. To show how resource consumption is related to the choice of k and the error, figure 6.8 shows graphs, for all instances, where the relative error $\tilde{\epsilon_i}(k) = \frac{\epsilon_i(k)}{\max\limits_k \epsilon_i(k)}$ and the relative number of enumerated solutions $\frac{k}{k_{\max}}$ (both on the Y-axis) are plotted against the number of search nodes that Toulbar2 needed to expand to generate the solutions. The latter is a machine independent measure of resource consumption: runtime scales linearly with the number of nodes expanded in the search tree that the branch-and-bound algorithm in Toulbar2 traverses. This figure gives an impression on how much resources are needed to force the error below a certain percentage, and which percentage of the number of feasible COP solutions has been generated until then.

In the last experiment described in this section we measured runtime and memory needed to compute exact success probabilities for all our instances with the model-based diagnosis approach and with the probabilistic reasoning approach. For the former we used Toulbar2 with no approximation as solving backend (i.e. $k = k_{\max}$), for the latter Ace. Note that Toulbar2 and Ace differed slightly ($\triangle < 0.0001$) in their exact results, most likely due to precision loss or Toulbar2 using a lower bound to cut off solutions.

Table 6.5 shows the results for both approaches. The purpose of this comparison is to find out if there is a large difference between the two tools or not. On the left side for Toulbar2, search time in seconds, memory usage in megabytes and the expanded search tree nodes are shown. The time it took to compute the success probability from the generated solutions was below 10 milliseconds and is thus negligible. For Ace on the right side, the table shows compilation + evaluation time, both in seconds, and memory usage during compilation in megabytes (the usage during evaluation is approximately the same or less, since it is the compiled arithmetic circuit that dominates memory usage in this stage). We show compilation and evaluation time summed, because currently, as argued in section 5.3.3, it might be necessary to perform the compilation step online.

Each value shown is the mean over three runs, the standard deviation is shown in parentheses next to it. As mentioned earlier, Ace had precision loss problems with the three bigger instances with long $\mathcal{P}$, (fm2(long $\mathcal{P}$), fm3(long $\mathcal{P}$) and fm3(long $\mathcal{P}$) without observations). Results for the fm3(long $\mathcal{P}$) could be obtained, but not for fm2(long $\mathcal{P}$), which is why the according table cell is marked with “ERR” (for “error”).

Table 6.5.: Measurements for the model-based diagnosis approach using Toulbar2 and the probabilistic reasoning approach using Ace. With both exact success probabilities for our instances (i.e. $k = k_{\max}$ for Toulbar2) were computed. For Toulbar2, runtime in seconds, used memory in mb and exanded search nodes are shown. For Ace, runtime in seconds (Ace compilation + evaluation time) and used memory in mb are shown. * Results obtained with Ace 3.0.

instance	Toulbar2	Ace
fm1 [121]	0.02 (0.00) / 8 (0) / 186 (0)	0.52 (0.13) + 0.08 (0.00) / 10 (1)
fm2	0.09 (0.01) / 10 (0) / 1650 (174)	1.03 (0.18) + 0.11 (0.02) / 131 (36)
fm3	0.23 (0.06) / 10 (0) / 3454 (1548)	0.43 (0.11) + 0.06 (0.01) / 5 (0)
fm2(long $\mathcal{P}$)	0.88 (0.02) / 20 (0) / 11280 (43)	1128.04 + ERR / $\approx$ 900
fm3(long $\mathcal{P}$), no **o**	2.54 (0.06) / 39 (0) / 10073 (1221)	(1.42 (0.04) + 0.25 (0.03) / 55 (0))*
fm3(long $\mathcal{P}$)	1.05 (0.04) / 26 (0) / 5709 (660)	(1.39 (0.08) + 0.23 (0.01) / 55 (0))*
sm [129]	0.04 (0.01) / 8 (0) / 164 (0)	0.95 (0.12) + 0.02 (0.00) / 188 (1)
em	0.59 (0.17) / 12 (0) / 19870 (5424)	0.59 (0.06) + 0.07 (0.01) / 15 (0)

6.5. Discussion of Results

The results in tables 6.2 and 6.3 clearly show that both approaches are capable of computing correct success probabilities for plan assessment, assuming that the PHCA models are correct. Only one of our instances caused problems for the probabilistic approach with Ace, which failed to compute results due to precision loss. For all other instances the probabilities obtained with both approaches agree down to four places after the decimal point, which we see as a strong indicator of the correctness of the two approaches.

The constraint based approach can also compute diagnoses in form of most probable trajectories. Even if we choose to adapt the approach for success probabilities it seems to work out fine in most cases. As can be seen from table 6.2 and from figure 6.6, this choice produced an incorrect diagnosis (right in figure 6.6) only for the instance where we left out all available observations, which is a rather contrived situation. The other choice, adapting for correct diagnosis, seems more problematic, causing erroneous success probabilities in half of the instances.

The results for our main example of a cognitive factory illustrate how plan assessment can support autonomous decisions. As described in the beginning in section 2.1 we computed success probabilities (table 6.2) and a diagnosis (figure 6.5):

- Robot0 and Maze0 are predicted to fail (success probability 0, table 6.2).
- Maze1 is fairly probable to succeed (success probability of .83, table 6.2).
- The diagnosis shows the station Machining0 becoming faulty at time step 1 (figure 6.5).

An AI controller that implements a decision procedure as outlined in section 4.3 can now use this information. As suggested in section 2.1, it may decide to

1. Discard Maze0 as it is already broken.
2. Continue with Maze1 as it is only at low risk.
3. Try to re-schedule Robot0 such that Machining0 is avoided.

At this point results do not warrant a definite statement about scalability. We can nevertheless make some interesting observations and try to give an estimate based on them. First, let us look at how instances scale with respect to the length of the plan $\mathcal{P}$. When we compare the runtime (from the results in table 6.5) of fm3 with fm3(long $\mathcal{P}$), we see that roughly thrice the schedule length requires roughly four times the memory and roughly five times the runtime (of fm3). This is good: apparently the solvers are indeed able to exploit the Markov property for this instance. The picture is somewhat different with instances fm2 and fm2(long $\mathcal{P}$). Here, the difference is much greater, which is maybe due to the fact that fm2 uses a more sophisticated sub-PHCA for the machining station, which models the cutter going blunt before breaking. In general, depending on problem instance, both approaches could run into trouble between 50 and 100 time steps.

Next, we consider scalability with respect to model (PHCA) size, independent of time. To this end, we compare the three instances fm2, fm3 and em, which all range over 9 steps. Again, the situation is ambiguous. Instance fm3 has one more station and one more product than fm2. When solving it with Toulbar2 (model-based diagnosis approach), this results in approximately double the runtime. Instance em also has one more station and one more product, but here this results in roughly six times the runtime and a slight increase in memory usage. With Ace (probabilistic reasoning approach) the situation is different still, as runtime in the first case actually *decreases* to roughly half the runtime. What if we try to scale up to 10 stations and 10 products?

On the one hand, if things go as nice as the step from fm2 to fm3 we end up with $2^7 \times 0.23s \approx 30s$. If the result with Ace for these instances is any indicator it could be even better. On the other hand, if things develop as hinted by the step from fm2 to em, runtime could be as bad as $6^7 \times 0.59s \approx 45h$. Since both solving backends, Toulbar2 and Ace, are capable of exploiting structure in a general fashion, it's reasonable to assume that it won't be that bad. All in all, we see that increasing model size has a larger impact than increasing the number of time steps.

It is not surprising that scalability remains an issue if we consider that computing success probabilities exactly may be an NP-hard problem. A practical alternative is to approximate. And indeed, for Toulbar2 we can see that, for our instances, significant savings in resource usage can be made if we accept a modest error. The graphs in figure 6.8 show that, often, less than half the nodes needed for the exact computation must be expanded to force the (relative) error below 10%. On most of our instances, this amounts to enumerating 40% of the feasible COP solutions. In absolute terms, figure 6.7 shows us that for most of our instances, less than 40 COP solutions are needed to achieve an absolute error of less than 0.02. In case of Toolbar, the situation is different, as increasing k hardly seems to affect resource consumption (runtime in this case, see table 6.4), especially if the mini-bucket search heuristic is strong (bigger i-values). For weaker heuristics, the influence is slightly stronger. To highlight the behavior of the error with respect to increasing k we go back once more to the results for Toulbar2 in figures

6.8 and 6.7. We can see that the error goes to a low value fairly quickly, however often it shows some erratic behavior in form of few sharp increases after it went down. Only for one instance, fm3(long $\mathcal{P}$), it takes long until the (relative) error drops. However, this is due to the fact that for this instance the absolute error is low from the start, as is obvious from figure 6.7.

We now turn to comparisons of the different solving backends. First, we consider Toolbar and Toulbar2, applied within the model-based diagnosis approach. Both seem to scale approximately equally, Toolbar however seems to scale better with time. This is probably due to the typical effect of the involved A* algorithm of trading off memory for less runtime. As can be seen in table 6.4, memory usage increases much more than runtime with increased problem size (left to right). We can more directly compare their respective performances on the instance fm1. Toulbar2 was ran on the biggest of the variations of this instance used for the experiments with Toolbar, i.e. the rightmost in table 6.4. Clearly, Toulbar2 is magnitudes faster and more memory efficient than Toolbar on this instance. Apparently the more recent combination of branch-and-bound with soft-constraint consistency is much stronger than the older A* with automatic heuristic generation based on approximate inference.

Finally, we compare the performances of Toulbar2 and Ace. We see essentially similar performance of both tools on our instances, with Toulbar2 tending to be somewhat better. It makes sense to differentiate two cases, namely where we have to perform Ace compilation online and where we can take it offline. As discussed in 5.3.3, the former might be necessary if a moving horizon scheme is to be applied, where we don't handle the complete length of $\mathcal{P}$ all at once (see section 7.1 for such a scheme).

In case we have to do the Ace compilation online, Ace still performs very well, giving us results in under two seconds for five instances out of eight. That trend is continued by Ace 3.0, which also solves the biggest instances fm3(long $\mathcal{P}$) with and without observations in under two seconds. However, except for the fm3(long $\mathcal{P}$) instance without observations, Toulbar2 is better than Ace in either runtime (e.g., fm2(long $\mathcal{P}$)) or memory usage (e.g., sm). In some cases, this might be due to the fact that Toulbar2 is implemented completely in C++, while parts of Ace are implemented in Java. In others, however, the advance of Toulbar2 is harder to explain away with implementation, see, e.g., the runtime for instances fm1, fm2 and sm or the memory usage for fm2 and sm (note that we used measurements offered by Ace itself, and we take it that these don't include irrelevant overhead induced by, e.g., the choice of implementation).

In case we can take Ace compilation offline, the two tools are definitely in the same league. For some instances Ace now has an edge over Toulbar2, see, e.g., instances fm3, sm or em, while for others, Toulbar2 is still slightly better (e.g. fm1, fm2).

An interesting side observation is that the two instances fm2 and fm2(long $\mathcal{P}$) seem to be harder for the Ace arithmetic circuit compilation than the *larger* instances fm3 and fm3(long $\mathcal{P}$).

Let us sum up the discussion. From our results we conclude that both approaches work, at least for our examples. In comparison, both approaches seem viable alternatives to consider for plan assessment: While Toulbar2 (model-based diagnosis approach) tends to be better on our instances, Ace (probabilistic reasoning approach) still performs well, and performs as well as or even better than Toulbar2 under the optimistic assumption that we don't have to perform Ace compilation online. Furthermore, Ace might scale better for bigger instances. While we consider the approaches in general equal alternatives, more experiments with different models could very well uncover a clear preference for one or the other approach+solver combination.

For the model-based diagnosis approach with Toulbar2 the limited scalability can be remedied, to an extent, by approximation. When approximating, a good choice for k to start with would be $k = 40$, when using Toulbar2. The results indicate that this is the better choice than the older Toolbar.

7. Extensions

This chapter presents three extensions of our plan assessment approaches. The first extension addresses problem instances with plans that would be too long to be solved all at once. It introduces a step-wise approach that computes most probable trajectories within a fixed time window and then moves this window along the time line.

The second extension addresses the problem of error bounds for the success probabilities, which was already briefly covered for the model-based diagnosis approach. The extension exploits statistical properties of sampling algorithms that can be exploited when using our probabilistic reasoning approach. These properties allow to compute *confidence intervals* around approximated success probabilities.

The third extension is motivated by the fact that, often, systems behave in ways that are easy to capture with a combination of discrete state transitions and continuous evolutions described with differential equations. To make this sort of modeling available for plan assessment we introduce a hybrid extension of PHCA and develop an automated translation to purely discrete models. Then, we can simply apply the framework we developed in section 5.2.

7.1. A Receding Horizon Extension for the Model-Based Diagnosis Approach

Up to now, we have always considered the complete length of the plan $N_{\mathcal{P}}$, i.e. $N = N_{\mathcal{P}}$. However, since enumerating trajectories for plan assessment is exponential in the number of time steps, this can quickly grow intractable with growing number of time steps. In our experiments we found that the model-based diagnosis approach did not scale beyond $N_{\mathcal{P}} \approx 14$ time steps when using the solver Toolbar. Toulbar2 can solve bigger problems with $N_{\mathcal{P}} \approx 30$. However, the general complexity of enumeration remains the same for Toulbar2, hence we suspect that with larger plans, e.g. $N_{\mathcal{P}} \approx 100$, Toulbar2 will also run into trouble. From the comparison results in chapter 6 we see that the probabilistic reasoning approach with Ace apparently has problems, too, with larger plans.

In [120] we address this scalability issue by combining the model-based diagnosis approach (as presented in [121]) with a receding time horizon scheme [108, 129]. In this section, we present a revised version of this work.

The receding horizon scheme slides a time window of fixed, small length N along the time line in order to cover larger schedules, for example $N_{\mathcal{P}} \approx 100$. Most probable trajectories are only generated within this window, which renders the problem exponential only in $N \ll N_{\mathcal{P}}$. Preliminary results show that the approach works in principle. However, a problem remains: the moving of the time window makes the involved constraint optimization problem potentially much harder.

Note that for this approach the same design choice between accurate diagnosis or accurate success probabilities is necessary as described in section 5.2.

7.1.1. Plan Assessment Based on Time Window Filtering

Given a manufacturing plan $\mathcal{P}$ and a goal G_i for some product i, let its end time be t_{G_i}. To compute $Pr(\mathcal{G}_i \,|\, \mathbf{o}^{0:t})$ for this product, we move the time window until it covers t_{G_i}. Each moving step involves enumerating k-best trajectories. When we reach t_{G_i}, we perform our approximative summation over goal-achieving trajectories among the k-best.

The idea of our moving-time-window-scheme is to combine trajectory enumeration with hidden Markov model filtering. Remember that PHCAs have a hidden Markov model semantic. In filtering, the distribution over all states at time t_0 is computed, given the distribution for the previous time point and current observations. Trajectory enumeration computes the k most likely sequences of length N, allowing to approximate a distribution over trajectories. Time window filtering (k-N-TWF) combines both methods: within a time window $t_0..t_N$, k trajectories of length N are generated. Then the time window of length N is moved one step forward and the distribution $Pr(X^{t_1} = m^{t_1}, \mathbf{O}^{0:t} = \mathbf{o}^{0:t})$ over the new initial states is computed recursively from distributions over previous initial states at t_0 and over trajectories within $t_0..t_N$.

Trajectories are sequences of markings, i.e. $\theta = m^1, m^2, \ldots, m^n$. In both the described translations, to constraint nets and to Bayesian logic nets (and Bayesian nets), a marking in the end is an assignment to binary location variables, e.g. $X^{t_1} = m^{t_1}$. Let location variables be mapped, for each time point t, to a single combined variable X^t with all possible markings for that time point as their values. PHCA variables Π are either known (commands, observations) or are determined through location variables, and can therefore be ignored here. The same applies for unknown observations within the time window, which don't influence the conditional dependency relations.

We abbreviate assignments to random variables in $Pr(X^{t_1} = m^{t_1}, \mathbf{O}^{0:t} = \mathbf{o}^{0:t})$ to $Pr(m^{t_1}, \mathbf{o}^{0:t})$. We denote the starting and ending time points of the time window with $t = t_0$ and $t = t_N$, respectively. $t = 0$ is the first time point of the plan and $t = t_c$ is the current time point, i.e. no observations are available after t_c. Unless stated otherwise, we mean the current time point when writing t. As an example consider the following markings, which represent the first and second marking relative to the plan start time $t = 0$, the first and last marking within a time window of length N, the marking at the current time point and the marking when product i is expected to finish: m^0, m^1, m^{t_0}, m^{t_N}, m^t, $m^{t_{G_i}}$. The notations $t+x$ and $t-x$ indicate x time points ahead or behind time t. Observations, as well as observation and location variables, are indexed analogously.

Theorem 2. *Let* $\alpha = \frac{1}{\Pr(\mathbf{o}^{t_1:t})}$. *Given observations* $\mathbf{o}^{0:t}$ *we can compute the distribution over system states at* t_1 *recursively as*

$$Pr(m^{t_1}, \mathbf{o}^{0:t}) = \alpha \sum_{X^{t_0}, X^{t_2}, \ldots, X^{t_N}} Pr(m^{t_1}, \ldots, X^{t_N}, \mathbf{o}^{t_1:t} | X^{t_0})\, Pr(X^{t_0}, \mathbf{o}^{0:t})$$

Proof. Observe that

$$\begin{aligned} Pr(m^{t_1}, \mathbf{o}^{0:t}) &= \\ \sum_{X^0, X^1, \ldots, X^{t_0}, X^{t_2}, \ldots, X^{t_N}} Pr(X^0, X^1, \ldots, X^{t_0}, m^{t_1}, X^{t_2}, \ldots, X^{t_N}, \mathbf{o}^{0:t}) &= \\ \sum_{X^0, X^1, \ldots, X^{t_0}, X^{t_2}, \ldots, X^{t_N}} Pr(m^{t_1}, X^{t_2}, \ldots, X^{t_N} | X^0, \ldots, X^{t_0}, \mathbf{o}^{0:t})\, Pr(X^0, \ldots, X^{t_0}, \mathbf{o}^{0:t}) & \end{aligned}$$

Then with Markov property ($X^{t_1}, \ldots, X^{t_N}$ independent of $X^0, \ldots, X^{t_0 - 1}$ and $\mathbf{O}^{0:t_0}$) we have

$$\begin{aligned} &= \sum_{X^0, X^1, \ldots, X^{t_0}, X^{t_2}, \ldots, X^{t_N}} Pr(m^{t_1}, X^{t_2}, \ldots, X^{t_N} | X^{t_0}, \mathbf{o}^{t_1:t})\, Pr(X^0, \ldots, X^{t_0}, \mathbf{o}^{0:t}) \\ &= \alpha \sum_{X^0, X^1, \ldots, X^{t_0}, X^{t_2}, \ldots, X^{t_N}} Pr(m^{t_1}, X^{t_2}, \ldots, X^{t_N}, \mathbf{o}^{t_1:t} | X^{t_0})\, Pr(X^0, \ldots, X^{t_0}, \mathbf{o}^{0:t}) \\ &= \alpha \sum_{X^{t_0}, X^{t_2}, \ldots, X^{t_N}} Pr(m^{t_1}, X^{t_2}, \ldots, X^{t_N}, \mathbf{o}^{t_1:t} | X^{t_0}) \sum_{X^0, \ldots, X^{t_0 - 1}} Pr(X^0, \ldots, X^{t_0}, \mathbf{o}^{0:t}) \\ &= \alpha \sum_{X^{t_0}, X^{t_2}, \ldots, X^{t_N}} Pr(m^{t_1}, X^{t_2}, \ldots, X^{t_N}, \mathbf{o}^{t_1:t} | X^{t_0})\, Pr(X^{t_0}, \mathbf{o}^{0:t}) \end{aligned}$$

We now have the needed recursive form:

$$Pr(m^{t_1}, \mathbf{o}^{0:t}) \quad = \quad \alpha \sum_{X^{t_0}, X^{t_2}, \ldots, X^{t_N}} Pr(m^{t_1}, X^{t_2}, \ldots, X^{t_N}, \mathbf{o}^{t_1:t} | X^{t_0})\, Pr(X^{t_0}, \mathbf{o}^{0:t})$$

□

This recursive term represents the exact computation, enumerating all trajectories. As this becomes intractable quickly, we need an approximative time window scheme, based on the k most probable trajectories instead. Given these trajectories, we compute approximations for α and the distribution $Pr(X^t, \mathbf{o}^{0:t})$. Every time the time window is moved we add $Pr(X^t, \mathbf{o}^{0:t})$ as constraint C_{SD} to the original constraint problem. External solvers such as Toolbar or Toulbar2 then compute the k most probable trajectories along with their joint probabilities $Pr(m^{t_0:t_N}, \mathbf{o}^{0:t}) = Pr(m^{t_1}, \ldots, m^{t_N}, \mathbf{o}^{t_1:t} | m^{t_0})\, Pr(m^{t_0}, \mathbf{o}^{0:t})$ as basis for the next step[1]. Algorithm 7.1 implements k-N-TWF.

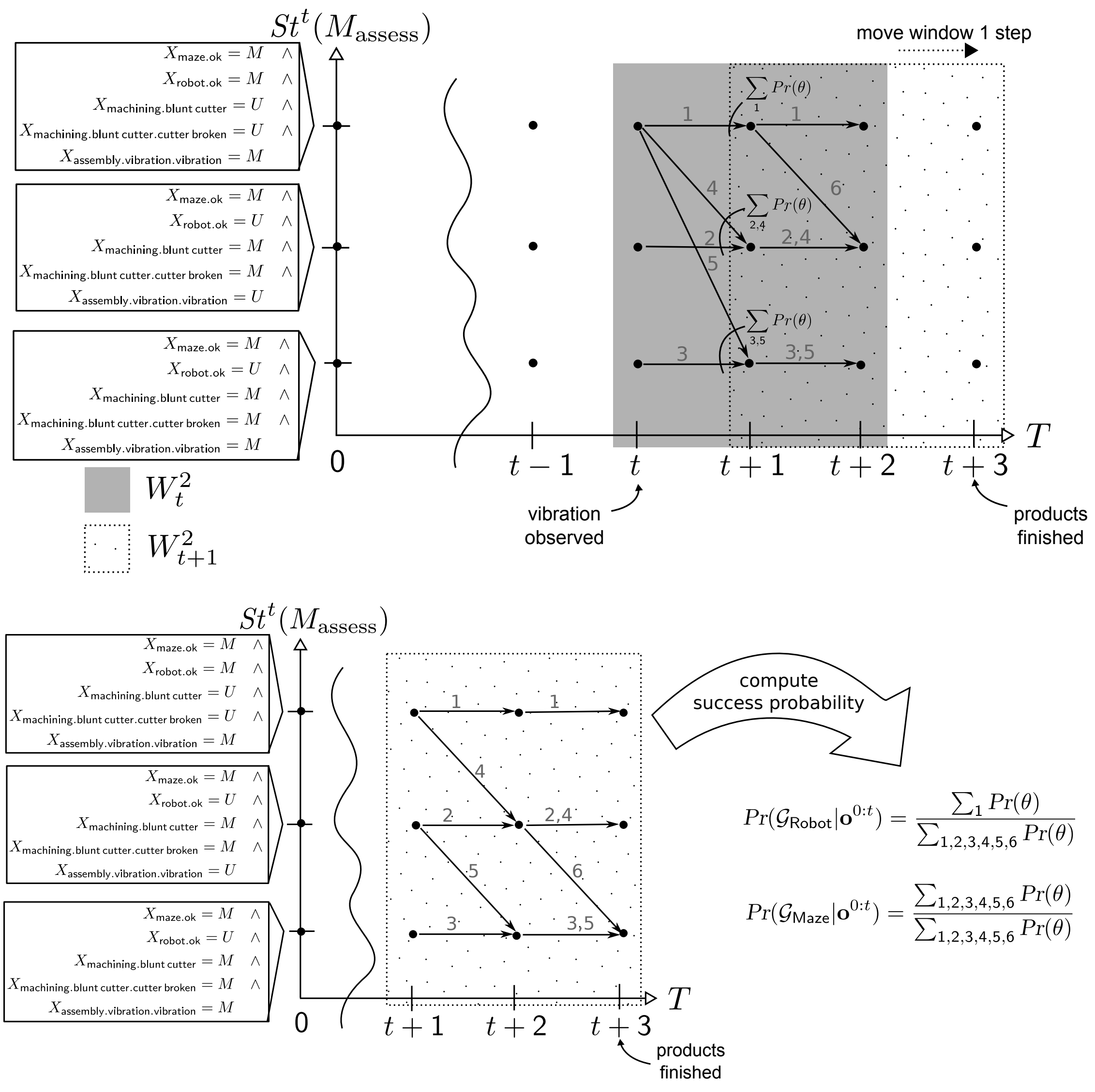
[1]The solvers compute full assignments along with their probabilities. However, for the summation necessary for computing $Pr(\mathcal{G}_i \,|\, \mathbf{o}^{0:t})$, it doesn't matter whether we sum over the full assignments and their probabilities, or over partial sums $Pr(m^{t_0:t_N}, \mathbf{o}^{0:t}) = \sum_{\mathbf{o}^{t+1:N}} Pr(m^{t_0:t_N}, \mathbf{o}^{0:t}, \mathbf{o}^{t+1:N})$.

Algorithm 7.1 The Time Window Filtering algorithm. Function AsUnaryConstraints maps observations to unary constraints over observation variables, function TimeStepStates maps a set of trajectories to the set of states for the given time point.

```
function kNTWF($W_{t_0}^N$, $o^{t_0:t}$, $\mathcal{P}$, $k$)
    $\Theta(k) \leftarrow$ ESTIMATE($W_{t_0}^N$, $o^{t_0:t}$, $\mathcal{P}$, $k$)
    $W_{t_0+1}^N \leftarrow$ SHIFTTIMEWINDOW($W_{t_0}^N$, $\Theta(k)$)
     return $(\Theta(k), W_{t_0+1}^N)$
end function
function ESTIMATE($W_{t_0}^N$, $o^{t_0:t}$, $\mathcal{P}$, $k$)
    $(\mathcal{R}, N, t_0) \leftarrow W_{t_0}^N$
    $C' \leftarrow \mathcal{R}.C \cup$ AsUnaryConstraints($\mathbf{o}^{t_0:t}$)
    $C' \leftarrow C' \cup$ ProductComponentLinks($\mathcal{P}$, $\mathcal{R}$)
    $\mathcal{R}' \leftarrow \langle \mathcal{R}.X, \mathcal{R}.D, C' \rangle$
    $\Theta(k) \leftarrow$ SOLVE($\mathcal{R}'$, $k$)
     return $\Theta(k)$
end function
function SHIFTTIMEWINDOW($W_{t_0}^N$,$\Theta(k)$)
    $(\mathcal{R}, N, t_0) \leftarrow W_{t_0}^N$
    if $t_0 = 0$ then
        Remove constraints for initial location marking from $\mathcal{R}$
    end if
    $C_{\mathrm{SD}} \leftarrow \{m^{t_1} \mapsto$ SumTrajectoriesForMarking($m^{t_1}$, $N$) $|m^{t_1} \in$ TimeStep-
States($\Theta(k)$, 1) $\}$
    $\mathcal{R}.C \leftarrow \mathcal{R}.C \cup C_{\mathrm{SD}}$                              $\triangleright$ overrides previously added $C_{\mathrm{SD}}$
     return $(\mathcal{R}, N, t_0 + 1)$
end function
function SumTrajectoriesForMarking($m^{t_1}$, $N$)
     return $\alpha \sum_{X^{t_0}, X^{t_2}, \ldots, X^{t_N}} Pr(m^{t_1}, X^{t_2}, \ldots, X^{t_N}, \mathbf{o}^{t_1:t} | X^{t_0})\, Pr(X^{t_0} | \mathbf{o}^{0:t})$
end function
```

As an example, we again consider the scenario where vibrations of potentially blunt cutters are picked up by sensors as observations, as described in section 6.3. Figure 7.1 illustrates the plan assessment using a moving time window. In this example we focus on three markings from M_{PHCA}: 1) a marking were both maze and robot arm are "ok", 2) a marking were the cutter broke and thus led to a flawed robot, but the maze is "ok" and 3) like 2, with the difference that here additionally vibrations in the assembly occurred (not only caused by the blunt/broken cutter). Note that for clarity the location domain $\{\mathsf{marked}, \mathsf{unmarked}\}$ is abbreviated to $\{\mathsf{M}, \mathsf{U}\}$ and the time point ^{t} is omitted in figure 7.1, e.g. $X_{\mathsf{maze.ok}} = \mathsf{M}$.

A time window of $N = 2$ (two time steps, i.e. three time points) is used and $k = 6$ trajectories are enumerated by the external solver. We use the notation $W_t^N = (\mathcal{R}, N, t)$



$St^t(M_{\text{assess}})$
$X_{\text{maze.ok}} = M \wedge$
$X_{\text{robot.ok}} = M \wedge$
$X_{\text{machining.blunt cutter}} = U \wedge$
$X_{\text{machining.blunt cutter.cutter broken}} = U \wedge$
$X_{\text{assembly.vibration.vibration}} = M$
$X_{\text{maze.ok}} = M \wedge$
$X_{\text{robot.ok}} = U \wedge$
$X_{\text{machining.blunt cutter}} = M \wedge$
$X_{\text{machining.blunt cutter.cutter broken}} = M \wedge$
$X_{\text{assembly.vibration.vibration}} = U$
$X_{\text{maze.ok}} = M \wedge$
$X_{\text{robot.ok}} = U \wedge$
$X_{\text{machining.blunt cutter}} = M \wedge$
$X_{\text{machining.blunt cutter.cutter broken}} = M \wedge$
$X_{\text{assembly.vibration.vibration}} = M$
move window 1 step
$\sum_1 Pr(\theta)$
$\sum_{2,4} Pr(\theta)$
$\sum_{3,5} Pr(\theta)$
1
4
2
5
3
6
2,4
3,5
T
0
$t-1$
t
$t+1$
$t+2$
$t+3$
vibration observed
products finished
W_t^2
W_{t+1}^2
compute success probability
$Pr(\mathcal{G}_{\text{Robot}}|\mathbf{o}^{0:t}) = \frac{\sum_1 Pr(\theta)}{\sum_{1,2,3,4,5,6} Pr(\theta)}$
$Pr(\mathcal{G}_{\text{Maze}}|\mathbf{o}^{0:t}) = \frac{\sum_{1,2,3,4,5,6} Pr(\theta)}{\sum_{1,2,3,4,5,6} Pr(\theta)}$

Figure 7.1.: (a) Moving time window W_t^2 one step, generating time window W_{t+1}^2. (b) Computing success probabilities after $k = 6$ trajectories have been enumerated in the new time window W_{t+1}^2.

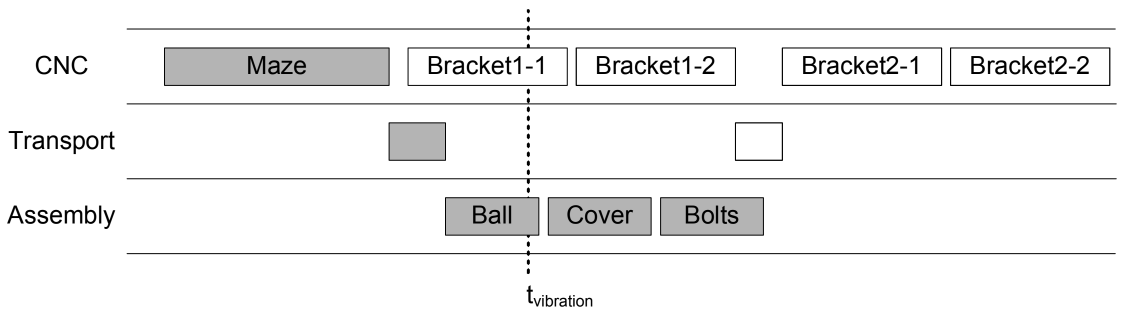


Figure 7.2.: A visualization of a schedule of a maze (dark) and a robot arm (bright) used as plan $\mathcal{P}$ for the receding horizon example.

for a time window of length N starting at time point t, encoded as a COP $\mathcal{R} = (X, D, C)$. Trajectories are numbered 1 through 6, with 1 most and 6 least probable. At first, the time window does not cover the time point when the products are finished (W_t^2) and thus success probabilities can't be computed. The time window is moved (W_{t+1}^2), and in doing so probabilities of trajectories leading to the same initial marking in the next window are summed up. They are normalized using the approximation $\alpha^* = \frac{1}{\sum_{\theta \in \Theta(k)} Pr(\theta, \mathbf{o}^{0:t})}$. This is done for all initial markings, and thereby the approximate initial distribution for the next time window is created. Now the window covers the finishing time point. Again, $k = 6$ trajectories are enumerated within this time window. In this example, only one of these trajectories is goal-achieving for the robot arm, while all of them lead to a properly finished maze. Summing over the respective goal-achieving and normalizing over all enumerated trajectories yields approximate success probabilities for the maze and the robot arm. Note that the time window is moved into the future until it covers the end time t_{G_i} for some goal G_i.

7.1.2. Preliminary Experimental Results and Discussion

Here we present results that were first presented in [120]. We implemented the k-N time window filtering (k-N-TWF) and ran experiments on an Intel core2duo machine with 2.53 Ghz and 4GB of RAM, using a simple factory PHCA model (see figure 7.3) for our scenario. The original scenario, introduced in the previous section, has $N_{\mathcal{P}} = 9$ time steps, which can be solved without TWF with an $N = 9$ time window. We used the older Toolbar solver in these experiments, with which we could solve variations of our scenario with plan length up to 14 time steps. These variations where generated by simply multiplying planned actions. In a preliminary experiment using our novel TWF method together with Toolbar we could compute success probabilities for a plan with $N_{\mathcal{P}} = 29$,

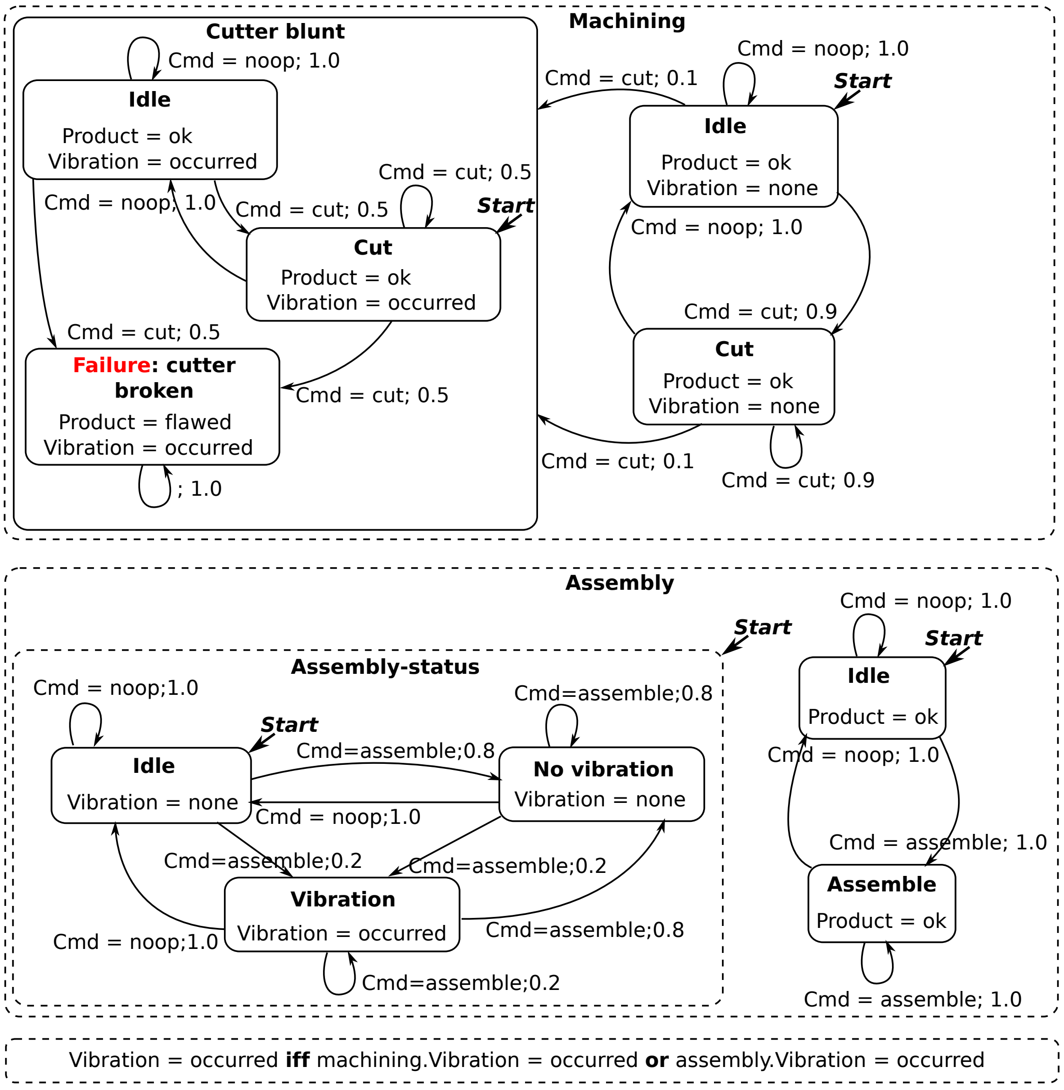


Figure 7.3.: PHCA model used for k-N-TWF experiments. Not shown here are the product models for the robot and the maze, which are the same as in the model shown in figure 3.6.

using a time window of length $N = 5$ in 240 seconds. We moved the time window along the whole plan length, doing 24 shift operations, where each interleaved estimation step took 4-10 seconds.

While it seems that Toulbar2 can solve problem instances of this size much quicker, it is well possible that it will hit its border soon, too, due to the generally exponential dependency on plan length. Just as TWF can be used to overcome the limitation of Toolbar, it should be possible to overcome similar limitations of Toulbar2 as well.

TWF allows to address bigger plans. However, if we compare the TWF performance with computing success probabilities without TWF for short enough plans, e.g. our problem with $N_{\mathcal{P}} = 9$, we see that TWF needs much more runtime. The latter is done in about 4 s, while the former, TWF, takes much longer to solve the same problem (see table 7.1). Partly this is due to our prototype implementation. However, the bigger problem is that the shift operation makes the subsequent COP, solved to generate trajectories, much harder: Computing the distribution over initial states for the next time window, even if it is only approximated, yields a big constraint over many location variables in the COP. The older Toolbar solver fails to even load the problem if we have more than 30 PHCA locations, because the mentioned constraint becomes too large. Toulbar2 might not have this problem, it has routines that automatically decompose large constraints.

Table 7.1 shows how much the success probability computed for the robot for this instance deviates from the exact computation (done by enumerating all feasible trajectories) when varying k and N. Clearly, if k is big enough to enumerate all non-zero trajectories (> 80 for our problem instance), TWF yields the exact solution independently of N. When choosing only few trajectories ($k \leq 5$), increasing N seems to reduce the error eventually, while for more trajectories an increased N can actually *increase* the error. Increasing k eventually increases the accuracy of the approximation. However, this happens in a non-monotonic way, just like for the instances investigated in our evaluation experiments, described in chapter 6.

7.1.3. Future Work

For future work, the first thing would be to try the TWF algorithm with Toulbar2 as backend and see whether we will have similar problems as with Toolbar. Then, a significant improvement would be to extend the shift operation such that the time window can be moved multiple time steps at once, ideally, for its complete length N. This would reduce the number of shift-estimate iterations needed to cover the length of a given plan. Finally, in order to become practical, it should be possible to obtain success probabilities with shift operations being done alongside the step-wise execution of the plan. Currently,

Table 7.1.: Comparing the success probability error (above) and runtime (in seconds, below) for the robot product for different number of trajectories k and time window sizes N. The error is $\epsilon_i(k) = |Pr(\mathcal{G}_i \mid \mathbf{o}^{0:t}) - Pr^k(\mathcal{G}_i(k) \mid \mathbf{o}^{0:t})|$, that is the absolute difference between exact and approximate success probability.

k	$N=1$	$N=2$	$N=3$	$N=4$	$N=5$	$N=6$	$N=7$	$N=8$	$N=9$
2	0.46	0.46	0.46	0.46	0.26	0.21	0.21	0.21	0.21
	39.77	36.14	34.44	35.84	50.82	57.56	57.14	47.50	27.65
3	0.42	0.46	0.46	0.46	0.46	0.25	0.25	0.25	0.25
	39.68	36.08	34.39	36.54	50.85	58.05	57.58	47.28	27.97
4	0.26	0.30	0.36	0.36	0.36	0.28	0.28	0.28	0.28
	39.56	35.96	34.35	36.02	50.81	58.62	57.34	48.88	27.94
5	0.25	0.31	0.34	0.32	0.26	0.21	0.21	0.21	0.21
	39.54	36.47	34.55	36.68	51.25	59.26	58.61	48.15	29.30
10	0.06	0.13	0.09	0.09	0.11	0.12	0.12	0.12	0.12
	40.92	36.09	34.89	36.49	53.37	59.68	59.11	48.45	28.34
25	0.00	0.02	0.03	0.06	0.06	0.05	0.05	0.05	0.05
	41.03	36.66	35.49	37.73	54.97	61.95	61.73	48.50	28.62
50	0.00	0.00	0.00	0.01	0.01	0.01	0.01	0.01	0.01
	40.32	37.19	37.69	39.19	54.46	60.72	60.04	50.79	29.34
100	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	40.89	39.26	37.53	39.23	54.87	61.56	60.76	50.56	29.98

this is not possible because, in order to compute a success probability for some goal G_i, this goal's time point needs to be within the limited horizon of the time window. With the current approach, this requires to move the time window until it covers this time point, potentially requiring many shift operations. What is needed is an extension of plan assessment which allows to compute some approximation of the success probability based on the available time horizon, even if the goal lies further in the future.

7.2. Stochastically Bounded Approximation of Success Probabilities with Sampling

For our model-based diagnosis approach to plan assessment, the question of how to compute the error of approximation depending on the number of generated trajectories is still an open question. However, with the translation of PHCA to BLN, introduced in the previous section, another class of approximative algorithms becomes available for the computation of success probabilities: sampling. As explained in section 3.5.2, sampling poses itself as an attractive anytime approximation alternative, which generates probable system trajectories randomly, according to the probabilities provided by a model such as a PHCA.

In this section, we present a revised version of work that has been published before in [118]. Therein we propose to combine sampling with a stochastic accuracy guarantee for the computation of marginals, in particular success probabilities, in terms of the *confidence interval.* The more samples are generated, the narrower this interval becomes. We demonstrate this approach on a simplified, manually created BLN model for our example plan assessment scenario. While this model is hand-crafted, it is a BLN exactly like those generated by our translation from PHCAs. Therefore, this approach is readily applicable to PHCA models, too.

Typical plan assessment models are in large parts deterministic. Sampling in these types of models is prone to the rejection problem (explained in section 3.5.2), i.e. many samples are drawn that produce inconsistent assignments. A recently proposed scheme called SampleSearch [75] addresses this problem by exploiting constraint-solving methods to quickly exclude inconsistent samples. We compare this approach against two other sampling algorithms, namely likelihood weighting [71] and backward simulation [70], in terms of how fast they can reduce the confidence interval size.

7.2.1. Computing Confidence Intervals for Plan Success Probabilities

We are now interested in computing bounds on the success probability $Pr(\mathcal{G}_i \mid \mathbf{o}^{0:t})$, rather than the probability itself. That is, in this section we address the following modified plan assessment problem:

Problem 8. Given a BLN model and observations $\mathbf{o}^{0:t}$ obtained up to time point t, compute "soft" lower and upper bounds p_l^* and p_u^* on each success probability $Pr(\mathcal{G}_i \mid \mathbf{o}^{0:t})$ for all goals $\{G_i\}$ of a given manufacturing plan $\mathcal{P}$, such that $p_l^* \leq Pr(\mathcal{G}_i \mid \mathbf{o}^{0:t}) \leq p_u^*$ holds with a predefined probability γ.

The bounds p_u^* and p_l^* are "soft" because it is not guaranteed that the success probability $Pr(\mathcal{G}_i \mid \mathbf{o}^{0:t})$ is within these bounds. Only a stochastic guarantee is given that the success probability is within the bounds with *coverage probability* γ. The coverage probability is a parameter, to be specified beforehand by the user. The higher this probability is chosen, i.e. the more sure we want to be, the more samples must be generated. The bounds form what is called a confidence interval. Next, we will see that the bounds we seek are given by the so-called Clopper-Pearson interval [39], defined for Bernoulli-distributed Boolean random variables.

The key to computing the bounds is the fact that sampling induces a *distribution* over potential probability values for the success probability we are interested in. With this distribution the bounds can be computed analytically using the inverse cumulative distribution function of this distribution.

Before we start, we have to analyze the relationship between a goal G_i and its success probability $Pr(\mathcal{G}_i \mid \mathbf{o}^{0:t})$. Goals are defined as tuples (l, t) of PHCA locations l that should be marked at some time t. In both our approaches this boils down to being encoded as a binary variable X_l^t, which takes values from domain $\{\mathsf{marked}, \mathsf{unmarked}\}$. It is a binary random variable that is *Bernoulli-distributed* with parameter $p = Pr(\mathcal{G}_i \mid \mathbf{o}^{0:t}) = Pr(X_l^t = \mathsf{marked} \mid \mathbf{o}^{0:t})$. The distribution over a binary variable with probabilities p and $1-p$ is called *Bernoulli-distribution.*

When we sample full assignments of the BN generated from the PHCA that contains location l, this implies sampling the variable X_l^t according to its (unknown) success probability. This process of sampling a binary random variable corresponds to the stochastic *Bernoulli-process* (which is, formally, a finite or infinite sequence of independent Bernoulli-distributed binary random variables, each having the same parameter p). It is usually imagined as repeatedly flipping a coin that is potentially unfair (as p could take other values than $\frac{1}{2}$). The sampling generates a number of $X_l^t = \mathsf{marked}$ assignments and a number of $X_l^t = \mathsf{unmarked}$ assignments, a and b, respectively. These numbers are the parameters of the distribution over potential success probability values we talked about. The distribution is the beta distribution, if we assume that the potential success probability values, i.e. parameters p, are uniformly distributed when we have no samples [22, pp. 67–74]. The cumulative distribution function of the beta function is given as the regularized incomplete beta function.

Clopper and Pearson introduced a specific type of confidence interval for p, with bounds derived from the inverse regularized incomplete beta function: the Clopper-Pearson interval [39]. It allows us to analytically compute stochastic bounds for p.

Theorem 3. *The bounds p_u^* and p_l^* on success probability $Pr(\mathcal{G}_i \mid \mathbf{o}^{0:t})$ are given by the Clopper-Pearson interval [39].*

Proof. Let G be a Bernoulli-distributed Boolean random variable with parameter p, which is being sampled in a Bernoulli-process, counting appearances of $G = \mathsf{true}$ and $G = \mathsf{false}$ as a and b, respectively. Let γ be the coverage probability. Then the Clopper-Pearson interval defines bounds $p_l^* = F_{a,b,\gamma}^{-1}(1 - \frac{\alpha}{2})$ and $p_u^* = F_{a,b,\gamma}^{-1}(\frac{\alpha}{2})$, where $\alpha = 1 - \gamma$ and $F_{a,b,\gamma}^{-1} = I_{a+1,b+1}^{-1}$, I being the regularized incomplete beta function; I^{-1} is thus the inverse of the cumulative distribution function (CDF) of the beta distribution. Any plan assessment goal $G_i = (l, t)$ has an associated Bernoulli-distributed Boolean random variable X_l^t with parameter $p = Pr(\mathcal{G}_i \mid \mathbf{o}^{0:t})$. We sample trajectories that correspond to the observations and entail assignments to X_l^t. Therefore, the trajectory sampling can be seen as a sampling of X_l^t. The sampling yields n samples, $n_{X_l^t}$ goal-achieving and $n - n_{X_l^t}$ goal-violating. If we now set $G = X_l^t$ (assuming an appropriate mapping of domains), $a = n_{X_l^t}$, $b = n - n_{X_l^t}$, the theorem follows. $\square$

We quickly recap the formalities of how this works. To abbreviate, we denote the random variable X_l^t as G (for goal). As mentioned, the quantities n_G, $n - n_G$ determine a beta distribution over p [22]. The cumulative distribution function $F_{n_G,n-n_G}(x) = Pr(p \leq x)$ allows to compute the probability that p is at most x. Observe now that the complement of the given $\gamma = Pr(p_l^* \leq p \leq p_u^*)$, i.e. the probability that p is *outside* the bounds of the interval, can be written as $Pr(p < p_l^*) + Pr(p > p_u^*) = 1 - \gamma = \alpha$. We can rewrite this equation as $Pr(p \leq p_l^*) + 1 - Pr(p \leq p_u^*) = \alpha$, where all probabilities are represented through the cumulative distribution function $F_{n_G,n-n_G}(x)$. Now we can use the inverse cumulative distribution function $F_{n_G,n-n_G}^{-1}(y)$ to compute the bounds p_l^* and p_u^*. Except in extreme cases, we can assume that α is to equal parts composed of $Pr(p \leq p_l^*)$ and $1 - Pr(p \leq p_u^*)$, i.e. $Pr(p \leq p_l^*) = \frac{\alpha}{2} = 1 - Pr(p \leq p_u^*)$. Resolving for p_l^* yields $p_l^* = F_{n_G,n-n_G}^{-1}(\frac{\alpha}{2})$ and for p_u^* gives $p_u^* = F_{n_G,n-n_G}^{-1}(1 - \frac{\alpha}{2})$.

Table 7.2.: Confidence intervals on success probabilities for the mazes in scenario 1 obtained with likelihood weighting and exact results obtained through variable elimination. The first two rows show results for two different coverage rates γ. The last row shows the runtimes in seconds for both coverage rates.

coverage rate γ		Number of samples 100	2500	10000	Exact
0.95	mazeOK(M2,T7)	$[0.002, 0.054]$	$[0.035, 0.051]$	$[0.035, 0.042]$	0.039
	mazeOK(M1,T9)	$[0.593, 0.772]$	$[0.591, 0.629]$	$[0.587, 0.606]$	0.604
	mazeOK(M0,T5)	$[0.000, 0.029]$	$[0.000, 0.001]$	$[0.000, 0.000]$	0.000
0.999	mazeOK(M2,T7)	$[0.010, 0.162]$	$[0.030, 0.057]$	$[0.035, 0.048]$	0.039
	mazeOK(M1,T9)	$[0.359, 0.677]$	$[0.573, 0.637]$	$[0.594, 0.626]$	0.604
	mazeOK(M0,T5)	$[0.000, 0.066]$	$[0.000, 0.003]$	$[0.000, 0.001]$	0.000
	runtime	0.06	1.61	6.24	58.76
		0.06	1.52	6.00	58.76

Of course we would like to have as narrow intervals as possible. Increasing the number of samples gives us narrower intervals. Thus, a practical stop criterion for sampling algorithms is to predefine the size of the interval to be sufficiently small, and then sample until the interval is narrower than this predefined size. Note that an estimate for the success probability itself can be computed in the usual way by dividing the number of goal-achieving samples by the number of all samples, $Pr(\mathcal{G}_i \mid \mathbf{o}^{0:t}) \approx \frac{n_G}{n}$.

The above applies to rejection sampling, where $\frac{n_G}{n}$ is indeed the estimate. However, applying more advanced importance sampling techniques such as sample search require only a simple adaptation. The count n_G must be replaced with $n\tilde{p}$, where $\tilde{p}$ is the estimate of the importance sampler.

7.2.2. Experimental Results

We inferred the success probability of mazes for three different scenarios, with corresponding ground BNs instantiated from the manually created BLN model shown in section 3.7. Remember that this hand-crafted model abstractly models multiple possible scenarios. The three mentioned sampling algorithms [71, 70, 75] were tested, most notably SampleSearch. The results presented in tables 7.2 and 7.3 are those obtained within our joint work [118], where we ran Java implementations of the algorithms on an Intel core2duo with 2.53 Ghz and 4GB of RAM. The algorithms were implemented by Dominik Jain and are publicly available in his BLN toolbox[2].

[2]`http://www9-old.cs.tum.edu/people/jain/dl.php?get=probcog` (06.2011)

In all three scenarios there were two machining stations (Mach0/1) and one assembly station (Assembly0). Remember that here we denote time points as entities T1, T2, T3 and so on. Also, since we consider manufacturing, plans $\mathcal{P}$ are again schedules.

1. In the smaller *scenario 1* (size 310 BN nodes) Mach0 has become faulty. Its schedule ranges over nine time points for three mazes (Maze0/1/2). Observations have been made up to T4, and a force alarm was triggered at T4 while the assembly station was pushing pins into Maze0.

2. In *scenario 2* (520 nodes) Assembly0 is faulty. Here, four mazes (Maze0/1/2/3) are scheduled, covering 12 time points. Observations are available up to T8. The pin assembly is done at T4, T6 and T8 for Maze0, Maze2 and Maze1 respectively. A force alarm is observed at all three time points.

3. *Scenario 3* (520 nodes) is similar to the former, with the difference that again Mach0 is faulty. Consequently, at T8 no force alarm is triggered. In all scenarios Maze0 and Maze2 are cut on Mach0, while Maze1 and (in scenarios 2 and 3) Maze3 are cut on Mach1. Further, Maze0/1/2/3 should be finished by T5/9/7/12, respectively.

For these scenarios, good intervals ($\gamma = 0.95$, width less than 0.01) can already be retrieved in under a minute, sometimes even in under a second (scenario 2, SampleSearch), with less than 1000 samples as can be see from table 7.3. This table presents results from comparing the three sampling algorithms we used. We also see that it seems, depending on the scenarios, SampleSearch is not always the best algorithm: for scenarios 1 and 2 SampleSearch is best, while for 3, likelihood weighting is the better choice. A reason might be that the SampleSearch implementation used here uses simple backtracking to search for consistent assignments. We expect much better performance for SampleSearch implementations that employ more sophisticated constraint reasoning methods. Trying this is definitely a worthwhile future research direction.

Table 7.2 illustrates how choosing stricter coverage probabilities γ widens the interval. It also confirms that increasing the number of samples results in better intervals.

7.3. Towards Plan Assessment with Hybrid Models

Real-world technical systems such as factory plants often have components like tanks for liquids or gases. These types of components are often more conveniently modeled

[3]Backward simulation did not produce any results for scenario 3, because the problem was too ill-conditioned, such that no countable samples could be generated. SampleSearch does not have this problem and will always generate usable samples much quicker.

Table 7.3.: Comparing algorithms on scenarios 1 / 2 / 3 (columns) by average number of samples (above) needed to reach a target confidence interval width, and runtime in seconds (below). The interval is termed I. Different rows show different interval width'.

	Algorithm								
$\max_I \lvert I \rvert$	likelihood weighting			backward simulation			SampleSearch		
≤ 0.025	5900	320	2020	5820	200	-[3]	6180	200	1380
	3.61	0.90	3.82	1.23	5.44	-	0.84	0.06	13.27
≤ 0.01	36800	1000	11800	37280	300	-	38440	660	5080
	22.42	2.67	21.47	7.00	7.56	-	4.76	0.16	42.08
≤ 0.0025	588020	17420	185820	588860	1200	-	614700	6460	181320
	384.38	50.06	362.08	111.95	31.75	-	79.70	1.40	1551.25

with continuous rather than discrete variables, for example reals. A common example are filling stations such as the one depicted in figure 7.4. Shown is a photograph of a small, industrial plant that fills a granulate material (simulating liquids) into glass bottles. The plant was built for student training purposes at the engineering department of the Technische Universität München, and is described in detail in [54]. If, for example, we want to model the fill level of the tanks in this plant, it is more convenient to describe the fill level and its behavior over time with a real-valued variable and a *differential equation* over that variable, rather than hand-crafting a finite state machine.

The PHCA formalism allows to define complex, parallel running components such as stations working different products. However, it does not allow modeling with real-valued variables and differential equations. In this section, we further extend our model-based diagnosis approach to plan assessment towards *hybrid* models. These are models that encompass finite/discrete as well as continuous variables. We present a revised version of work published in [119], which introduces a hybrid extension to PHCA, namely Hybrid PHCA or HyPHCA. It allows continuous modeling with linear ordinary differential equations (linear ODE).

Hybrid models define an infinite number of possible trajectories. The problem is therefore to make computing of most probable trajectories tractable, in order to compute solutions to, for instance, the plan assessment problem. We address this problem with an abstraction-based approach that combines concepts, techniques and formalisms from AI (constraint optimization, hidden Markov model reasoning), fault diagnosis in hybrid systems (stochastic abstraction of continuous behavior), and hybrid systems verification (hybrid automata, reachability analysis). The approach automatically converts a Hybrid PHCA into a discrete model, which is then further converted into a constraint net. From

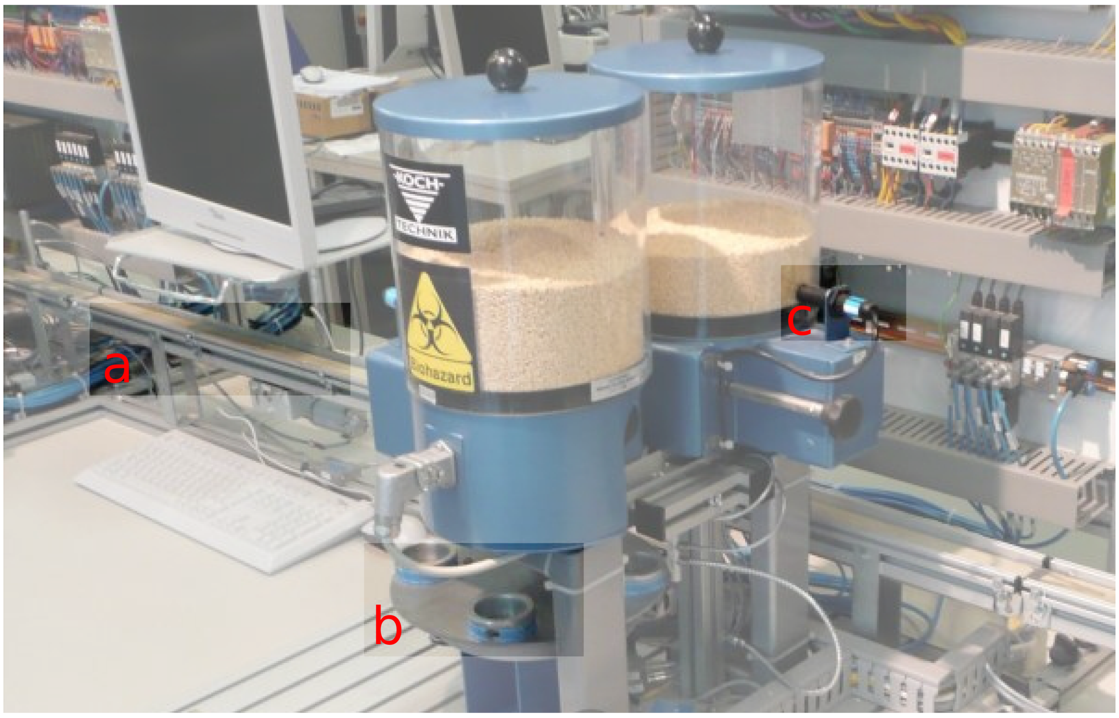


Figure 7.4.: Filling station with a) a conveyor belt that brings empty bottles, b) a swivel that moves the bottles below the nozzle of the silo and c) a photo sensor that gives a signal if the silo is empty. The bottles are placed upon the swivel by a pneumatic arm, which is not in the picture (it is behind the silos).

here, we can proceed as described in section 5.2, i.e. the constraint net is fed into a constraint solver to compute solutions for the plan assessment problem. Results with a model of the filling station shown in figure 7.4 demonstrate that the approach can be used to compute most probable diagnoses.

The key difference to similar existing approaches (which we describe in the next section) is that we do not develop special algorithms for one particular model framework (e.g. HyPHCA), but instead try, in line with the general theme of this thesis, to exploit off-the-shelf tools and general frameworks to compute solutions. Chiefly, we exploit constraint optimization [127] by using the constraint solver Toulbar2, but we also use existing solutions for other steps in our abstraction-based approach where possible. This avoids reinventing the wheel and it becomes easier to stay up-to-date with recent developments. In software engineering terms, we achieve a deeper separation of concerns with our approach. Furthermore, by extending PHCAs, a modeling framework which is explicitly designed for model-based development of embedded systems, we are moving closer to the over-arching ideal of one-model-fits-it-all, i.e. from system design to online model-based monitoring and control.

Before detailing the extension of PHCA and our abstraction-based approach, we introduce an example for a hybrid system and give a quick introduction to hybrid diagnosis, covering relevant state-of-the-art in the field.

7.3.1. A Filling Station as Hybrid System Example

As an example we use the previously mentioned plant, which is primarily composed of a filling station. The station fills a granulate material in small bottles, which are transported to and away from the station on a conveyor belt. A pneumatic arm moves bottles from the conveyor onto a swivel and back when they are finished. The swivel positions the bottles below a silo, where they are filled by a screw mechanism powered by an electrical motor. A photo sensor (binary signaled) indicates when the silo is empty.

We created a simplified model of the filling station (shown in figure 7.5), which models a silo, the filling mechanism with the electric motor and the silo sensor. Just as previously done for plan assessment, all these components are modeled as composite locations of a PHCA model, their respective behaviors (including potential faults) as locations and probabilistic transitions between them. For example, initially the filling station waits (location Wait is marked) until the command is given to switch the motor on with the aim to fill a bottle. Upon this command the station transitions to location Fill, which means the bottle is being filled. Should the silo become empty during filling, the station transitions to location Empty.

We also need to model the silo's fill level. It is convenient to model it continuously: the filling is modeled with a differential equation $\dot{U}_{\mathrm{lvl}} = -\mathrm{fR}$, where fR is the constant fill rate. To test and demonstrate the capability of our approach, we use the more complex (although still linear) equation $\dot{U}_{\mathrm{lvl}} = -\mathrm{fR} \cdot U_{\mathrm{lvl}}$. Combining these equations with the PHCA model of the components gives us a hybrid model: a *hybrid* PHCA, or HyPHCA for short.

We also modeled potential faults of the station components, the silo and the sensor. Within the silo component the electrical control of the motor might fail, causing it to ignore any issued commands and keep running. As a result of that fault the silo is continuously being emptied. We refer to this fault as the motor-switch-fault. The sensor component might fail in three different ways. It might be stuck on, stuck off, or show some unknown fault behavior. If stuck on, the sensor always signals an empty silo, and accordingly a non-empty silo if stuck to off. An unknown fault is modeled by allowing

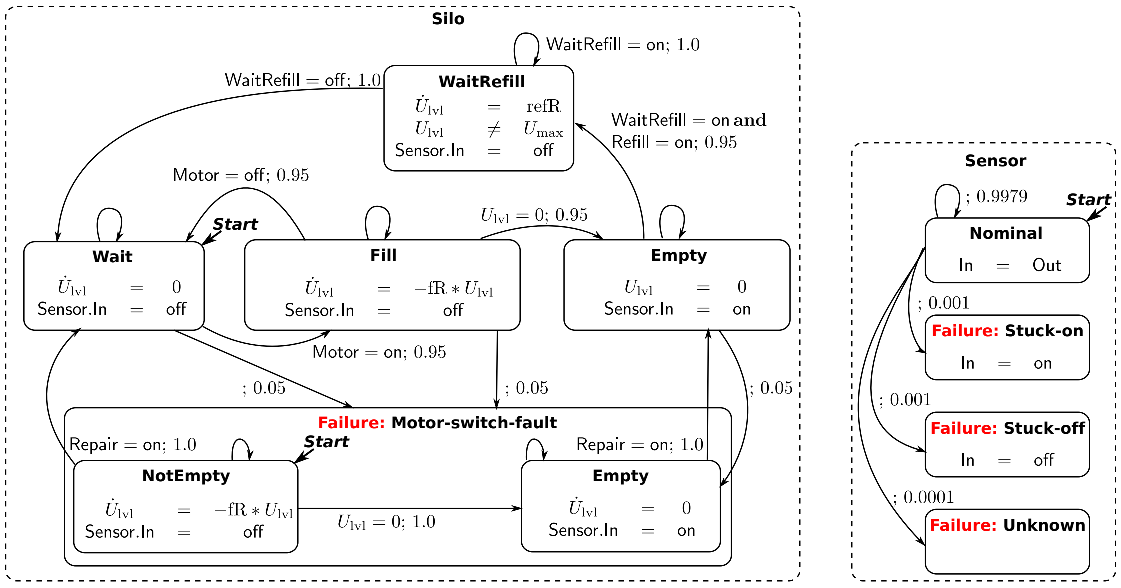


Figure 7.5.: HyPHCA modeling the silo and the silo empty sensor of a filling station.

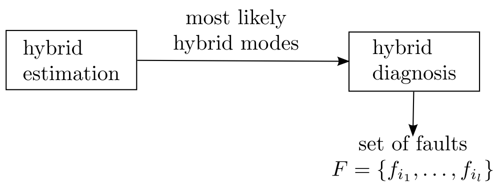


Figure 7.6.: Rough architecture of hybrid diagnosis based on hybrid estimation.

just any possible assignment to the sensor's In, Out variables. We refer to these three faults as s-on, s-off and s-unknown.

The complete model is shown in figure 7.5. The formal definition of hybrid PHCA will be given shortly; first, we give a quick overview of reasoning with hybrid models, especially hybrid diagnosis.

7.3.2. Hybrid Discrete/Continuous Model-Based State Estimation and Diagnosis

We know from section 4.4 that plan assessment is strongly related to diagnosis and state estimation. When it comes to hybrid models, a well studied problem in the literature is known as *hybrid diagnosis*. It is composed of two steps: First, estimating the current most likely hybrid state or estimating a most likely sequence of hybrid states or modes. This step is called *hybrid estimation*. Second, based on this estimate, compute a set of

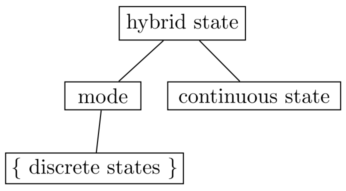


Figure 7.7.: A hybrid state consists of a discrete mode and a continuous state. The discrete mode represents a set of discrete states.

l faults $F = \{f_{i_1}, \ldots, f_{i_l}\}$. A system operating nominally is simply represented by an empty fault set F. Figure 7.6 illustrates the hybrid diagnosis process.

7.3.2.1. Hybrid Estimation and Diagnosis

Estimating the hybrid state of a system based on a hybrid model is an instance of belief state estimation. That is, we want to compute a distribution over model states. We assume that we have modeled our system with a hybrid discrete/continuous model M, and that we have observations $\mathbf{o}^{0:t}$ for t time points. Then the task of hybrid estimation is to compute $P(X \mid \mathbf{o}^{0:t})$.

X is a random variable which either represents the hybrid state or a sequence thereof (of predefined length, a time window). The hybrid state of a system is a tuple $(m, \mathbf{u})$, where m is the current discrete mode (such as ascending for a plane, or filling for a filling station currently filling a bottle) and $\mathbf{u}$ a vector of (usually) real values representing the continuous state of a system (e.g., velocity, acceleration, fill level, etc.). The mode itself typically represents a *set* of discrete states. For example, the locations in PHCA models represent modes of a system or system component. Figure 7.7 shows an overview diagram of hybrid state, mode, discrete and continuous states.

The set of possible hybrid states is defined through the hybrid model M. A well known formalism for hybrid models are hybrid automata, as introduced in [83]. Hybrid automata are a combination of finite state machines with differential equations, where the former model potential modes and transitions between them, while the latter describe the evolution of continuous variables in the model. In this work a variant of hybrid automata is used to define models. Note that the term “model” is very broadly used. In [126], for example, a sequence of modes is called “model”. In our case, “model” refers to a hybrid automaton type of state machine (A PHCA for example), if not stated otherwise.

Systems that lend themselves to hybrid modeling are called hybrid systems. In such systems the two major issues of uncertainty and complexity are just as pressing as in discrete models, often even more so. Uncertainty occurs in the observations made and during the discrete/continuous evolution of the system. This is especially a problem if faults manifest themselves through small deviations of continuous signals. The problem here is to filter out noise as well as (possibly nondeterministic) mode switches. Complexity reveals itself through the number of possible modes, which grow quite large quickly [87]. This poses the problem of making hybrid state estimation tractable by approximating the distribution $P(X \mid \mathbf{o}^{0:t})$. One well known approach to approximation is to compute and track the k most probable hybrid states or sequences of hybrid states [87]:

$$\left(\underset{x}{\text{argmax}}[j] P(X = x \mid \mathbf{o}^{0:t}) \right)_{j \in \{1,\ldots,k\}}$$

The hybrid diagnosis task is, given a hybrid estimate, to identify a minimal set of l faults $F = \{f_{i_1}, \ldots, f_{i_l}\}$ which explain off-nominal behavior, e.g., signal-deviations. In the simpler case the model specifies fault modes, which then can be read from the estimated hybrid state or state sequence. In case a given model does not explicitly specify fault modes the task becomes more complicated. However, since in this work we assume fault modes to be specified in form of special PHCA locations, we focus on the former case.

7.3.2.2. State-of-the-Art in Hybrid Estimation and Diagnosis

Hybrid systems have long been at the center of interest in model-based verification and increasingly gain attention in areas such as model-based diagnosis, or in general where system models guide problem solving. Henzinger introduced the formalism of hybrid automata as a modeling framework for hybrid systems [83], which is nowadays a widely accepted standard not only in hybrid systems verification. Recent advances in modeling concurrent stochastic hybrid systems have been published by Alur et al. [6, 19].

The hybrid diagnosis problem was formulated [126], which is to identify the most probable sequence of modes over time and a set of associated candidate qualitative diagnoses that are consistent with that sequence and available observations. A candidate qualitative diagnosis defines a fault mode, the time of the fault's occurrence and parameters associated with that mode. Parameters specify the continuous behavior in any given mode. As solution approach, the authors propose a two-step approach: in a first step techniques from qualitative reasoning are exploited to identify an initial set of candidate qualitative diagnoses with their associated mode sequences. This greatly reduces the search space of possible mode sequences and parameters. Then, these sequences are refined using parameter estimation and model fitting techniques ("Model" here referring to a mode sequence).

More recent works that combine qualitative with quantitative reasoning for hybrid diagnosis are [43] and [44]. In [43] the authors present a qualitative approach to isolate multiple faults in continuous systems based on analyzing the deviations of measurements. They specifically address the problem of faults masking or compensating each others' effects. In [44] the same authors use a *qualitative abstraction* of measurement deviations for hybrid diagnosis.

An often employed class of approximate algorithms for state estimation problems with hybrid models is particle filtering [55]. The general approach works iteratively from time point to time point, computing a distribution over system states for the given time point from observations for the current time point and the distribution from the previous time point. Unlike other such recursive methods, e.g. hidden Markov model filtering, the distribution is approximated based on sampling from the previous distribution. Each sample forms a particle that represents a specific potential system state. A second sampling step focuses the particles on those areas of the state space that assign the highest likelihood to the observations of the current time step. Such a particle filtering algorithm has been applied to hybrid diagnosis in [105]. Specifically, the authors address a hybrid estimation problem for systems with distributed resources. They introduce an approach that uses two different models for the same system: a more abstract, qualitative model, and a finer, hybrid model.

A work that addresses the scalability issues for systems with a large number of possible modes is [87]. The authors propose a method that tracks only the k most likely trajectories of a system, based on system models that are related to PHCA. They combine them with linear continuous models and use Kalman filters to track continuous behavior with the latter. Additionally, the authors address the problem of unknown failures or events by introducing modes that do not impose any restrictions on model variables, i.e. "everything is possible" in these modes.

The hitherto referenced works have addressed problems involving hybrid models with methods and algorithms that more or less work directly on the hybrid models. An entirely different approach is to convert a hybrid model into a purely discrete one and then apply methods that work with discrete models. We base our own work on such a discretization step, which was introduced together with the concept of stochastic automata in [115]. The authors formulate a method that automatically generates probabilistic state machines (which are discrete in nature) from hybrid system dynamics. A more detailed and recent description of this method can be found in [23, chapter 9]. In our approach for hybrid PHCA, we modify this method by combining it with reachability analysis from verification, implemented in a freely available tool called PHAVer [67]. The advantage over methods like the ones described above is that we can readily combine this with the constraint optimization used in our model-based diagnosis approach for plan assessment, described in section 5.2.

The works referenced so far (and also this work) make the assumption that certain technical systems are better suited for hybrid modeling than for discrete modeling. In [76] the very question of suitability for one or the other is addressed. That is, the work asks which systems are "not suitable" for discrete modeling, and thus should be modeled "hybridly", i.e. with parts being described with continuous variables and differential equations.

7.3.3. Compiling Hybrid PHCA to Discrete Constraint Nets

As an approach to compute solutions to the plan assessment problem for hybrid models, we propose to convert the continuous parts of a hybrid system model, described with linear ODEs, to discrete Markov chains. These are then recombined with the discrete rest of the system model, rendering it a completely discrete model. Our approach integrates the discretization method from [115] with the earlier described translation to constraint nets, allowing for an approach to hybrid plan assessment that builds on the constraint optimization approach described in 5.2. The hybrid models that form the input for our approach are described as hybrid PHCA, which we describe next.

7.3.3.1. Hybrid PHCA

We define Hybrid Probabilistic Hierarchical Constraint Automata (HyPHCA) in style of the well known hybrid automata [83], i.e. a combination of state machines with differential equations. They combine the modeling power of PHCA [162] with linear ODEs. Linear ODEs are a widely used standard for modeling continuous system evolution. A linear ODE is of the form $\dot{U} = aU + b$, where U is some real-valued variable and a, b are constants. When modeling a system, typically a vector of real-valued variables $\mathbf{U} = [U_1, \ldots, U_n]^T$ describes the system's continuous state. Its time-continuous evolution is consequently expressed with a set of linear ODEs, which written in vector form yields the equation $\dot{\mathbf{U}} = \mathbf{A}\mathbf{U} + \mathbf{b}$, with vectors $\dot{\mathbf{U}} = [\dot{U}_1, \ldots, \dot{U}_n]^T$ and $\mathbf{b} = [b_1, \ldots, b_n]^T$ and the $n \times n$-matrix $\mathbf{A}$ for all coefficients a. Following our notations in this work, we will use indexed capital letters to denote real-valued variables, e.g. $U_1, U_2, \ldots, U_i, \ldots$.

Definition 4. Hybrid Probabilistic Hierarchical Constraint Automata (HyPHCA) A HyPHCA is a tuple $HA = \langle \Sigma, P_\Xi, \Pi, \mathcal{U}, \mathcal{C}, \mathcal{C}_\mathcal{U}, \mathcal{F}, P_T, s_U^{t_0}, \triangle t \rangle$ where

- P_T, Σ, P_Ξ, Π and $\mathcal{C}$ are analog to the PHCA definition.

- $\mathcal{U} = U \cup \dot{U} \cup U'$ is a set of real-valued variables $U = \{U_1, \ldots, U_n\}$, their first derivatives $\dot{U} = \{\dot{U}_1, \ldots, \dot{U}_n\}$ and a set of reset variables $U' = \{U'_1, \ldots, U'_n\}$ representing values of U right after discrete transitions.

- $\mathcal{C}_{\mathcal{U}}$ is a set of constraints either over real-valued state variables U or real-valued reset variables U'. Constraints over U take one of the forms $U_i = a$, $U_i < a$, $U_i \leq a$, $U_i > a$, $U_i \geq a$. Constraints over U' take the form $U'_i = a$. The latter assignment means to reset the *state* variable U_i with constant a at the next time step. The function $f_{\mathcal{C}_U} : \Sigma \cup \mathcal{T} \to \mathcal{C}_{\mathcal{U}}$ associates locations or transitions with constraints from $\mathcal{C}_U$. Usually, we write $\mathcal{C}_U(l)$ instead of $f_{\mathcal{C}}(l)$ to denote the constraint for some location $l \in \Sigma$, and likewise $\mathcal{C}_U(\tau)$ instead of $f_{\mathcal{C}}(\tau)$ to denote the guard for some transition $\tau \in \mathcal{T}$.

- $\mathcal{F}$ is the set of constraints over real-valued state variables and their derivatives $U \cup \dot{U}$ in the form of *linear ordinary differential equations.* The function $f_{\mathcal{F}} : \Sigma \to \mathcal{F}$ associates a location with a set of such differential equations, $f_{\mathcal{F}}(l) \equiv \dot{\mathbf{U}}_l = \mathbf{A}_l \mathbf{U}_l + \mathbf{b}_l$. As with $\mathcal{C}$, we usually write $\mathcal{F}(l)$ instead of $f_{\mathcal{F}}(l)$ to denote the set of differential equations for some location $l \in \Sigma$. The *continuous flow* $f_{\mathbf{U}_l} : \mathbb{R} \to \mathbb{R}^n$ of a location l is given as a solution of the differential equations (n is the number of state variables in vector $\mathbf{U}_l$).

- $s_U^{t_0}$ is an initial assignment to the continuous state variables.

- $\triangle t$ is the constant time interval between two consecutive transitions.

Guards for transitions of a HyPHCA may now include continuous variables from U and U'. Like PHCA, HyPHCA are defined for discrete time steps. PHCA do not explicitly specify a clock interval. Since with HyPHCA we consider continuous evolution over time, we have to make that interval explicit in the form of $\triangle t$. Finally, when creating a HyPHCA model on must pay attention to a number of things:

- $\mathcal{F}(l)$ must be defined such that locations that may be marked in parallel do not share continuous variables.

- Guard constraints must be defined such that events/transitions occur only aligned with the discrete time points $t_0 + k\triangle t$.

- If transition guards can become entailed because of continuous states (e.g. $U = 0$), then the respective continuous variables must be reset as needed.

Algorithm 7.2 HyPHCA unroll function that creates all possible sequences of states of a given HyPHCA model. The operator $\smile$ concatenates tuples.

```
1: function unrollHyPHCA(HA, $\mathbf{c}^{t_0:t_N}$, $\triangle t$)
2:     $S \leftarrow \{(s_U^{t_0}, m^{t_0}) \mid m^{t_0}$ is an initial marking of HA$\}$
3:     for $i = 1 .. N$ do
4:         for $\mathbf{s}^{t_0:t_{i-1}} \in S$ do
5:             $\langle (s_U^{t_0}, m^{t_0}), \ldots, (s_U^{t_{i-1}}, m^{t_{i-1}}) \rangle \leftarrow \mathbf{s}^{t_0:t_{i-1}}$
6:             $Q \leftarrow \{\textsc{Step}(m^{t_{i-1}}, M_{\text{PHCA}}, \mathbf{c}^{t_i}, T) \mid T \in P_T\}$
7:             $S' \leftarrow \{\mathbf{s}^{t_0:t_{i-1}} \smile \langle (\textsc{executeFlow}(s_U^{t_{i-1}}, m^{t_{i-1}}, m^{t_i}, \triangle t), m^{t_i}) \rangle \mid m^{t_i} \in Q\}$
8:         end for
9:         $S \leftarrow S'$
10:     end for
11:     return $S$
12: end function
13: function executeFlow($s_U^{t_{i-1}}$,$m^{t_{i-1}}$,$m^{t_i}$,$\triangle t$)
14:     $L \leftarrow \{(source(\tau), \tau) \mid \tau$ is a transition between locations $l_1 \in m^{t_{i-1}}$ and $l_2 \in m^{t_i}\}$
15:     for $(l, \tau) \in L$, $\mathcal{F}(l) \neq \emptyset$ do
16:         $\mathbf{u}_l^{t_{i-1}} \leftarrow$ sub-assignment from $s_U^{t_{i-1}}$ for location $l$
17:         $\mathbf{u}_l^{t_i} \leftarrow (u_1^{t_i}, \ldots, u_{n_l}^{t_i}) \leftarrow f_{\mathbf{U}_l}(\mathbf{u}_l^{t_{i-1}}, \triangle t)$
18:         if scope of guard $\mathcal{C}(\tau)$ contains reset variables $\{U'_j \in U'\}$ then
19:             for all $j$ do                    ▷ Reset values override values from flow.
20:                 $\mathbf{u}_l^{t_i} \leftarrow (u_1^{t_i}, \ldots, u'_j, \ldots, u_{n_l}^{t_i})$
21:             end for
22:         end if
23:         $s_U^{t_i} \leftarrow s_U^{t_i} \smile \mathbf{u}_l^{t_i}$
24:     end for
25:     return $s_U^{t_i}$
26: end function
```

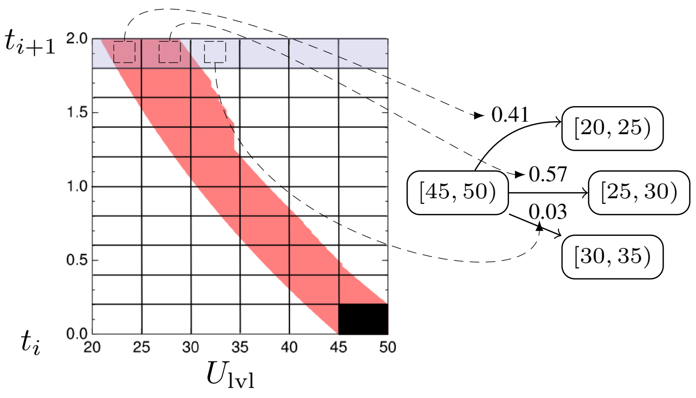


Figure 7.8.: Reachable set R_{start} for $\dot{U}_{\mathrm{lvl}} = -\mathrm{fR} * U_{\mathrm{lvl}}$ starting from the marked grid cell G_{start,t_i}. Right: the derived PHCA $A^{\mathrm{Markov}}_{\triangle t}$.

Next we define HyPHCA states and trajectories with an unrolling function that extends the PHCA semantic specified in algorithm 4.1 in section 4.

This unroll function for HyPHCA shown as algorithm 7.2 extends PHCA UNROLL in that it additionally executes the continuous flows of locations that have one. It thus creates for a given HyPHCA HA and given commands the set of all possible trajectories of length N. This set is non-empty as long as the differential equations have a solution (which is the case for linear ODEs). Each trajectory $\theta_{\mathrm{HA}} = \langle (s_U^{t_0}, m^{t_0}), \ldots, (s_U^{t_N}, m^{t_N}) \rangle$ is a sequence of HyPHCA states $s^t = (s_U^t, m^t)$, where $s_U^t \in \mathbb{R}^{|U|}$ is an assignment to all variables $u \in U$ at time t, called continuous state, and $m^t \in \mathcal{M}$ a marking analogous to PHCA markings (with $\mathcal{M} \subseteq 2^{\Sigma}$ the set of all markings).

The function executeFlow uses the continuous flow $f_{\mathbf{U}_l}$ of locations l (in case they have associated differential equations $\mathcal{F}(l)$) to map the current continuous state to the continuous state for the next time point (line 16). This happens synchronously with the discrete transitions of the system. In case transition guards reset continuous state variables via constraints of the form $U' = a$, the reset values a override respective values that result from the continuous flow (lines 18 – 20). The operator $\smile$ in lines 7 and 22 concatenates tuples.

7.3.3.2. From Hybrid to Abstract Discrete Models

The core idea behind our translation approach is to abstract continuous dynamics with Markov chains. For example, consider the case when the silo of our filling station is being emptied (see figure 7.8). Its dynamics over a period of time $\triangle t$ define a continuous trajectory in the continuous space of the model. In a first step this space is mapped onto a discrete space, a grid in our case. The initial fill level now falls into one of the grid cells, the final into another. These grid cells represent the abstracted initial and final fill levels respectively. We can see these grid cells as connected locations of an automaton, connected because the dynamics dictate that from the given initial fill level we will arrive at the final fill level.

What if the initial fill level was only slightly different, such that we would start in the same abstract location? The continuous trajectory for that fill level might end in a different cell, just next to the target cell for the previous initial fill level. So at the abstract, grid cell level, the initial cell must also be connected to a different target cell. If we only know about the abstract initial cell, we are uncertain about the target cell that we will end up in. At this point, probability is used as a means of abstractly representing this uncertainty. To determine these probabilities, we look at where the continuous trajectories for *all* initial fill levels within a single grid cell lead to. All the cells being "hit" by these trajectories become target locations for the corresponding location representing the abstract initial fill level. From the "area" which the trajectories cover within a specific target cell we derive the probability for the transition leading to the corresponding target location. Of course, we cannot really handle all these trajectories one by one, as there are uncountably many. The approach is to use over-approximations that enclose the set of all trajectories.

The mentioned automaton that is being created encodes a Markov chain, which in turn is a probabilistic abstraction of the continuous behavior. This automaton is a special PHCA, so the idea suggests itself to replace the locations with continuous dynamics in the original HyPHCA with locations that contain the abstracted dynamics as composite sub-location, thus forming a completely discrete PHCA. However, two non-trivial issues in conjunction with the hierarchical nature of PHCA complicate the matter:

1. One problem is that the PHCA formalism doesn't allow transitions originating from a composite location $l \in \Sigma_c$, they must originate from primitive locations within l. If discretized continuous dynamics, encoded as a composite location, are to be inserted into a location that is primitive in the original HyPHCA, its outgoing transitions must be adapted. That is, they must receive as sources primitive locations of the composite location encoding the dynamics.

2. A second, more demanding problem is this: Let's assume we simply embed continuous dynamics of some variable U_i, discretized as a sub-PHCA A_{U_i}, into a location l, rendering this a composite location. The PHCA marking semantics demand that sub-locations of l can only be marked when l itself is marked. Let's further assume

that l is marked at t_j and that a transition occurs such that l is not marked at t_{j+1}. Specifically, all locations of A_u are unmarked at t_{j+1}. The problem is to determine the value of variable X_{U_i} (the discretization of U_i) for t_{j+1}. It should be a result of the dynamics encoded in A_{U_i}, which however requires one of its locations to be marked.

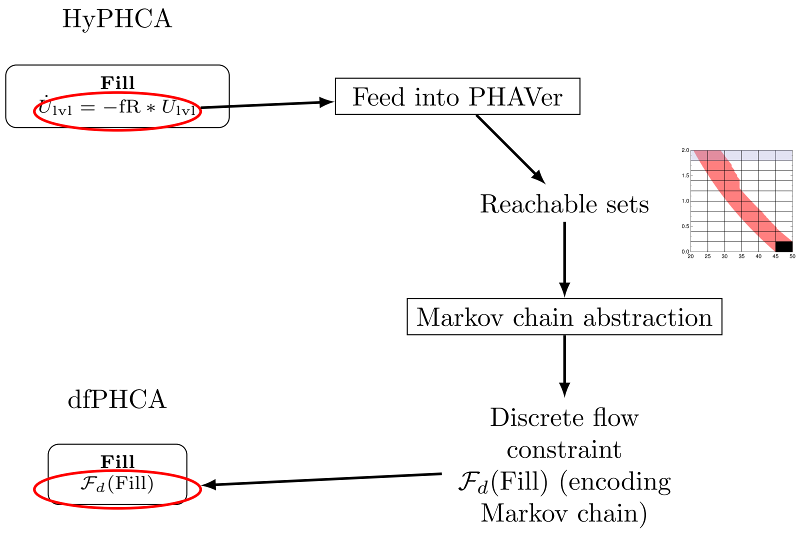
Both of these issues would result in very complicated translation steps, possibly creating many additional transitions. Therefore, we chose a simpler approach, which involves an intermediate construct called *discrete flow* PHCA (dfPHCA). This construct represents discretized continuous dynamics explicitly as so-called discrete flow constraints. If we see the discretized dynamics as special PHCA, these constraints encode only the distribution P_T over transition functions of this PHCA. Their main advantage is that they can be encoded as soft constraint functions with only minor adaptation, which yields a very compact encoding.

Our approach is thus to convert each continuous flow occurring in some location l in a HyPHCA into a discrete flow of a corresponding location in a dfPHCA. The process is illustrated in figure 7.9. The dfPHCA is then directly translated to a soft constraint net, using a modified version of the translation described in [129]. From here, the approach to plan assessment as described in 5.2.3 takes over. Next, we define discrete flow PHCA.

Discrete Flow PHCAs The evolution of a continuous variable $u \in U$ in between two time points t_i and t_{i+1} is mapped to a discrete transition between the quantized states of u at time t_i and time t_{i+1} (given a predefined quantization). These discrete evolutions are encoded as *discrete flow constraints* of a discrete flow PHCA (dfPHCA).

Definition 5. Discrete flow PHCA A tuple $A_{\text{df}} = \langle \Sigma, P_\Theta, \Pi, \Pi_{\mathcal{U}}, \mathcal{C}, \mathcal{F}_d, P_T, s^{t_0}_{\Pi_U}, \triangle t \rangle$ is called a dfPHCA (parameterized with fixed-length time interval $\triangle t$), where:

- $\Sigma, P_\Theta, \Pi, P_T, \triangle t$ are analog to the HyPHCA definition.
- $\Pi_{\mathcal{U}} = \Pi_U \cup \Pi_{U'}$ is a set of finite domain variables. It is analogous to $\mathcal{U}$ of a HyPHCA, with the exception that no analog for the derivatives exists.
- $\mathcal{F}_d$ is the set of *discrete flow constraints*. A discrete flow constraint encodes the discrete abstraction of some continuous flow constraint as probability distribution P_T over transition functions $T : \Sigma_A \to 2^{\Sigma_A}$ of a Markov chain $A^{\text{Markov}}_{\triangle t}$. Σ_A is the set of possible states of $A^{\text{Markov}}_{\triangle t}$. The function $f_{\mathcal{F}_d} : \Sigma \to \mathcal{F}_d$ associates locations with constraints in $\mathcal{F}_d$ over the discrete flow variables of these locations. We usually will write $\mathcal{F}_d(l)$ instead of $f_{\mathcal{F}_d}(l)$.



HyPHCA
Fill
$\dot{U}_{\mathrm{lvl}} = -\mathrm{fR} * U_{\mathrm{lvl}}$
Feed into PHAVer
Reachable sets
Markov chain abstraction
Discrete flow constraint $\mathcal{F}_d$(Fill) (encoding Markov chain)
dfPHCA
Fill
$\mathcal{F}_d$(Fill)

Figure 7.9.: Illustration of the discretization process: the continuous dynamics of location Fill, described by a differential equation, is fed into the verification tool PHAVer, which conservatively abstracts the dynamics by computing reachability sets, i.e. polyhedrons that enclose all reachable continuous states. Then, the Markov chain is generated from these reachability sets and the discrete flow constraint is derived.

- $\mathcal{C}$ is a set of constraints over finite domain variables Π and $\Pi_{\mathcal{U}}$.
- $s^{t_0}_{\Pi_U}$ is a combined discrete initial state for the discrete flow constraints.

The unroll semantic for dfPHCA is analogous to the one of HyPHCA, only that instead of a continuous flow its discretization is being executed. This step is encapsulated in function executeDiscreteFlow shown as algorithm 7.3. This function maps the discretized states $s^{t_{i-1}}_{\Pi_U}$ to states in the next time step. A crucial difference to executeFlow is that due to over-approximation the discrete flow may result in multiple discretized follow-up states $s^{t_i}_{\Pi_U}$. For all locations l the function iteratively applies all possible transition functions T_l of the discrete flow $\mathcal{F}_d(l)$ (line 8), resulting in partial follow-up states $\mathbf{x}^{t_i}_l$. While doing that it forms the possible combinations of the states $\mathbf{x}^{t_i}_l$ across all locations l (lines 14–18). The result is the set S of all possible follow-up states $s^{t_i}_{\Pi_U}$.

With executeDiscreteFlow in place of executeFlow, algorithm 7.2 creates for a given dfPHCA all dfPHCA trajectories $\theta = \{s^{t_i}, s^{t_{i+1}}, \ldots, s^{t_{i+N}}\}$ of length N, with dfPHCA states $s^{t_i} = (s^{t_i}_{\Pi_U}, m^{t_i})$. Here $s^{t_i}_{\Pi_U}$ is an assignment of values to discretized continuous variables $X_u \in \Pi_U$ at time t_i, and m^{t_i} a marking analogous to PHCA states.

To avoid confusion when denoting trajectories of a particular formalism, we write θ_x with $x = A_{\mathrm{df}}, HA$ for dfPHCA and HyPHCA trajectories, respectively.

Algorithm 7.3 Function to execute the discrete flows of a dfPHCA.

```
1: function EXECUTEDISCRETEFLOW($s_{\Pi_U}^{t_{i-1}}$,$m^{t_{i-1}}$,$m^{t_i}$)
2:     $L \leftarrow \{(source(\tau), \tau) \mid \tau$ is a transition between locations $l_1 \in m^{t_{i-1}}$ and $l_2 \in m^{t_i}\}$
3:     $S \leftarrow \emptyset$
4:     for $(l, \tau) \in L$ do
5:         $S_l \leftarrow \emptyset$
6:         for all transition functions $T_l$ of discrete flow $\mathcal{F}_d(l)$ do
7:             $\mathbf{x}_l^{t_{i-1}} \leftarrow$ sub-assignment from $s_{\Pi_U}^{t_{i-1}}$ for location $l$
8:             $\mathbf{x}_l^{t_i} \leftarrow (x_1^{t_i}, \ldots, x_{n_l}^{t_i}) \leftarrow T_l(\mathbf{x}_l^{t_{i-1}}, \mathsf{true})$
9:             if scope of guard $\mathcal{C}(\tau)$ contains reset variables $\{X_j' \in \Pi_{U'}\}$ then
10:                for all $j$ do                                   ▷ Reset values override.
11:                    $\mathbf{x}_l^{t_i} \leftarrow (x_1^{t_i}, \ldots, x_j', \ldots, x_{n_l}^{t_i})$
12:                end for
13:            end if
14:            if $S = \emptyset$ then          ▷ Concatenate possible discrete flow states at $t_i$.
15:                $S_l \leftarrow S_l \cup \{\mathbf{x}_l^{t_i}\}$
16:            else
17:                $S_l \leftarrow S_l \cup \{s_{\Pi_U}^{t_i} \smile \mathbf{x}_l^{t_i} \mid s_{\Pi_U}^{t_i} \in S\}$
18:            end if
19:        end for
20:        $S \leftarrow S \cup S_l$
21:    end for
22:    return $S$
23: end function
```

Converting HyPHCAs to Discrete Flow PHCAs The conversion is illustrated in figure 7.10. Before we go into the details of this process, we define further required entities. For a transition τ we denote with source(τ) and dest(τ) its source and destination location, and with guard(τ) its guard constraint. Let $HA = \langle \Sigma, P_\Xi, \Pi, \mathcal{U}, \mathcal{C}, \mathcal{C}_\mathcal{U}, \mathcal{F}, P_T, s_U^{t_0}, \triangle t \rangle$ be an arbitrary HyPHCA. $G_{\mathbb{R}^{|U|}} = \{G_\lambda\}$ denotes a set of disjunct grid cells, also called quantization cells, partitioning the continuous state space of HA: $\bigcup_\lambda G_\lambda = \mathbb{R}^{|\mathcal{U}|}$. We call this set the quantization of the continuous space of HA. Let $A = A_{\mathrm{df}} = \langle \Sigma, P_\Xi, \Pi, \Pi_\mathcal{U}, \mathcal{C}, \mathcal{F}_d, P_T, s_{\Pi_U}^{t_0}, \triangle t \rangle$ be the dfPHCA resulting from converting HA. We will refer to elements that both of the respective automata have, like Σ and P_Ξ, with the notation $HA.\Sigma$ and $A.\Sigma$, $HA.P_\Xi$ and $A.P_\Xi$, etc.

The conversion of PHCA elements, that is locations, initial probability distributions, discrete variables and transition probability distributions is straight forward: $A.\Sigma = HA.\Sigma$, $A.P_\Xi = HA.P_\Xi$, $A.\Pi = HA.\Pi$ and $A.P_T = HA.P_T$.

Based on the quantization $G_{\mathbb{R}^{|U|}}$ the real-valued variables U and U' of HA are converted to variables with finite domains. The grid cells $G_\lambda \in G_{\mathbb{R}^{|U|}}$ are mapped onto intervals of the variables in U and U'. Index sets of these intervals then form the domains of the discretized, finite domain variables Π_U and $\Pi_{U'}$. That is, the values of, for example, a variable $X_{U_i} \in \Pi_U$ represent intervals of corresponding variable $U_i \in U$. Discrete versions of the derivatives $\dot{U}$ are not needed, therefore $\Pi_\mathcal{U} = \Pi_U \cup \Pi_{U'}$ forms the discrete counterpart for $\mathcal{U}$. The initial state $s_U^{t_0}$ is converted to the vector of index values $s_{\Pi_U}^{t_0}$, which identifies the grid cell that encloses $s_U^{t_0}$. The finite domain constraints of A are created as $A.\mathcal{C} = HA.\mathcal{C} \cup \mathrm{conv}(HA.\mathcal{C}_\mathcal{U})$. The function conv, provided by the modeler, maps the arithmetic constraints to corresponding finite domain constraints.

If a primitive location $l \in HA.\Sigma$ has differential equations $\mathcal{F}(l)$, it is converted to a discrete flow constraint $\mathcal{F}_d(l)$, which is added to the corresponding location $l \in A.\Sigma$. The discrete flow constraint encodes the abstracted continuous flow by directly relating variables $X_{U_j} \in A.\Pi_U$ for two time points t_i and t_{i+1}. We will describe the process of creating $\mathcal{F}_d(l)$ from $\mathcal{F}(l)$ in detail shortly.

A discrete flow constraint $\mathcal{F}_d(l)$ of some location l may conflict a guard constraint resetting the respective variables. The semantic demands in this case that the guard takes precedence over $\mathcal{F}_d(l)$ (see lines 9–13 in algorithm 7.3). An example is the guard g in figure 7.10.

Discrete Abstraction of Continuous Flow Let l be a location of some HyPHCA HA with associated differential equations $\mathcal{F}(l)$. The evolution of l's continuous variables $U_i \in U$ in between two time points t_i and t_{i+1} is mapped to discrete, unguarded probabilistic

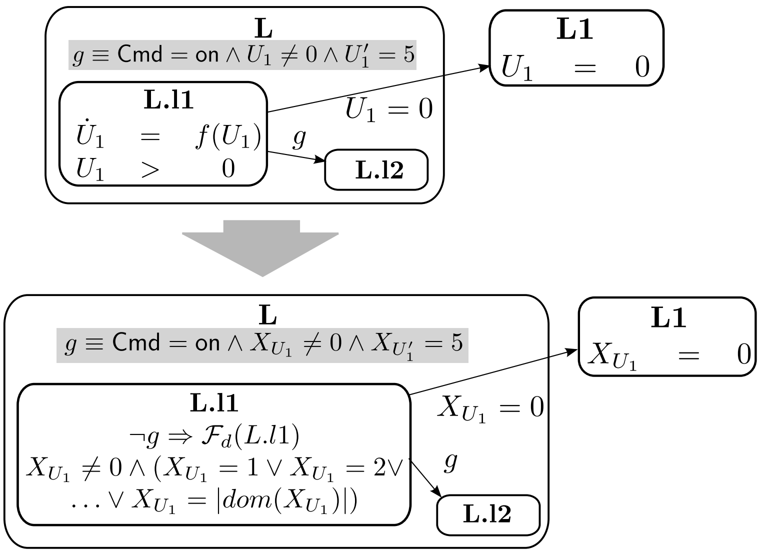

L
$g \equiv \mathsf{Cmd} = \mathsf{on} \wedge U_1 \neq 0 \wedge U_1' = 5$
L.l1
$\dot{U}_1 = f(U_1)$
$U_1 > 0$
$U_1 = 0$
g
L.l2
L1
$U_1 = 0$
L
$g \equiv \mathsf{Cmd} = \mathsf{on} \wedge X_{U_1} \neq 0 \wedge X_{U_1'} = 5$
L.l1
$\neg g \Rightarrow \mathcal{F}_d(L.l1)$
$X_{U_1} \neq 0 \wedge (X_{U_1} = 1 \vee X_{U_1} = 2 \vee \ldots \vee X_{U_1} = |dom(X_{U_1})|)$
$X_{U_1} = 0$
g
L.l2
L1
$X_{U_1} = 0$

Figure 7.10.: HyPHCA (above) is converted to a dfPHCA (below).

transitions between locations of a special PHCA $A^{\text{Markov}}_{\triangle t}$, encoded in $\mathcal{F}_d(l)$. It has only primitive locations, corresponding to grid cells of $G_{\mathbb{R}^{|U|}}$, and represents a Markov chain that conservatively approximates the continuous evolution. Figure 7.8 (right side) shows such a special PHCA.

Let $\Pi_{\mathcal{U}}$ be the discrete variables of the dfPHCA that results from converting *HA*. The grid cells $G_\lambda \in G_{\mathbb{R}^{|U|}}$ that correspond to values of the discrete variables in $\Pi_{\mathcal{U}}$ form locations of $A^{\text{Markov}}_{\triangle t}$. We determine the transitions between these locations, or more precisely their probabilities, with a conservative estimate based on the geometric shape of the grid cells and the *reachable set* of grid cells. The reachable set of a grid cell is the set that contains all points reachable from points in that grid cell via continuous HyPHCA trajectories, for some predefined time interval. We adapt the geometric abstraction method introduced in [115] and further described in [23, chapter 9].

We recap this method shortly. The quantized state space is combined (via cartesian product) with a partition of the time interval $[t_i, t_{i+1}]$, with $\triangle t = t_{i+1} - t_i$. The finer this partition, the more accurate the computed probabilities will be. Source locations of transitions of $A^{\text{Markov}}_{\triangle t}$ are associated with quantization cells within the first partition element in $[t_i, t_{i+1}]$ and destination locations with the last. Currently, we ignore the elements in between. Let now G_{start,t_i} be the quantization cell of start location l_{start} and $G_{\lambda,t_{i+1}}$ the cells of all possible destination locations l_λ (with λ indexing cells and locations). The reachable set R_{start} is computed as a geometric shape, which is as small as possible yet guaranteed to include all continuous states reachable from G_{start,t_i} within

$[t_i, t_{i+1}]$. Now the probabilities for the transitions from l_{start} to destination locations l_λ are computed as

$$P(l_\lambda \,|\, l_{\text{start}}) = \frac{V(G_{\lambda,t_{i+1}} \cap R_{\text{start}})}{V(\bigcup_x G_{x,t_{i+1}} \cap R_{\text{start}})}.$$

Function $V()$ returns the volume of the given set. The complete process is illustrated in figure 7.8.

Our adaptation mainly lies in mapping the various computation steps to appropriate, publicly available tools and combining them into a single conversion process. We have chosen tools that allow for fast yet general computation: The set R_{start} is computed via reachability analysis implemented in the tool PHAVer [67]. PHAVer accepts hybrid automatons with linear ODEs as models, and returns R_{start} as general polyhedra. For the subsequent geometric operations $\cup$ and $\cap$ we use general polyhedron algorithms provided by the Parma polyhedra library [7]. The volumes of polyhedra, i.e. $V()$, are computed with the open source tool Vinci by the authors of [33].

An advantage of this component-based approach is its flexibility. For example, for reachability analysis different approaches can be employed. Regarding abstraction of hybrid models, we can build on a lot of related work in the area of automated verification of model properties. In particular, Stursberg et al. combine Markov chains abstracting continuous behavior with a more advanced reachability analysis in [5]. They address the problem of online verification of properties such as that the planned path of a cognitive vehicle doesn't cross the path of another vehicle.

Another example is the choice of the state space quantization. Since we employ very general geometric methods, many different quantizations could be used, for example to address the problem of spurious solutions/trajectories. A too coarse state space quantization can lead to feasible abstract trajectories that, when projected to the hybrid space, contain unreachable states (the experimental results demonstrate such a situation in section 7.3.4). Currently, the right number of partitions must be determined empirically. Hofbaur and Rienmüller introduced a method to intelligently quantize the continuous state space based on qualitative properties of piecewise affine systems [85]. The method might be a useful extension to our approach as it automatically chooses a good number of partition elements, balancing precision of the abstraction against tractability, and reduces the number of spurious solutions.

dfPHCAs as Conservative Abstraction We tested the correctness of our approach empirically by generating a discretized model for our example HyPHCA. The reachable set and Markov chain probabilities in figure 7.8 have been computed with our implementation.

An open issue is the theoretical correctness of the conservative abstraction, i.e. that a dfPHCA indeed conservatively captures all behaviors of the HyPHCA it was created from.

We do not provide a proof here as this would go beyond this work's scope. However, we prepare the ground for future work by presenting the proposition that formalizes the correctness property.

For abstraction correctness, it must be shown that a dfPHCA A_{df}, generated as described above from a HyPHCA *HA*, is a conservative abstraction in terms of the probabilities of system trajectories. The idea is to show that all the continuous trajectories of *HA* that are subsumed by a trajectory of A_{df} have a probability less than or equal to the probability of the a trajectory of A_{df} (given observations and commands):

Definition 6. (Set of abstracted HyPHCA trajectories) Let $G : D_{\Pi_U} \to G_{\mathbb{R}^{|U|}}$ be a function that maps assignments to discretized continuous variables Π_U to grid cells $G_\lambda \in G_{\mathbb{R}^{|U|}}$. Let $\theta_{A_{\mathrm{df}}}$ be a trajectory of A_{df}. Then $\chi(\theta_{A_{\mathrm{df}}}) := \{\theta_{\mathrm{HA}} | \forall\, t_i : (s_U^{t_i}, m^{t_i}) \in \theta_{\mathrm{HA}} \wedge (\hat{s}_{\Pi_U}^{t_i}, \hat{m}^{t_i}) \in \theta_{A_{\mathrm{df}}} \Rightarrow m^{t_i} = \hat{m}^{t_i} \wedge s_U^{t_i} \in G(\hat{s}_{\Pi_U}^{t_i})\}$ [4] is the set of all HyPHCA trajectories abstracted by $\theta_{A_{\mathrm{df}}}$.

Proposition 2. *Let $\mathbf{o}^{0:t}, \mathbf{c}^{0:t}$ be arbitrary finite sequences of observations $\mathbf{o}^{t_i} \in D_{\Pi_{\mathrm{Obs}}}$ and commands $\mathbf{c}^{t_i} \in D_{\Pi_{\mathrm{Cmd}}}$ and $\langle t_i \rangle$ the corresponding sequence of time points. Then, for a trajectory $\theta_{A_{\mathrm{df}}}$ consistent with $\mathbf{o}^{0:t}, \mathbf{c}^{0:t}$ (i.e. $P(\theta_{A_{\mathrm{df}}} \,|\, \mathbf{o}^{0:t}, \mathbf{c}^{0:t}) > 0$), the following holds:*

$$\left(\int_{\theta_{\mathrm{HA}} \in \chi(\theta_{A_{\mathrm{df}}})} f_{\mathrm{HA}}(\theta_{\mathrm{HA}} \,|\, \mathbf{o}^{0:t}, \mathbf{c}^{0:t}) \right) \leq P(\theta_{A_{\mathrm{df}}} \,|\, \mathbf{o}^{0:t}, \mathbf{c}^{0:t})$$

$f_{\mathrm{HA}}(\theta_{\mathrm{HA}} \,|\, \mathbf{o}^{0:t}, \mathbf{c}^{0:t})$ is the density function of a distribution over discrete-time HyPHCA trajectories, conditioned on the sequences $\mathbf{o}^{0:t}, \mathbf{c}^{0:t}$.

7.3.3.3. Encoding Discrete Flow with Soft Constraints

To encode dfPHCA as constraint nets, we extended the framework described in 5.2.2 with a soft constraint encoding of discrete flow constraints. The discrete flow constraint itself is a function mapping discrete flow variables in $\Pi_U^{t_i}$ and $\Pi_U^{t_{i+1}}$ to transition probabilities, and thus could be directly encoded as soft constraint. However, we have to regard that a discrete flow is not always active. A discrete flow is active if and only if its associated location is marked and if it is not in conflict with a guard that influences the same discretized variables. Remember that in this latter case the guard would take precedence over the discrete flow in determining the value of said variables. To determine when

[4] The hat $\hat{}$ is used to differentiate the HyPHCA state $(s_U^{t_i}, m^{t_i})$ from the dfPHCA state $(\hat{s}_{\Pi_U}^{t_i}, \hat{m}^{t_i})$

exactly such conflicts arise, additional constraints need to be added to the constraint net. This requires a fairly complicate encoding. A much simpler solution is possible if we can ignore the conflicts.

The conflicts between flows and guards occur if the variables U' (and their discrete counter parts $\Pi_{U'}$) are used, for example to reset a variable U_i after a transition. To avoid these variables, these kind of situations could be worked around by introducing explicit resetting locations that have behavior constraints of the form $U_i = x$. Therefore, we think avoiding variables U' altogether is a rather minor restriction, which allows us to use the simpler solution. We will, however, sketch how to extend this simple solution in order to encompass the full expressiveness of HyPHCAs.

In the simple solution, the discrete flow being active or not is formally described by

$$\forall t_i, l \text{ has } \mathcal{F}_d(l).\ X_l^{t_i} = \mathsf{marked} \Leftrightarrow X_{\mathcal{F}_d(l)}^{t_i} = \mathsf{active}$$

Since activeness depends only on location l being marked or not, we can spare the extra variable $X_{\mathcal{F}_d(l)}^{t_i}$ for encoding activeness explicitly. Therefore we create, for each discrete flow and for each time point, a soft constraint function over variables $\Pi_U^{t_i}$, $\Pi_U^{t_{i+1}}$ and $X_l^{t_i}$. If $X_l^{t_i} = \mathsf{marked}$, this function maps partial assignments to the respective probabilities of transitioning from $\Pi_U^{t_i}$ to $\Pi_U^{t_{i+1}}$, as given by the discrete flow. If unmarked, all partial assignments are mapped to 1.0, meaning that all possible transitions are allowed.

Of course this encoding of dfPHCAs leads to a certain overhead, which however is linear in the model size. For each location with discrete flow and for each time point the encoding adds an additional soft constraint. We assume that in many cases only few components have continuous behavior, which would further reduce this overhead. Also note that, just like the PHCA translation, the translation of dfPHCA to soft constraints can be done offline, as well as the discretization and Markov chain abstraction. From this point on we can proceed as described in section 5.2.3 to compute solutions to the plan assessment problem.

Finally, we describe how an advanced encoding could be developed that respects potential conflicts between flows and guards. The discrete flow now depends on its location being marked and on not being overridden by some guard, i.e. formally:

$$\forall t_i, l \text{ has } \mathcal{F}_d(l).\ X_{override(\mathcal{F}_d(l))}^{t_i} = \mathsf{false} \wedge X_l^{t_i} = \mathsf{marked} \Leftrightarrow X_{\mathcal{F}_d(l)}^{t_i} = \mathsf{active}$$

Now we have to explicitly add the variables $X_{override(\mathcal{F}_d(l))}^{t_i}$ with domain $\{\mathsf{true}, \mathsf{false}\}$ and $X_{\mathcal{F}_d(l)}^{t_i}$ with domain $\{\mathsf{active}, \mathsf{inactive}\}$, which indicate an override of a discrete flow and its activation, respectively. Hard constraints have to be added to implement the above formula.

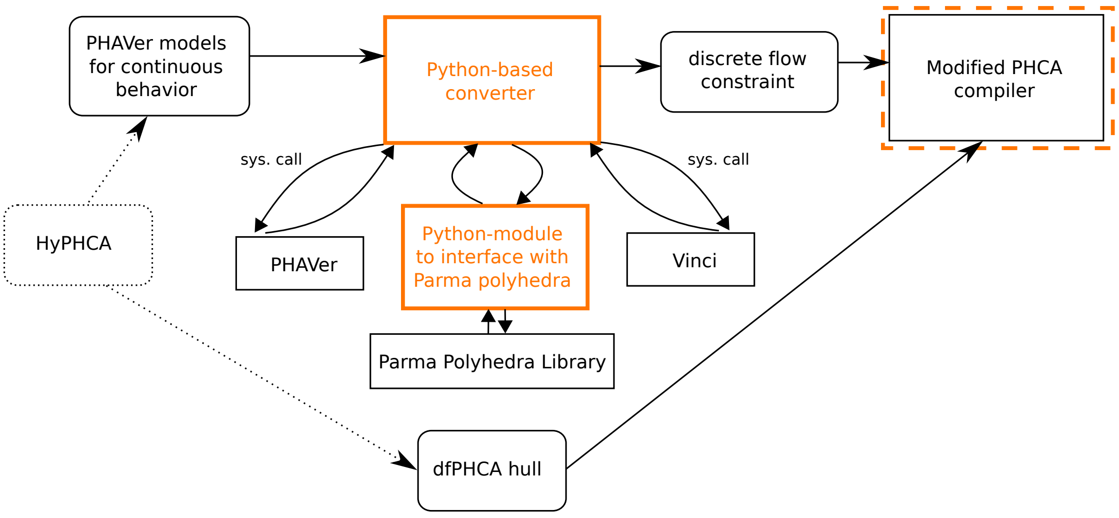


Figure 7.11.: Illustration of implemented components for the discretization process.

The soft constraints encoding the discrete flow functions would not be much different: their scope would contain $X^{t_i}_{\mathcal{F}_d(l)}$ instead of $X^{t_i}_l$, and condition $X^{t_i}_{\mathcal{F}_d(l)} = \mathsf{active}$ would replace $X^{t_i}_l = \mathsf{marked}$. Additional constraints would be needed to encode when exactly guards conflict with discrete flows, determining the value of $X^{t_i}_{override(\mathcal{F}_d(l))}$. For each guard that contains a variable $X_{U'_i}$, a constraint could be added that enforces $X^{t_i}_{override(\mathcal{F}_d(l))} = \mathsf{true}$ for the corresponding discrete flow over variable X_{U_i} if the transition of this guard is enabled.

7.3.3.4. Implementation of Conversion Process

Ideally, the conversion process would start with a HyPHCA model that is split into its continuous and discrete parts. However, since no description language for HyPHCA exist yet, we implemented a process that starts with a manually created model for the continuous behavior (a PHAVer model in this case) and a dfPHCA "hull". This "hull" is described using the existing PHCA syntax (created by the authors of [162]). The syntax does not allow to use $\Pi_{U'}$ variables, which however falls in line with the restriction that we already made in the previous section. Another restriction is that discrete versions of simple arithmetic constraints such as $U_1 \leq a$ have to be defined manually.

The continuous model part is the input for the conversion component implemented for this work. The component is implemented as set of Python modules that are used, e.g., by the scripts running our experiments. It creates discrete flow constraints for the flows described in the continuous model part, calling the employed external tools, e.g.

PHAVer, in the process. The discrete flow constraints and the dfPHCA "hull" are then fed into a dfPHCA compiler, which is a modified version of the PHCA compiler created by the authors of [129]. Figure 7.11 illustrates the components involved in the conversion process. Implementations and modifications done for this work are highlighted. A dashed line indicates that an existing component was modified for our purposes.

Our conversion component implements, in particular, three interfaces for the tools that we use to perform the various geometric computations on polyhedra (PHAVer: reachable sets; Parma polyhedra library: $\cup$ and $\cap$ on polyhedra; Vinci: volume of polyhedra). The data being passed between the tools and the converter are consequently coded polyhedra. All involved tools describe a general polyhedron with sets of convex polyhedra, which in turn are described with the linear equations that make up their boundaries. These equations take the form $0 \backsim a_0 + a_1 U_1 + \ldots + a_n U_n, \backsim \in \{\leq, =, \geq\}$. They are, however, coded differently for each tool.

The first interface defines a parser to read the output of PHAVer, the reachable sets. The second defines a native Python module, written in C, that directly calls functions of the Parma polyhedra library, which is written in C++. This spares the overhead of reading and writing to the hard disk, at the cost of a somewhat more complex implementation. The cost was limited in this case as our access to the library is very specific. Finally, the third interface specifies a writer that creates input files for Vinci, and a parser that reads its output. PHAVer, Parma polyhedra library and Vinci are available with source code on the internet[5].

The discrete flow constraints, the output of our conversion component, are stored in a custom format that is human-readable yet easy to parse. This simplified the next step, the modification of the PHCA compiler (written in C++). Here we implemented a parser for said format, and added a function that creates the soft constraints as described in section 7.3.3.3 (using the simple encoding).

7.3.4. Results of Diagnosis Experiments

We ran experiments for a scenario in which the filling station receives commands to fill two bottles. The results have been previously published in [119]. The scenario has a duration of 8 seconds with 5 time points $t_0, \ldots, t_4$ with a duration of $\triangle t = 2s$ for each step in between. The silo of the station has an initial fill level of 50 units, its fill rate is $\mathrm{fR} = 0.38s^{-1}$. After 8 seconds at t_4 the sensor indicates an empty silo.

[5] PHAVer: `http://www-verimag.imag.fr/~frehse/phaver_web/` (08.2011)
Parma polyhedra library: `http://www.cs.unipr.it/ppl/Download/` (08.2011)
Vinci: `http://www.math.u-bordeaux1.fr/~enge/software/vinci/vinci-1.0.5.tar.gz` (08.2011)

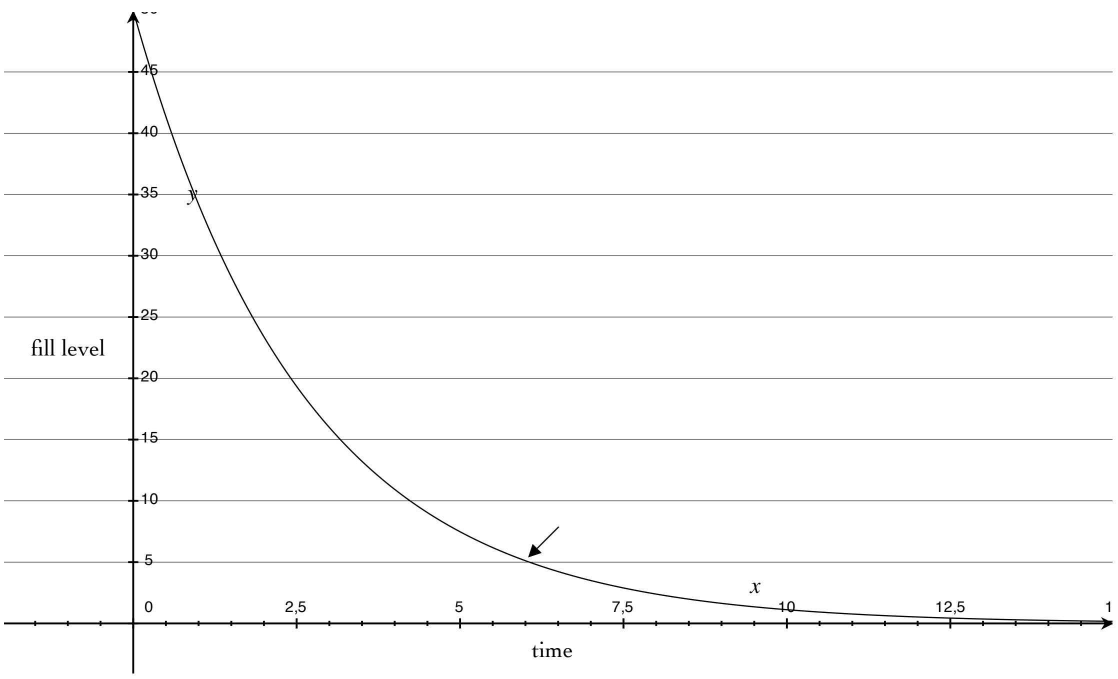


Figure 7.12.: Graph of $\dot{U}_{\text{lvl}} = -\text{fR} \cdot U_{\text{lvl}}$, plotted with a standard plotting tool. Fill level U_{lvl} (y-Axis) is plotted against time. The arrow indicates the fill level after the silo motor ran continuously for 6 seconds. The horizontal lines indicate the discretization d10 of U_{lvl}, i.e. 10 partition elements.

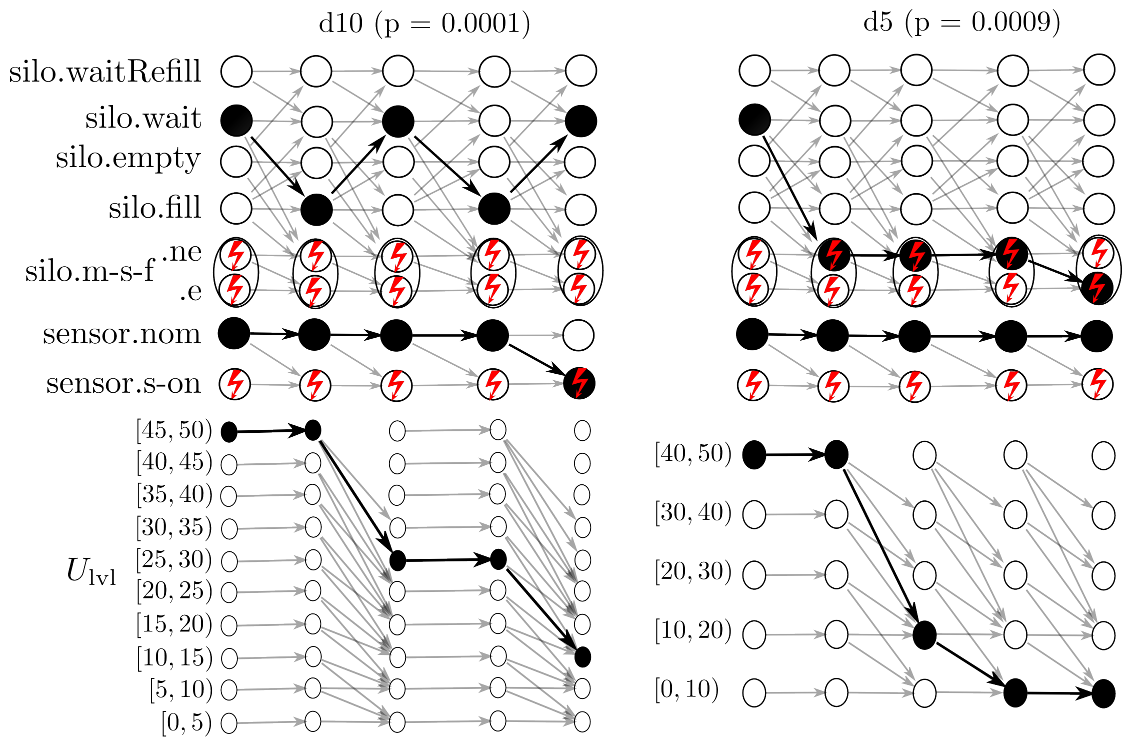


Figure 7.13.: The inferred system trajectories (black filled circles and arrows) for the sensor-fault scenario as trellis diagram for d10 for U_{lvl} (left) and for d5 (right). Grey shaded arrows show possible transitions. To simplify the graphics we omitted the other two sensor faults, stuck-off and unknown.

The underlying assumption for this scenario is that the sensor fault s-on occurs, and not, e.g., the motor-switch-fault. Additionally, we assume that the silo is not faulty from the start, i.e. components are functioning nominally at least until t_1. The diagnosis problem is solved correctly based on a discrete model if the most probable trajectory reveals the s-on fault. We can verify the scenario using the continuous filling behavior of the silo: The motor-switch-fault at its earliest can happen at t_1. If it does, the silo is being emptied for 6 seconds, until the sensor signal occurs. Within this period, however, the silo cannot become empty via filling, as can be seen from the plot of its continuous filling behavior $\dot{U}_{\text{lvl}} = -\text{fR} \cdot U_{\text{lvl}}$ in figure 7.12. This means, given that the discretized model is sufficiently accurate, the motor-switch-fault should be ruled out as possible explanation.

To show that our approach works for diagnosis and how varying degrees of abstraction influence the diagnosis quality, we generated discrete models for four different discretizations of U_{lvl} with 2, 5, 10 and 25 finite values. We denote these models with d2, d5, d10 and d25. All were generated using equally sized grid cells. We created COP instances from these models and solved them with Toulbar2 on a machine with an Intel core2duo dual core 2.2 Ghz CPU with 2GB RAM, running the Linux operating system. We tried Toulbar2's default configuration and a second one that exploits a tree decomposition (compare section 3.3.2) created prior to solving the instance. The idea was that the model structure extracted with tree decomposition would boost the online solving step (the tree decomposition can possibly be pushed offline). The COP size was for all instances 840 variables and 920 constraints (the discretization doesn't influence the number of variables or constraints, but the size of the constraints encoding the discrete flow).

Figure 7.13 shows the most probable system trajectories for our scenario for models d10 and d5. The trajectories are depicted as decomposed trellis diagrams. Big black arrows and black filled circles mark the trajectory found as most probable solution, grey arrows show possible transitions. The trellis diagrams are decomposed in the sense that partial trajectories for each component (silo and sensor) as well as for the discretized continuous behavior are shown with separate trellis diagrams.

From this figure it can be seen that if we choose a sufficiently fine-grained discretization, d10 in this case, the most probable diagnosis computed with the discrete model indeed identifies the correct fault s-on, while a too coarse abstraction (d5) does not result in a diagnosis which contains the s-on fault. We assume spurious solutions to be the culprit: The coarser the abstraction, the more probable become evolutions of U_{lvl} which in reality are practically impossible. With too coarse an abstraction (d5), the combination of the more likely motor-switch-fault and a spurious evolution of U_{lvl} with heightened probability becomes most probable. With a sufficiently fine grained abstraction (d10), the spurious evolution's probability is reduced to zero, which rules out the incorrect motor-switch-fault and leaves the sensor.stuck-on fault as most probable.

Although we only computed diagnoses with our approach we think these results indicate that plan assessment could indeed be extended towards hybrid models with our abstraction-based approach. As we have seen, going from diagnosis to plan assessment just means, in practical terms, to generate k COP solutions instead of one. The average online runtimes shown in table 7.4 seem to be on a par with what we have seen for plan assessment on discrete models, at least for this scenario. Therefore we are optimistic that continuous parts in models for plan assessment won't be a bottleneck. Not surprisingly, a modest increase in runtime can be seen for the more fine grained discretization d25. However, as we have seen, for a correct diagnosis in our scenario the much less expensive discretization d10 is sufficient. What did surprise us was that the offline decomposition did not, as we expected, lower the computational effort, but instead increased it significantly. At this

Table 7.4.: Runtime results in seconds for computing the most probable diagnosis based on different discrete models. Results were obtained by translating the models to COP instances and solving them with Toulbar2, using its standard configuration and a second one that uses an offline extracted tree decomposition of the COP. All instances had 840 variables and 920 constraints. Finer discretizations lead to bigger discrete flow constraints.

Toulbar2 config.	Discretization	Online runtime
default	d2	0.02s
parameters	d5	0.04s
	d10	0.04s
	d25	0.10s
with	d2	0.25s
tree	d5	0.25s
decomposition	d10	0.24s
	d25	0.33s

point we can only speculate about the reasons; maybe the model is too small for the expected increase to occur.

The runtimes of the three offline steps discretization, Markov chain generation and soft constraint encoding for d2, d5, d10 and d25 are 16.5, 39.0, 138.5 and 215.4 seconds. They show that the effort for hybrid model abstraction and encoding is considerable, even for such a small model. However, runtime is still within manageable bounds. Memory consumption might be a bigger issue, the offline steps for d25 consumed $\approx$ 300 MB. It remains for future experiments to show the limits of our method. The biggest portion of the resources are consumed by the Markov chain generation, which is not surprising: Converting the discrete part of a HyPHCA to a discrete PHCA and encoding the final discrete model as soft constraints is linear in the size of the model, whereas the step of Markov chain generation is exponential in the dimension of the continuous subspace associated with the abstracted continuous flow.

Our results do not provide sufficient ground yet for conclusions with respect to the scalability of our approach. However, our intuition is that it scales well as long as the number of components showing *different* continuous behavior is comparably small. The most expensive step is the generation of Markov chains to retrieve the discrete flows. Scalability can be improved, if unnecessary generation is avoided, i.e. by sharing the same abstraction among components with the same continuous behavior. Also, the expensive reachability analysis could be improved,e.g., by optimizing PHAVer parameters (or use a better tool). Finally, intelligent state space quantization, e.g. as introduced in [85], would reduce the number of quantization cells, resulting in fewer Markov chain states and thus a much less expensive abstraction step and smaller abstract models.

8. Related Work

Much of the literature related to this work has already been treated in a more suited context in different chapters and sections. In this chapter we turn our attention to some related fields that also deal with the assessment, evaluation or checking of plans. First, we look into the field of verification, where we explain general similarities and differences to plan assessment and highlight some of the more closely related publications. Then we attend to some works whose relation to this work is weaker, yet can also be seen as some sort of plan assessment or evaluation.

8.1. Verification

Verification is a diverse field with many applications and approaches. In the following, we don't try to give an exhaustive overview, which would be out of the scope of this work. Rather, we primarily focus on the closely related sub-areas of probabilistic model-checking and runtime verification. We also cast a quick glance at classical verification methods that build on industry standards in factory automation.

8.1.1. Probabilistic Model-Checking and Runtime Verification

The reader might already have guessed relations to verification from the fact that we exploit methods developed for verification (see section 7.3). Plan assessment is about defined goals and the probability of achieving them, given the observations we make during runtime of a system. Computing a probability of goal success based on observations can be seen as a variant of checking, or verifying, the runtime behavior of the system against certain properties, which is at the core of *runtime verification* [12]. Another related area is probabilistic model-checking [144, 8], which addresses the verification of properties for systems that are modeled probabilistically.

Both of these areas are rooted in model-checking [97], which asks the question: Given a model of a system and a property, do all potential system trajectories satisfy this property? For example, let $\mathcal{G}$ be a set of goal states and M a system model, then a typical model-checking property is "Eventually, a state in $\mathcal{G}$ is reached". Model-checking would then determine whether every potential trajectory permitted by M eventually reaches some state in $\mathcal{G}$.

Runtime verification considers properties like that, but doesn't try to verify them against *all potential* trajectories. Instead, it focuses on trajectories that are recorded during runtime of the system. A typical approach is to compile properties into monitors that can run alongside the system, collecting observations. They trigger an alarm as soon as it becomes clear that the property is violated. A variant called *runtime reflection* [11] combines this with diagnosis to find the faults causing the violation.

Probabilistic model-checking, as the name suggests, extends model-checking with Markovian probabilistic models and many probabilistic and statistical properties, such as probabilities to reach certain states or expected number of times a state is reached. The task of computing plan success probabilities could be cast as checking a property against a probabilistic system model within this framework. The success probability for a specific product could be formulated as a property "At $t+n$, with $p > \omega_{\text{success}}$ the product state is in $\mathcal{G}$", which can be expressed in the temporal probabilistic logic employed in probabilistic model-checking.

While many links between plan assessment and these two areas exist, a key difference is that, with few exceptions, the above approaches don't seem to regard autonomous behavior in the form of a system dynamically generating plans it later executes. Verification approaches are typically interested in formulating static properties to be checked, e.g. safety properties. Plan assessment, in contrast, "checks" dynamically generated plans, which is reflected in the execution adaptation function $\mathcal{E}_{\mathcal{P}}$. Another important difference is that plan assessment can handle flows of items through a system, for example product flows (via product models and the execution adaptation function). To our knowledge, no works in these areas seem to address this. Finally, we are not aware of works that exploit generic algorithms in constraint optimization or probabilistic reasoning via an automatic compilation of expressive, probabilistic hierarchical models to generic problem formats that these algorithms understand. However, this is not to say that plan assessment couldn't profit from methods and approaches developed in either runtime verification or probabilistic model-checking. It would be interesting to try and recast the plan assessment problem within these frameworks.

We highlight now some particularly relevant works from these two areas of verification. One such work is [5] about online verification of autonomous decisions of a cognitive car, which was already mentioned in section 7.3. It is assumed the car can automatically generate short-range drive plans, e.g. to overtake another car. The task is to minimize the risk of collision with oncoming traffic. This is achieved by verifying that on this planned track the probability to collide with an oncoming car is below a certain low threshold. Both the cognitive and the oncoming car are modeled with hybrid automata, which encode their velocity and acceleration, with different modes and mode-transitions encoding the gear shifting. Should the probability of collision be to high, the decision can be revised. This can be seen as sort of plan assessment, although focused on very specific goals. The work, however, doesn't cover in detail the adaptation of the method to explicitly represented plans, as this work does with execution adaptation for plan assessment. Moreover, no hierarchical modeling or flow of items is considered.

A very recent work addresses runtime verification of properties over *hidden* states of probabilistic models [160]. Especially interesting is the fact that systems are being modeled with PHCA. Verification is then formulated as a form of hidden Markov like filtering over a composition of the PHCA with the property to check. What this work doesn't address are things like product flow and dynamically generated plans.

In [110] an overview of probabilistic model-checking is given as well as current research directions and challenges. While it seems that expressive hierarchical modeling is not being addressed yet, one interesting direction is so-called compositional probabilistic model-checking. To scale up to larger systems the idea is to try and check the constituent components of a system individually, rather than all at once. Also addressed since very recently is online probabilistic model-checking using observations of the system. In [65] the authors propose an offline compilation step to make online probabilistic model-checking tractable. A key to efficiency lies in limiting the number of variable transition probabilities, following the assumption that typically only few transitions in the model are variable. The approach is based on discrete time Markov chains as models, no hierarchy is supported. None of these works, however, address diagnosis or product flows.

Finally, the authors of [116] introduce another interesting probabilistic verification approach based on the RMPL framework. The problem is to determine the most likely circumstances under which a high-level control program drives the system towards a goal violating state. A plan can be understood as such a high level control program; so in general, this problem is similar to the plan assessment problem. However, our problem differs in that we are interested in *sets* of goal achieving system trajectories, from which we derive the plan's success probability, while the verification problem is only interested in the single most probable goal violating trajectory. Another difference is that product flow is not being addressed.

8.1.2. Verification Based on Industry Standards for Automation

Classical verification approaches often build on models that follow industry standards for the design of automation controllers, for example [81, 158]. Both works perform design stage verification of the controllers based on automata models of the plant. If no safety properties are violated, implementation code for the controllers is generated automatically. As models variants of Petri nets [136] are used. In [81], for example, hierarchical Petri nets are employed. Another important aspect is distributed computing, addressed, for example, in [158]. Compared to plan assessment, however, these classical verification approaches usually don't address at least one of the following: 1) probabilistic models, 2) diagnosis, 3) automatically generated and explicitly represented production plans, 4) product flow. While [158] also models products, they don't seem to handle multiple products and their flow through the plant.

8.2. Other Variants of Plan Evaluation

Given the rather general meaning of the terms "plan" and "assessment", many works could be cited that however are only very loosely connected to this work. In this section we focus on some works whose relation to this thesis, in our view, is weaker than that of the works in the previous section, but which none the less constitute some interesting type of plan evaluation or assessment.

One application of automated planning is large scale operations management, e.g. reacting to oil spill disasters. [161] gives an overview of such planning scenarios and discusses, among other things, an oil spill response configuration system. It creates plans based on a spill trajectory forecast and then evaluates how much oil a specific plan is able to remove. It uses a special evaluation model along with the projected oil flow and the plan. The system monitors plan execution and is able to react to new events or goals provided by human operators. The main difference to these approaches is the scope (large scale operations planning vs, for example, scenarios of product manufacturing) and the fact that plans are not evaluated against a fixed goal (i.e. how likely they are to succeed), but against optimality criteria such as how much oil is being removed.

Early work addressed the problem of *automated manufacturability analysis*, which is to evaluate a process plan for machining a product from stock with respect to, e.g., design tolerances (does plan execution violate tolerances?) or the plan's cost. [133] discusses the IMACS (Interactive Manufacturing Analysis and Critique System), which generates process plans and then checks them against, e.g., design tolerance constraints, to evaluate whether the plans can reach their machining goals. Another work in this domain is [101], which introduces a petri net based approach to process plan evaluation. The authors estimate costs of process plans by representing the plans along with plant machines and their tools and configurations as special petri nets. They propose two kinds of petri net models, each with a special algorithm to compute the costs. Clearly there is some similarity to our plan assessment problem, especially the idea of generating models of machines and plans for plan evaluation. The main difference is that we're interested in dynamic machine behavior (nominal and off-nominal) induced by plan execution, whereas automated manufacturability analysis is concerned with evaluating plans against static constraints, such as design tolerances.

A typical problem in manufacturing is path planning and collision avoidance of industrial robot arms. A potential path is checked for collisions with other parts of the factory or even with itself (which can happen easily given the dexterity of today's robots) [73]. In the past, online collision avoidance has also been considered [74]. Especially the latter could be seen as sort of plan assessment, where a path plan is evaluated with respect to possible collisions, given the current situation (orientation/position of other robots, objects or humans in the work place). It differs from our work mainly in that we consider general, more abstract system trajectories and probabilistically modeled behavior.

9. Conclusion

We investigated solution approaches for a novel problem, called plan assessment, that lies at the intersection between model-based diagnosis and probabilistic reasoning. This thesis formally defines and analyzes this problem and develops two different solution approaches for it. One is rooted in model-based diagnosis, the other in probabilistic reasoning.

We consider plan assessment highly relevant as a means to support autonomous decisions in technical systems. For example, the problem arises when enriching automated production facilities with the autonomy necessary to handle high product variability. Automated production of individualized goods require individual production plans. However, the huge number of variants of completely individualized products will make it too expensive to hard-wire and verify those plans all in advance, as is current practice.

In general, research addresses this problem by augmenting systems with a certain amount of autonomous behavior. In the example, it means the facility dynamically creates plans, executes them, and autonomously reacts to potential plan failures. A plan might fail, for example, because of a faulty component such as a broken cutter of a machining station. This sort of autonomous behavior requires informations such as whether goals are still likely to be achieved, and if not, what might be the cause. This leads to the mentioned problem of assessing plans with respect to the goals they should achieve. While potentially more relevant to domains such as manufacturing, we developed plan assessment to be generic. It is readily applicable to every system that can be modeled with a probabilistic hierarchical constraint automaton (PHCA)[162], for example a household robot.

The solution approaches developed in this work are unique in that they use models described in engineering-style modeling languages and exploit generic algorithms provided in off-the-shelf toolboxes. This is motivated by the idea that approaches are more likely to be accepted if they are adapted to the tools used in the development of such technical systems, i.e. expressive modeling languages and existing toolboxes. Expressive languages allow efficient, hierarchical modeling of complex systems, while off-the-shelf toolboxes offer the opportunity to replace hand crafted code and algorithms by standardized, general algorithms, which are constantly being improved by their respective communities.

9.1. Summary of Contributions and Results

This thesis made the following two key contributions:

Definition and analysis of plan assessment (chapter 4): It formally defined the plan assessment problem based on probabilistic hierarchical constraint automata (PHCA). It analyzed how the problem manifests in autonomous manufacturing and how plan assessment can contribute to autonomous decisions in technical systems. Furthermore, it investigated relations of plan assessment to similar problems.

Computational approaches for plan assessment (chapter 5): It developed two approaches that both translate PHCA models to generic problem descriptions for external solvers, which implement generic algorithms. The first approach extends a previously developed model-based diagnosis method that computes most probable system trajectories as best solutions of a constraint optimization problem. The second approach, residing in the area of probabilistic reasoning, builds on generalized Bayesian networks. Notably, the thesis contributed a translation of PHCA models to these networks. It was tested in practice and a proof of its theoretical correctness was provided.

Additionally, the thesis made the following contributions:

Evaluation (chapter 6): Both approaches were tested and compared using off-the-shelf solving tools: For the first approach, the constraint optimizing tools Toolbar [27] and Toulbar2 [4] were used, and for the second the probabilistic inference tool Ace [46]. Results were obtained with a prototypical tool chain developed for this work. A separate result summary is given below.

Time window filtering extension (chapter 7): To break the exponential dependency on the number of time steps, the thesis developed the time-window-filtering algorithm. It allows to compute trajectories within a fixed-size horizon, which is then moved to cover the complete time horizon.

Stochastically Bounded Approximation extension (chapter 7): The thesis investigated how probabilistic sampling for plan assessment allows for stochastic error bounds based on confidence intervals.

Hybrid discrete/continuous models extension (chapter 7): Since systems are often being modeled using discrete automaton states mixed with continuous elements such as differential equations, the thesis developed a prototypical extension of plan assessment towards hybrid models that allow linear ordinary differential equations. It introduced a hybrid variant of PHCA, HyPHCA, and developed a method to convert a HyPHCA model into a (discrete) constraint net, including automated conversion of the continuous parts to discrete representations. This method incorporates off-the-shelf tools for the different computationally intensive steps, most notably the verification tool PHAVer [67]. Furthermore, a theoretical property for correctness of this translation was formulated.

As major result this thesis provides implementations of both approaches with which we obtained the results for our evaluation. Notably, we developed and implemented a novel translation to Bayesian logic nets and realized the elements necessary for a plan assessment component.

The evaluation results show that both approaches are capable of efficiently computing solutions for the plan assessment problem. In comparison, both approaches seem viable alternatives for the plan assessment problem: while differences can be seen, we see those as differences within the same class. Therefore, we interpret these as differences of the solving backends Toulbar2 and Ace, rather than as significant differences of our two general approaches. Concerning the scalability, while both approaches show good results for our models, we speculate that further work is necessary for larger problem sizes. For example, with more than 10 stations and products or more than 100 time steps the approaches could run into trouble. The constraint optimization approach allows approximation, which can remedy the limited scalability to an extent: on our instances, significant savings could be achieved if a modest error in the result is accepted. Furthermore, preliminary results with the time-window-filtering extension showed an improved scalability with time. Turning to the extension towards hybrid models, it seems that plan assessment can be extended with this method with a minor impact on online runtime. The effort for offline compilation was within manageable bounds for our example.

9.2. Current Limitations of Plan Assessment

Some important limitations exist that might restrict the use of plan assessment as a support for autonomous control.

- **Equal cost products:** This work didn't focus on utility values such as product costs. However, we suspect that plan assessment without utilities will stay limited: effectively, it forces the assumption that all goals have equal cost.
- **Fixed probabilities in models:** The probabilities in PHCA models are determined during design. Currently, they are fixed. However, what usually leads to unpredictable faults is component wear, which is better captured with *dynamically* adapted probabilities. This could be achieved by combining our approaches with model learning, such as introduced in [50], which estimates probabilities from long term measurements.
- **Error bounds available for sampling only:** We showed how to compute stochastic bounds on success probabilities when using probabilistic sampling to approximate solutions for plan assessment. For our constraint optimization based approach, the presented sketch to determine the approximation error has to be further developed.
- **No modeling language for HyPHCA:** There is no model description language for HyPHCA yet, which means that models must be specified manually with two separate parts, a discrete PHCA part and a continuous part with differential equations. What is needed is said language and a compiler that either generates these two parts automatically, or directly encodes to constraint nets.

9.3. Future Work

Apart from addressing the limitations mentioned above, this work opens up interesting directions for future work. First of, since plan assessment is meant as a support for autonomous decision routines, fully developing such a routine suggests itself. In this work, we already sketched such a procedure. In line with that falls the integration of utility values, for example product costs. The k-best constraint optimization for plan assessment could be extended with a procedure that evaluates the k most probable trajectories for utility as a second optimization criterium. The off-the-shelf tools we investigated in this work don't support general multi-criteria combinatorial optimization yet. Evaluating the k-best solutions in a second run would be a simple alternative.

Next, we think that in manufacturing examples automatic composition of PHCA models is necessary to acquire the flexibility needed for design-to-fabrication. Currently, in our example we use a static model of involved plant stations and products. However, to allow many different plans, a static model would have to contain all stations and products that are *potentially* involved. This is far too many, therefore models should be composed automatically from station and product sub-models. The model would then include only those components that are used in a plan. The hierarchical and concurrent nature of PHCA might simplify this task.

A component that can compose PHCA models automatically would also allow to investigate the scalability of our approaches on a larger scale. Manufacturing scenarios for example could be created quickly with a tool that creates PHCAs from automatically generated schedules and simple graph layouts of factory floors with predefined stations. The tool would use the composition component to construct the PHCA from predefined sub-models for the stations and products, including only those sub-models whose stations or products occur in the schedule. That would allow to quickly create artificial yet realistic models with different numbers of stations and products.

More in line with industrial requirements, an interesting direction is a distributed approach to plan assessment. Industry standards are already in place for the development of distributed controllers [92, 93, 94]. Currently, they do not incorporate all capabilities necessary to implement plan assessment, for example probabilistic modeling. It is certainly worthwhile to investigate how to marry plan assessment with distributed computation.

We also would like to investigate more solving backends, for example implementations of dynamic Bayesian network algorithms [102]. Of course, this would require developing compilers for dynamic Bayesian networks.

Last but not least, we would like to further investigate probabilistic sampling for plan assessment. The concept of the SampleSearch algorithm [75] to search for consistent (probability > 0) samples seems especially promising in conjunction with state-of-the-art constraint reasoning techniques. Such techniques are implemented in solvers such as Toulbar2, for example. However, it could prove more practical to try and couple SampleSearch with constraint reasoning and programming libraries such as Choco [152], which offer a concise programming interface.

Bibliography

[1] Rui Abreu, Peter Zoeteweij, and Arjan J. C. Van Gemund. A New Bayesian Approach to Multiple Intermittent Fault Diagnosis. In *Proc IJCAI-2009*, pages 653–658, San Francisco, CA, USA, 2009. Morgan Kaufmann.

[2] J. Adams, E. Balas, and D. Zawack. The Shifting Bottleneck Procedure for Job Shop Scheduling. *Management Science*, 34(3):391–401, March 1988.

[3] Armen Aghasaryan, Eric Fabre, Albert Benveniste, Renée Boubour, and Claude Jard. Fault Detection and Diagnosis in Distributed Systems: An Approach by Partially Stochastic Petri Nets. *Discrete Event Dynamic Systems*, 8:203–231, June 1998.

[4] D. Allouche, S. de Givry, and T. Schiex. Toulbar2, an Open Source Exact Cost Function Network Solver. Technical Report UR 875, INRA, F-31320 Castanet Tolosan, France, 2010. Contributors: M. Sanchez (SP), S. Bouveret (F), H. Fargier (F), F. Heras (SP), P. Jégou (F), J. Larrosa (SP), K. L. Leung (CN), S. N'diaye (F), E. Rollon (SP), C. Terrioux (F), G. Verfaillie (F), M. Zytnicki.

[5] Matthias Althoff, Olav Stursberg, and Martin Buss. Online Verification of Cognitive Car Decisions. In *Proc. IV-2007*, pages 728–733, 2007.

[6] Rajeev Alur, Radu Grosu, Insup Lee, and Oleg Sokolsky. Compositional Modeling and Refinement for Hierarchical Hybrid Systems. *Journal of Logic and Algebraic Programming*, 68(1-2):105–128, 2006.

[7] R. Bagnara, E. Ricci, E. Zaffanella, and P. M. Hill. Possibly Not Closed Convex Polyhedra and the Parma Polyhedra Library. Quaderno 286, Dipartimento di Matematica, Università di Parma, Italy, 2002.

[8] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*, chapter 10, pages 745–900. MIT Press, 2008.

[9] A. Bannat, T. Bautze, M. Beetz, J. Blume, K. Diepold, C. Ertelt, F. Geiger, T. Gmeiner, T. Gyger, A. Knoll, C. Lau, C. Lenz, M. Ostgathe, G. Reinhart, W. Roesel, T. Ruehr, A. Schuboe, K. Shea, I. Stork genannt Wersborg, S. Stork, W. Tekouo, F. Wallhoff, M. Wiesbeck, and M.F. Zaeh. Artificial Cognition in Production Systems. *Automation Science and Engineering, IEEE Transactions on*, 8(1):148–174, jan. 2011.

[10] Anthony Barrett. Model Compilation for Real-Time Planning and Diagnosis with Feedback. In *Proc. IJCAI-2005*. Professional Book Center, 2005.

[11] A. Bauer, M. Leucker, and C. Schallhart. Model-Based Runtime Analysis of Distributed Reactive Systems. In *Proc. ASWEC-2006*, pages 243–252, April 2006.

[12] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(4), 2009. in press.

[13] Michael Beetz. *Concurrent Reactive Plans: Anticipating and Forestalling Execution Failures*, volume 1772 of *LNCS*. Springer Publishers, 2000.

[14] Michael Beetz, Jan Bandouch, Alexandra Kirsch, Alexis Maldonado, Armin Müller, and Radu Bogdan Rusu. The Assistive Kitchen - A Demonstration Scenario for Cognitive Technical Systems. In *Proc. COE Workshop on human adaptive mechatronics*, 2007.

[15] Michael Beetz, Martin Buss, and Dirk Wollherr. Cognitive Technical Systems — What is the Role of Artificial Intelligence? In *Proc. KI-2007*, volume 4667 of *LNCS*, pages 19–42. Springer, 2007.

[16] Michael Beetz, Ulrich Klank, Ingo Kresse, Alexis Maldonado, Lorenz Mösenlechner, Dejan Pangercic, Thomas Rühr, and Moritz Tenorth. Robotic Roommates Making Pancakes. In *11th IEEE-RAS International Conference on Humanoid Robots*, Bled, Slovenia, October, 26–28 2011. Accepted for publication.

[17] Michael Beetz, Lorenz Mösenlechner, and Moritz Tenorth. CRAM – A Cognitive Robot Abstract Machine for Everyday Manipulation in Human Environments. In *IEEE/RSJ International Conference on Intelligent RObots and Systems.*, 2010.

[18] E. Bensana, G. Verfaillie, J. Agnse, N. Bataille, and D. Blumstein. Exact and Inexact Methods for the Daily Management of an Earth Observation Satellite. In

Proc. Intl. Symposium on Space Mission Operations and Ground Data Systems, Munich, Germany, 1996.

[19] Mikhail Bernadsky, Raman Sharykin, and Rajeev Alur. Structured Modeling of Concurrent Stochastic Hybrid Systems. In Yassine Lakhnech and Sergio Yovine, editors, *Proc. FORMATS/FTRFT-2004*, volume 3253 of *LNCS*, pages 309–324, Grenoble, France, September 2004. Springer.

[20] G. Berry and G. Gonthier. The Synchronous Programming Language ESTEREL: Design, Semantics, Implementation. *Science of Computer Programming*, 19(2):87–152, November 1992.

[21] C.M. Bishop et al. *Pattern Recognition and Machine Learning.* Springer, 2006.

[22] C.M. Bishop et al. *Pattern Recognition and Machine Learning*, chapter 2, pages 67–74. Springer, 2006.

[23] Mogens Blanke, Michel Kinnaert, Jan Lunze, Marcel Staroswiecki, and J. Schröder. *Diagnosis and Fault-Tolerant Control.* Springer, Secaucus, NJ, USA, 2006.

[24] Hans L. Bodlaender. A Linear-Time Algorithm for Finding Tree-Decompositions of Small Treewidth. *SIAM Journal on Computing*, 25(6):1305–1317, 1996.

[25] Guido Boella and Rossana Damiano. A Replanning Algorithm for Decision Theoretic Hierarchical Planning: Principles and Empirical Evaluation. *Applied Artificial Intelligence*, 22(10):937–963, 2008.

[26] Blai Bonet and Hector Geffner. Solving pomdps: Rtdp-bel vs. point-based algorithms. In Craig Boutilier, editor, *IJCAI*, pages 1641–1646, 2009.

[27] S. Bouveret, F. Heras, S.de Givry, J. Larrosa, M. Sanchez, and T. Schiex. ToolBar: A State-of-the-Art Platform for WCSP. `http://www.inra.fr/mia/T/degivry/ToolBar.pdf` (08/2011), 2004.

[28] Joseph Brady, Ellen Monk, and Bret Wagner. *Concepts in Enterprise Resource Planning.* Course Technology, 1st edition, 2001.

[29] A. W. Brander and M. C. Sinclair. A Comparative Study of k-Shortest Path Algorithms. In *In Proc. of 11th UK Performance Engineering Workshop*, pages 370–379, 1995.

[30] A. Bratukhin, BA Khan, and A. Treytl. Resource-Oriented Scheduling in the Distributed Production. In *Proc. Industrial Informatics*, volume 2, 2007.

[31] Christian Brecher, editor. *Integrative Production Technology for High-Wage Countries*. Springer, 1st edition, November 2011.

[32] Hendrik Van Brussel, Jo Wyns, Paul Valckenaers, Luc Bongaerts, and Patrick Peeters. Reference Architecture for Holonic Manufacturing Systems: PROSA. *Comput. Ind.*, 37(3):255–274, November 1998.

[33] Benno Büeler, Andreas Enge, and Komei Fukuda. *Polytopes — Combinatorics and Computation*, chapter Exact Volume Computation for Polytopes: A Practical Study, pages 131–154. Number 29 in DMV Seminar. Birkhäuser, 2000.

[34] Martin Buss, Michael Beetz, and Dirk Wollherr. CoTeSys — Cognition for Technical Systems. In *Workshop Proc. HAM-2007*, 2007.

[35] B. Cabon, S. de Givry, L. Lobjois, T. Schiex, and J.P. Warners. Radio Link Frequency Assignment. *Constraints*, 4(1):79–89, 1999.

[36] George Casella and Edward I. George. Explaining the Gibbs Sampler. *The American Statistician*, 46(3):167–174, August 1992.

[37] Mark Chavira and Adnan Darwiche. Compiling Bayesian Networks with Local Structure. In *Proc. IJCAI-2005*, pages 1306–1312, 2005.

[38] Mark Chavira and Adnan Darwiche. Compiling Bayesian Networks Using Variable Elimination. In *Proc IJCAI-2007*, pages 2443–2449, 2007.

[39] CJ Clopper and ES Pearson. The Use of Confidence or Fiducial Limits Illustrated in the Case of the Binomial. *Biometrika*, 26(4):404–413, 1934.

[40] Computing Community Consortium / Computing Research Association. A roadmap for us robotics — from internet to robotics. http://www.us-robotics.us/, May 2009.

[41] Gregory F. Cooper. The Computational Complexity of Probabilistic Inference Using Bayesian Belief Networks (Research Note). *Artif. Intell.*, 42(2-3):393–405, March 1990.

[42] M. Cooper, S. De Givry, M. Sanchez, T. Schiex, and M. Zytnicki. Virtual Arc Consistency for Weighted CSP. In *Proc. AAAI-2008*, volume 1, pages 253–258. AAAI Press, 2008.

[43] M. Daigle, X. Koutsoukos, and G. Biswas. A Qualitative Approach to Multiple Fault Isolation in Continuous Systems. In *Proc. AAAI-2007*, volume 1. AAAI Press, 2007.

[44] Matthew J. Daigle, Xenofon D. Koutsoukos, and Gautam Biswas. A Qualitative Event-Based Approach to Continuous Systems Diagnosis. *IEEE Transactions on Control Systems Technology*, 17(4):780–793, July 2009.

[45] R Danchick and GE Newnam. Reformulating Reid's MHT Method With Generalised Murty K-best Ranked Linear Assignment Algorithm. *IEE Proceedings — Radar Sonar and Navigation*, 153(1):13–22, FEB 2006.

[46] Adnan Darwiche. A Differential Approach to Inference in Bayesian Networks. *Journal of the ACM*, 50(3):280–305, 2003.

[47] Simon de Givry, Federico Heras, Matthias Zytnicki, and Javier Larrosa. Existential Arc Consistency: Getting Closer to Full Arc Consistency in Weighted CSPs. In *Proc. IJCAI-2005*, pages 84–89, 2005.

[48] Simon de Givry, Javier Larrosa, Pedro Meseguer, and Thomas Schiex. Solving Max-SAT as Weighted CSP. In *CP-2003*, pages 363–376, 2003.

[49] Johan de Kleer. Using Crude Probability Estimates to Guide Diagnosis. *Artificial Intelligence*, 45(3):381–391, 1990.

[50] Johan de Kleer, Lukas Kuhn, J.J. Liu, R. Price, M. B. Do, and R. Zhou. Continuously Estimating Persistent and Intermittent Failure Probabilities. In *Proc. SAFE Process 2009*, Barcelona, Spain, June 2009.

[51] Rina Dechter. *Constraint Processing.* Morgan Kaufmann, San Francisco, CA 94104-3205, 1st edition, 2003.

[52] Rina Dechter and Natalia Flerova. Heuristic search for m best solutions with applications to graphical models. In *Proc. Soft-2011 (a workshop of CP 2011)*, 2011.

[53] Rina Dechter and Judea Pearl. Generalized Best-1st Search Strategies and the Optimality of A-Star. *Journal of the ACM*, 32(3):505–536, 1985.

[54] Sierke Dominka. *Hybride Inbetriebnahme von Produktionsanlagen — Von der Virtuellen zur Realen Inbetriebnahme.* PhD thesis, Technische Universität München, 2007.

[55] Arnaud Doucet and Adam Johansen. A Tutorial on Particle Filtering and Smoothing: Fifteen Years Later. Technical report, Department of Statistics, University of British Columbia, 2008.

[56] Hilding Elmqvist, Sven Erik Mattsson, and Martin Otter. Object-Oriented and Hybrid Modeling in Modelica. *Journal Européen des Systèmes Automatisés*, 35(1):395–404, January 2001.

[57] Christoph Ertelt, Thomas Rühr, Dejan Pangercic, Kristina Shea, and Michael Beetz. Integration of Perception, Global Planning and Local Planning in the Manufacturing Domain. In *Proc. ETFA-2009*, 2009.

[58] EUROP — European Robotics Technology Platform. Robotic Visions: To 2020 and Beyond — The Strategic Research Agenda for Robotics in Europe. `http://www.robotics-platform.eu/cms/upload/SRA/2010-06_SRA_A4_low.pdf` (08.2011), July 2009.

[59] European Commission Ad-hoc Industrial Advisory Group for the Factory of the Future. Factories of the Future PPP — Strategic Multi-Annual Roadmap. `http://ec.europa.eu/research/industrial_technologies/useful-documents_en.html`, March 2010.

[60] Yousri El Fattah and Rina Dechter. Diagnosing Tree-Decomposable Circuits. In *Proc. IJCAI-1995*, pages 1742–1749, 1995.

[61] A. Felfernig, G. Friedrich, and D. Jannach. Conceptual Modeling for Configuration of Mass-Customizable Products. *Artificial Intelligence in Engineering*, 15(2):165–176, April 2001.

[62] A. Felner, S. Kraus, and R.E. Korf. KBFS: K-Best-First Search. *Annals of Mathematics and Artificial Intelligence*, 39(1-2):19–39, September 2003.

[63] L. Ferrarini, C. Veber, A. Luder, J. Peschke, A. Kalogeras, J. Gialelis, J. Rode, D. Wunsch, V. Chapurlat, and D.E. e Informazione. Control Architecture for Reconfigurable Manufacturing Systems: the PABADIS'PROMISE Approach. In *Proc. ETFA-2006*, pages 545–552, 2006.

[64] R.E. Fikes and N.J. Nilsson. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. Technical Report 43r, AI Center, SRI International, 333 Ravenswood Ave, Menlo Park, CA 94025, May 1971. SRI Project 8259.

[65] Antonio Filieri, Carlo Ghezzi, and Giordano Tamburrelli. Run-time Efficient Probabilistic Model Checking. In *Proc. ICSE-2011*, pages 341–350, Waikiki, Honolulu, HI, USA, 2011. ACM.

[66] Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*. Foundations of Artificial Intelligence. Elsevier, 2006.

[67] Goran Frehse. PHAVer: Algorithmic Verification of Hybrid Systems Past HyTech. In *Proc. HSCC-2005*, volume 3414 of *LNCS*, pages 258–273. Springer, 2005.

[68] Markus P. J. Fromherz, Daniel G. Bobrow, and Johan de Kleer. Model-Based Computing for Design and Control of Reconfigurable Systems. *AI Magazine*, 24(4):120–130, 2004.

[69] Markus P. J. Fromherz, Vijay A. Saraswat, and Daniel G. Bobrow. Model-Based Computing: Developing Flexible Machine Control Software. *Artificial Intelligence*, 114(1-2):157–202, 1999.

[70] R. Fung and B. Del Favero. Backward Simulation in Bayesian Networks. In *Proc. UAI-1994*, pages 227–234. Morgan Kaufmann, July 1994.

[71] R. M. Fung and K.-C. Chang. Weighting and Integrating Evidence for Stochastic Simulation in Bayesian Networks. In *Proc. UAI-1989*, pages 209–220. North-Holland Publishing, 1989.

[72] J. Gebhardt, H. Detmer, and A.L. Madsen. Predicting Parts Demand in the Automotive Industry — An Application of Probabilistic Graphical Models. In *Workshop Proc. Bayesian Modelling Applications (UAI-2003)*, Acapulco, Mexico, 2003. Morgan Kaufmann.

[73] C. Van Geem and T. Siméon. KCD: A Collision Detector for Path Planning in Factory Models. Technical report, LAAS-CNRS, 2001.

[74] M. Gerke and H. Hoyer. Fuzzy Collision Avoidance for Industrial Robots. In *Proc. IEEE IROS-1995*, volume 2, Los Alamitos, CA, USA, 1995. IEEE CompSoc Press.

[75] V. Gogate and R. Dechter. Samplesearch: A Scheme that Searches for Consistent Samples. In *Proc. AISTATS-2007*. Omnipress, 2007.

[76] C. Goodrich and J. Kurien. Continuous Measurements and Quantitative Constraints — Challenge Problems for Discrete Modeling Techniques. In *Proc. I-SAIRAS-2001*, 2001.

[77] Walter Hamscher, Luca Console, and Johan de Kleer, editors. *Readings in Model-Based Diagnosis.* Morgan Kaufmann, San Francisco, CA, USA, 1992.

[78] David Harel. Statecharts: A Visual Formulation for Complex Systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[79] Peter Hart, Nils Nilsson, and Bertram Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, February 1968.

[80] Sandra C. Hayden, Adam J. Sweet, and Scott E. Christa. Livingstone model-based diagnosis of earth observing one. In *Proc. AIAA Intelligent Systems Technical Conference 2004.* AIAA, September 2004.

[81] M. Heiner, P. Deussen, and J. Spranger. A Case Study in Design and Verification of Manufacturing System Control Software with Hierarchical Petri Nets. *International Journal of Advanced Manufacturing Technology*, 15(2):139–152, 1999.

[82] B. Helms, K. Shea, and F. Hoisl. A Framework for Computational Design Synthesis Based on Graph-Grammars and Function-Behavior-Structure. In *Proc. IDETC/CIE-2009*, volume 8, pages 841–851, San Diego, CA, USA, September 2009. ASME.

[83] Thomas Henzinger. The Theory of Hybrid Automata. In *Proc. LICS-1996*, pages 278–292, New Brunswick, New Jersey, 1996. IEEE CompSoc Press.

[84] Michael Himmelsbach, Thorsten Luettel, Falk Hecker, Felix von Hundelshausen, and Hans-Joachim Wuensche. Autonomous Off-Road Navigation for MuCAR-3 – Improving the Tentacles Approach: Integral Structures for Sensing and Motion. *Kuenstliche Intelligenz*, 25(2):145–149, 2011. Special Issue on Off-Road-Robotics.

[85] Michael W. Hofbaur and Theresa Rienmüller. Qualitative Abstraction of Piecewise Affine Systems. In *Workshop Proc. QR-2008*, pages 43–48, June 2008.

[86] Michael W. Hofbaur and Brian C. Williams. Mode estimation of probabilistic hybrid systems. In *Proc. HSCC-2002*, pages 253–266, Stanford, California, USA, 2002. Springer.

[87] Michael W. Hofbaur and Brian C. Williams. Hybrid Estimation of Complex Systems. *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, 34(5):2178–2191, 2004.

[88] Wei Hu. Most Probable Explanation Generation for a Bayesian Network. US patent 7899771, March 2011.

[89] Cecil Huang and Adnan Darwiche. Inference in Belief Networks: A Procedural Guide. *International Journal of Approximate Reasoning*, 15(3):225–263, 1996.

[90] Jinbo Huang, Mark Chavira, and Adnan Darwiche. Solving MAP Exactly by Searching on Compiled Arithmetic Circuits. In *Proc. AAAI-2006*, pages 143–148. AAAI, AAAI Press, July 2006.

[91] Jinbo Huang and Adnan Darwiche. On Compiling System Models for Faster and More Scalable Diagnosis. In *Proc. AAAI-2005*, pages 300–306. AAAI Press, 2005.

[92] IEC 61499-1. Function Blocks — Part 1: Architecture. Geneva: International Electrotechnical Comission, 2005.

[93] IEC 61499-2. Function Blocks — Part 2: Software Tool Requirements. Geneva: International Electrotechnical Comission, 2004.

[94] IEC 61499-4. Function Blocks — Part 4: Rules for Compliance Profiles. Geneva: International Electrotechnical Comission, 2005.

[95] Dominik Jain, Klaus von Gleissenthall, and Michael Beetz. Bayesian Logic Networks and the Search for Samples with Backward Simulation and Abstract Constraint Learning. In *Proc. KI-2011*, LNCS. Springer, 2011. To appear.

[96] Dominik Jain, Stefan Waldherr, and Michael Beetz. Bayesian Logic Networks. Technical report, Technische Universität München, 2009.

[97] Edmund M. Clarke Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999.

[98] R.E. Kalman et al. A New Approach to Linear Filtering and Prediction Problems. *Journal of Basic Engineering*, 82(1):35–45, 1960.

[99] Kalev Kask and Rina Dechter. Mini-Bucket Heuristics for Improved Search. In *Proc. UAI-1999*, pages 314–323, San Francisco, CA, 1999. Morgan Kaufmann.

[100] Kalev Kask, Rina Dechter, Javier Larrosa, and Fabio Cozman. Bucket-Tree Elimination for Automated Reasoning. Technical Report 92, Department of Information and Computer Science, University of California, Irvine, 2001.

[101] D. Kiritsis, K. P. Neuendorf, and P. Xirouchakis. Petri Net Techniques for Process Planning Cost Estimation. *Advances in Engineering Software*, 30(6):375–387, 1999.

[102] Uffe Kjaerulff. Dhugin - A Computational System for Dynamic Time-Sliced Bayesian Networks. *International Journal of Forecasting*, 11(1):89–111, March 1995.

[103] Bradley Knox and Ole Mengshoel. Diagnosis and Reconfiguration using Bayesian Networks: An Electrical Power System Case Study. In *Workshop Proc. SAS-2009*, 2009.

[104] Y. Koren, U. Heisel, F. Jovane, T. Moriwaki, G. Pritschow, G. Ulsoy, and H. Van Brussel. Reconfigurable Manufacturing Systems. *CIRP Annals — Manufacturing Technology*, 48(2):527–540, 1999.

[105] X. Koutsoukos, J. Kurien, and F. Zhao. Monitoring and Diagnosis of Hybrid Systems Using Particle Filtering Methods. In *Proc. MTNS-2002*, University of Notre Dame, IN, USA, August 12–16 2002.

[106] Lukas Kuhn, Bob Price, Johan de Kleer, Minh Binh Do, and Rong Zhou. Pervasive Diagnosis: The Integration of Diagnostic Goals into Production Plans. In Dieter Fox and Carla P. Gomes, editors, *Proc. AAAI-2008*, pages 1306–1312. AAAI Press, 2008.

[107] H. Kühnle. *Distributed Manufacturing: Paradigm, Concepts, Solutions and Examples.* Springer Verlag, 2009.

[108] James Kurien and P. Pandurang Nayak. Back to the Future for Consistency-Based Trajectory Tracking. In *Proc. AAAI-2000*, pages 370–377. AAAI Press, 2000.

[109] Jürgen Kuster, Dietmar Jannach, and Gerhard Friedrich. Applying Local Rescheduling in Response to Schedule Disruptions. *Annals of Operations Research*, 180(1):265–282, 2010.

[110] M. Kwiatkowska, G. Norman, and D. Parker. Advances and Challenges of Probabilistic Model Checking. In *Proc. 48th Annual Allerton Conference on Communication, Control and Computing.* IEEE Press, 2010.

[111] Javier Larrosa. Node and Arc Consistency in Weighted CSP. In *Proc. AAAI-2002*, pages 48–53, Menlo Park, CA, USA, 2002. AAAI Press.

[112] E. L. Lawler and D. E. Wood. Branch-and-bound methods: A survey. *Operations Research*, 14(4):699–719, 1966.

[113] Eugene L. Lawler. A Procedure for Computing the K Best Solutions to Discrete Optimization Problems and its Application to the Shortest Path Problem. *Management Science*, 18(7):401–405, 1972.

[114] Kenneth Levenberg. A Method for the Solution of Certain Non-Linear Problems in Least Squares. *The Quarterly of Applied Mathematics*, 2(2):164–168, 1944.

[115] Jan Lunze and Bernhard Nixdorf. Representation of Hybrid Systems by Means of Stochastic Automata. *Mathematical and Computer Modelling of Dynamical Systems*, 7(4):383–422, December 2001.

[116] Tazeen Mahtab, Greg Sullivan, and Brian C. Williams. Automated Verification of Model-based Programs Under Uncertainty. In *Proc. ISDA-2004*, August 2004.

[117] Paul Maier, Dominik Jain, and Martin Sachenbacher. Compiling AI Engineering Models for Probabilistic Inference. In *Proc. KI-2011*, volume 7006 of *LNCS*, pages 191–203. Springer, October 2011.

[118] Paul Maier, Dominik Jain, Stefan Waldherr, and Martin Sachenbacher. Plan Assessment for Autonomous Manufacturing as Bayesian Inference. In *Proc. KI-2010*, volume 6359 of *LNCS*, pages 263–271. Springer, September 2010.

[119] Paul Maier and Martin Sachenbacher. Diagnosis and Fault-Adaptive Control for Mechatronic Systems using Hybrid Constraint Automata. In *Proc. PHM-2009*, San Diego, CA, USA, September 2009.

[120] Paul Maier and Martin Sachenbacher. Receding Time Horizon Self-Tracking and Assessment for Autonomous Manufacturing. In *Workshop Proc. Self-X-2010*, Karlsruhe, Germany, September 2010. MV-Wissenschaft.

[121] Paul Maier, Martin Sachenbacher, Thomas Rühr, and Lukas Kuhn. Automated Plan Assessment in Cognitive Manufacturing. *Adv. Eng. Informat.*, 24(3):241–376, May 2010.

[122] Donald Marquardt. An algorithm for least-squares estimation of nonlinear parameters. *SIAM Journal on Applied Mathematics*, 11(2):431–441, 1963.

[123] H. B. Marri, A. Gunasekaran, and R. J. Grieve. Computer-Aided Process Planning: A State of Art. *The International Journal of Advanced Manufacturing Technology*, 14(4):261–268, 1998.

[124] R. Mateescu and R. Dechter. Mixed Deterministic and Probabilistic Networks. *Annals of Mathematics and Artificial Intelligence*, 54(1):3–51, 2008.

[125] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins. PDDL — The Planning Domain Definition Language. Technical Report 98-003/DCS TR-1165, Yale Center for Computational Vision and Control, October 1998.

[126] S.A. McIlraith, G. Biswas, D. Clancy, and V. Gupta. Hybrid Systems Diagnosis. In *Proc. HSCC-2000*, volume 1790 of *LNCS*, pages 282–295. Springer, 2000.

[127] Pedro Meseguer, Francesca Rossi, and Thomas Schiex. *Handbook of Constraint Programming*, chapter 9: Soft Constraints, pages 281–328. Foundations of Artificial Intelligence. Elsevier, 2006.

[128] Tsoline Mikaelian. Model-based Monitoring and Diagnosis of Systems with Software-Extended Behavior. Master's thesis, MASSACHUSETTS INSTITUTE OF TECHNOLOGY, Department of Aeronautics and Astronautics, June 2005.

[129] Tsoline Mikaelian, C. Brian Williams, and Martin Sachenbacher. Model-based Monitoring and Diagnosis of Systems with Software-Extended Behavior. In *Proc. AAAI-2005*, pages 327–333, Pittsburgh, USA, 2005. AAAI Press.

[130] L. Monostori, J. Váncza, and S.R.T. Kumara. Agent-Based Systems for Manufacturing. *CIRP Annals - Manufacturing Technology*, 55(2):697–720, 2006.

[131] K. G. Murty. An Algorithm for Ranking all Assignments on Order of Increasing Cost. *Operations Research*, 16(3):682–687, 1968.

[132] Nicola Muscettola, P. Pandurang Nayak, Barney Pell, and Brian C. Williams. Remote Agent: To Boldly Go Where No AI System Has Gone Before. *Artificial Intelligence*, 103(1-2):5–47, 1998.

[133] Dana S. Nau, Satyandra K. Gupta, and William C. Regli. Manufacturing Operation Planning Versus AI Planning. In *Integrated Planning Applications: Papers from the 1995 AAAI Spring Symposium*, pages 92–101. AAAI Press, 1995.

[134] James Park. Map complexity results and approximation methods. In *Porc. UAI-2002*, pages 388–396, San Francisco, CA, 2002. Morgan Kaufmann.

[135] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufmann, San Francisco, CA, USA, 1988.

[136] C. A. Petri. *Kommunikation mit Automaten.* PhD thesis, Institut für instrumentelle Mathematik, Bonn, 1962.

[137] David Poole. First-Order Probabilistic Inference. In *Proc. IJCAI-2003*, pages 985–991, Acapulco, Mexico, August 2003. Morgan Kaufmann.

[138] Dan Roth. On the Hardness of Approximate Reasoning. *Artif. Intell.*, 82(1-2):273–302, April 1996.

[139] Wheeler Ruml, Minh Binh Do, Rong Zhou, and Markus P. J. Fromherz. On-line Planning and Scheduling: An Application to Controlling Modular Printers. *Journal of Artificial Intelligence Research*, 40:415–468, 2011.

[140] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall Series in Artificial Intelligence. Prentice Hall, 2nd edition, December 2002.

[141] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*, chapter 15 Probabilistic Reasoning Over Time. Prentice Hall Series in Artificial Intelligence. Prentice Hall, 2nd edition, December 2002.

[142] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*, chapter 17 Making Complex Decisions. Prentice Hall Series in Artificial Intelligence. Prentice Hall, 2nd edition, December 2002.

[143] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*, chapter 11 Planning. Prentice Hall Series in Artificial Intelligence. Prentice Hall, 2nd edition, December 2002.

[144] J. Rutten, M. Kwiatkowska, G. Norman, and D. Parker. *Mathematical Techniques for Analyzing Concurrent and Probabilistic Systems*, volume 23 of *CRM Monograph Series.* American Mathematical Society, 2004.

[145] M. Sachenbacher, P. Struss, and R. Weber. Advances in Design and Implementation of OBD Functions for Diesel Injection Systems based on a Qualitative Approach to Diagnosis. In *Proc. SAE-2000 World Congress*, Detroit, USA, 2000. Society of Automotive Engineers.

[146] Martin Sachenbacher and Brian Williams. Diagnosis as Semiring-Based Constraint Optimization. In *Proc. ECAI-2004*, pages 873–877, Valencia, Spain, 2004. IOS Press.

[147] Subhash C. Sarin, Yuqiang Wang, and Dae B. Chang. A Schedule Algebra Based Approach to Determine the K-Best Solutions of a Knapsack Problem with a Single Constraint. In Nimrod Megiddo, Yinfeng Xu, and Binhai Zhu, editors, *Algorithmic Applications in Management*, volume 3521 of *LNCS*, pages 440–449. Springer, 2005.

[148] Thomas Schiex. Soft Constraint Processing. Teaching Document, `http://www.inra.fr/mia/T/schiex/Export/Ecole.pdf`, 2005. Accessed 07.2011.

[149] Thomas Schiex, Hélène Fargier, and Gerard Verfaillie. Valued Constraint Satisfaction Problems: Hard and Easy Problems. In Chris Mellish, editor, *Proc. IJCAI-1995*, pages 631–637, Montreal, 1995.

[150] Daniel Sheldon, Bistra Dilkina, Adam Elmachtoub, Ryan Finseth, Ashish Sabharwal, Jon Conrad, Carla Gomes, David Shmoys, Will Allen, Ole Amundsen, and Buck Vaughan. Optimal Network Design for the Spread of Cascades. In *Workshop Proc. CROCS-2010*, June 2010.

[151] Sampath Srinivas and P. Pandurang Nayak. Efficient Enumeration of Instantiations in Bayesian Networks. In Eric Horvitz and Finn Verner Jensen, editors, *Proc. UAI-1996*, pages 500–508. Morgan Kaufmann, 1996.

[152] CHOCO Team. Choco: an Open Source Java Constraint Programming Library. Research report 10-02-INFO, Ecole des Mines de Nantes, 2010.

[153] Cyril Terrioux and Philippe Jégou. Bounded Backtracking for the Valued Constraint Satisfaction Problems. In Francesca Rossi, editor, *Proc. CP-2003*, volume 2833 of *LNCS*, pages 709–723. Springer, 2003.

[154] A. Treytl, G. Pratl, and B.A. Khan. Agents on RFID Tags - A Chance to Increase Flexibility in Automation. In *Proc. ANIPLA-2006*, November 2006.

[155] Kanji Ueda, Jari Vaario, and Kazuhiro Ohkura. Modelling of Biological Manufacturing Systems for Dynamic Reconfiguration. *CIRP Annals - Manufacturing Technology*, 46(1):343–346, 1997.

[156] Winfried van Holland and Willem F. Bronsvoort. Assembly Features in Modeling and Planning. *Robot. Cim.-Int. Manuf.*, 16(4):277–294, August 2000.

[157] Guilherme E. Vieira, Jeffrey W. Herrmann, and Edward Lin. Rescheduling Manufacturing Systems: A Framework of Strategies, Policies, and Methods. *Journal of Scheduling*, 6(1):39–62, 2003.

[158] V. Vyatkin and H.M. Hanisch. Verification of Distributed Control Systems in Intelligent Manufacturing. *Journal of Intelligent Manufacturing*, 14(1):123–136, February 2003.

[159] Wade Roush. Immobots take control. *Technology Review*, 2002.

[160] Cristina Wilcox and Brian Williams. Runtime Verification of Stochastic, Faulty Systems. In Howard Barringer, Ylies Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann, editors, *Runtime Verification*, volume 6418 of *LNCS*, pages 452–459. Springer, 2010.

[161] David E. Wilkins and Marie desJardins. A Call for Knowledge-Based Planning. *AI Magazine*, 22(1):99–115, 2001.

[162] Brian C. Williams, Seung Chung, and Vineet Gupta. Mode Estimation of Model-based Programs: Monitoring Systems with Complex Behavior. In *Proc. IJCAI-2001*, pages 579–590, 2001.

[163] Brian C. Williams, Michel D. Ingham, S. H. Chung, and P. H. Elliott. Model-Based Programming of Intelligent Embedded Systems and Robotic Space Explorers. *Proceedings of the IEEE*, 91(1):212–237, January 2003.

[164] Brian C. Williams, Michel D. Ingham, Seung Chung, Paul Elliott, Michael Hofbaur, and Gregory T. Sullivan. Model-Based Programming Of Fault-Aware Systems. *AI Magazine*, 24(4):61–75, 2004.

[165] Michael Wooldridge and Nicholas R. Jennings. Intelligent Agents: Theory and Practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.

[166] M. F. Zaeh, M. Beetz, K. Shea, G. Reinhart, K. Bender, C. Lau, M. Ostgathe, W. Vogl, M. Wiesbeck, M. Engelhard, C. Ertelt, T. Rühr, M. Friedrich, and S. Herle. The Cognitive Factory. In Hoda A. ElMaraghy, editor, *Changeable and Reconfigurable Manufacturing Systems*, Springer Series in Advanced Manufacturing, pages 355–371. Springer London, 2009.

[167] Heming Zhang, Hongwei Wang, David Chen, and Gregory Zacharewicz. A Model-Driven Approach to Multidisciplinary Collaborative Simulation for Virtual Product Development. *Advanced Engineering Informatics*, 24(2):167–179, 2010.

A. Implementation Usage

Here we briefly describe how to use our implementation of the described approaches for plan assessment. We implemented a library in Python and scripts that exploit that library. For instance, scripts compute plan assessment results for single problem instances, or encode experiments over many instances. We first describe a script that represents a prototypical implementation of the plan assessment component using the model-based diagnosis approach. Then we describe our translator of PHCA models to BLNs, and how to use the BLN framework to compute success probabilities.

A.1. Requirements

These are the requirements for our tool suite:

- A Windows or Linux environment (due to Ace being only available for these platforms).
- Python 2.5 or higher.
- Jython 2.5 or higher.
- The BLN framework, which can be found at `http://www9-old.cs.tum.edu/people/jain/dl.php?get=probcog`(06.2011).
- The probabilistic inference engine Ace, which can be obtained at `http://reasoning.cs.ucla.edu/ace/` (03.2011).
- The PHCA-to-COP translator, which may be obtained by contacting the author `maierpa@in.tum.de`, or directly from its developers, the authors of [129].
- One of these constraint optimizers:
  - toolbar: `https://mulcyber.toulouse.inra.fr/projects/toolbar` (03.2011).
  - toulbar2: `https://mulcyber.toulouse.inra.fr/projects/toulbar2` (03.2011).

A.2. Model-Based Diagnosis Approach

We developed the Python script that combines the elements of our library roughly as shown in figure 6.1 to compute plan assessment results. It takes as input a PHCA model translated to a COP, a plan $\mathcal{P}$ and observations $\mathbf{o}^{0:t}$ and generates as output a) diagnoses in form of a list of trajectories sorted by their probability and b) success probabilities for given goals. Goals are either specified in a separate configuration file or on the command line. The script may be called as follows:

```
runPAExp.py --schedule=<file containing 𝒫>
    --observations=<file containing o^{0:t}>
    [--query=<variable assignment>] <COP file>
```

An example call is

```
runPAExp.py --schedule=operations.schedule
 --observations=observations.obs copModel.xml
```

The file copModel.xml is the output of the COP translator developed by Tsoline Mikaelian for the work presented in [129]. We encapsulated this translator in another Python script that is called with

```
runHCA.py --timeSteps=N <PHCA model file>
```

An example call is

```
runHCA.py --timeSteps=9 model.hca
```

More information on using these scripts can be obtained by calling them with the `--help` option. This option works on all scripts that are part of our implementation.

Listing A.1 shows an example configuration file. Its elements constitute the input for plan assessment and some global parameters like the solver to be employed. Some elements can also be specified on the command line, for instance the path to the observations file. In detail, the elements are the following:

K Number of solutions to compute. *Not available on command line.*

ESTIMATION_ONLY When set to `True`, estimation without time window filtering (described in section 7.1) is forced. Has no effect when the length of the time window is equal to the length of $\mathcal{P}$.

SCHEDULE_LENGTH The length of $\mathcal{P}$ in time steps. Has to be given only if ESTIMATION_ONLY is set to true. Otherwise the schedule is assumed to be as long as the number of time steps encoded in the COP (which is unfolded over the required number of time steps).

SCHEDULE_FP The path to the file containing $\mathcal{P}$.

OBSERVATIONS_FP The path to the observations file.

SOLVER Solver to use. Possible values are

```
estimator.ToolbarSolver()
estimator.SimpleToulbar2Solver()
estimator.MemoryMeasuringToulbar2Solver().
```

Not available on command line.

estimator.MBE_I Parameter that controls the accuracy of the heuristic for the A* search used in Toolbar. *Not available on command line.*

PRODUCTS List of goals, for instance products that must be finished. This element's name is out-dated, apart from special product goal objects, generic PAQuery objects may be given. They allow to ask for the probability of arbitrary variable assignments, for example `PAQuery("PROCESSING_IMAGE__5=MARKED")`. In general, a goal (l, t) with PHCA location l expected to be marked at t must be given as "<name of l>`__`<t>`=MARKED`". The more specific product goal objects take as parameters the name of the sub-PHCA modeling the product, the start time and latest allowed finishing time. Start time, however, is not being used at the moment. *Not available on command line.*

Listings A.1 and A.3 show examples for observation and plan files.

A final thing to note is that `runHCA.py` may also accept mixed schedule and observation files. These are files in schedule format that also contain assignments to observation variables. We used such files for the scripts that run our experiments. In this case it suffices to provide this file as `--schedule` parameter.

A.3. Probabilistic Reasoning Approach

For our probabilistic reasoning approach we implemented a translator from PHCA models to BLNs. The command line call is

```
from planAssessment.planAssessment import Product
from estimation import estimator


K = 1800
ESTIMATION_ONLY = False
SCHEDULE_LENGTH = None
SCHEDULE_FP = None
OBSERVATIONS_FP = None
SOLVER = estimator.SimpleToulbar2Solver()
estimator.MBE_I = 21


PRODUCTS = [Product("Maze0", 0, 4),
            Product("Maze1", 0, 9),
            Product("Robot0", 0, 7)]
```

Listing A.1: Example configuration file for the `runPAExp.py` Python script for computing plan assessment results. It's also written in Python code.

```
PFORCE__0=NONE
PFORCE__1=NONE
PFORCE__2=NONE
PFORCE__3=NONE
PFORCE__4=NONE
PFORCE__5=NONE
PFORCE__6=NONE
PFORCE__7=NONE
PFORCE__8=NONE
PFORCE__9=NONE
PFORCE__10=NONE
PFORCE__11=HIGH
PFORCE__12=HIGH
PFORCE__13=HIGH
PFORCE__14=HIGH
PFORCE__15=HIGH
```

Listing A.2: Example observations file for a manufacturing scenario as described in 2.1.3. The observation variable PFORCE encodes whether a force alarm was observed (HIGH) or not (NONE).

```
NoProduct , NoComponent , 0: MAZE0 - WORKER = OK
NoProduct , NoComponent , 0: MAZE1 - WORKER = OK
NoProduct , NoComponent , 0: ROBOT0 - WORKER = OK
NoProduct , NoComponent , 0: ASSEMBLY - LINK - HOLES = OK
NoProduct , NoComponent , 0: ASSEMBLY - CMD = NOCOMMAND
Maze0 , Machining0 ,       0: MACHINING0 - CMD = MILL
Maze1 , Machining1 ,       0: MACHINING1 - CMD = MILL

NoProduct , NoComponent , 1: ROBOT0 - WORKER = OK
NoProduct , NoComponent , 1: ASSEMBLY - LINK - HOLES = OK
NoProduct , NoComponent , 1: MACHINING0 - CMD = NOCOMMAND
Maze1 , Machining1 ,       1: MACHINING1 - CMD = MILL
Maze0 , Assembly ,         1: ASSEMBLY - CMD = ASSEMBLE - COVER

NoProduct , NoComponent , 2: ROBOT0 - WORKER = OK
NoProduct , NoComponent , 2: MACHINING0 - CMD = NOCOMMAND
Maze1 , Machining1 ,       2: MACHINING1 - CMD = MILL
Maze0 , Assembly ,         2: ASSEMBLY - CMD = ASSEMBLE - PINS

NoProduct , NoComponent , 3: ROBOT0 - WORKER = OK
Robot0 , Machining0 ,      3: MACHINING0 - CMD = MILL
Maze1 , Machining1 ,       3: MACHINING1 - CMD = MILL

...
```

Listing A.3: Part of an example plan/schedule file. It's the same operation sequence as shown in listing 4.1, reprinted for convenience here.

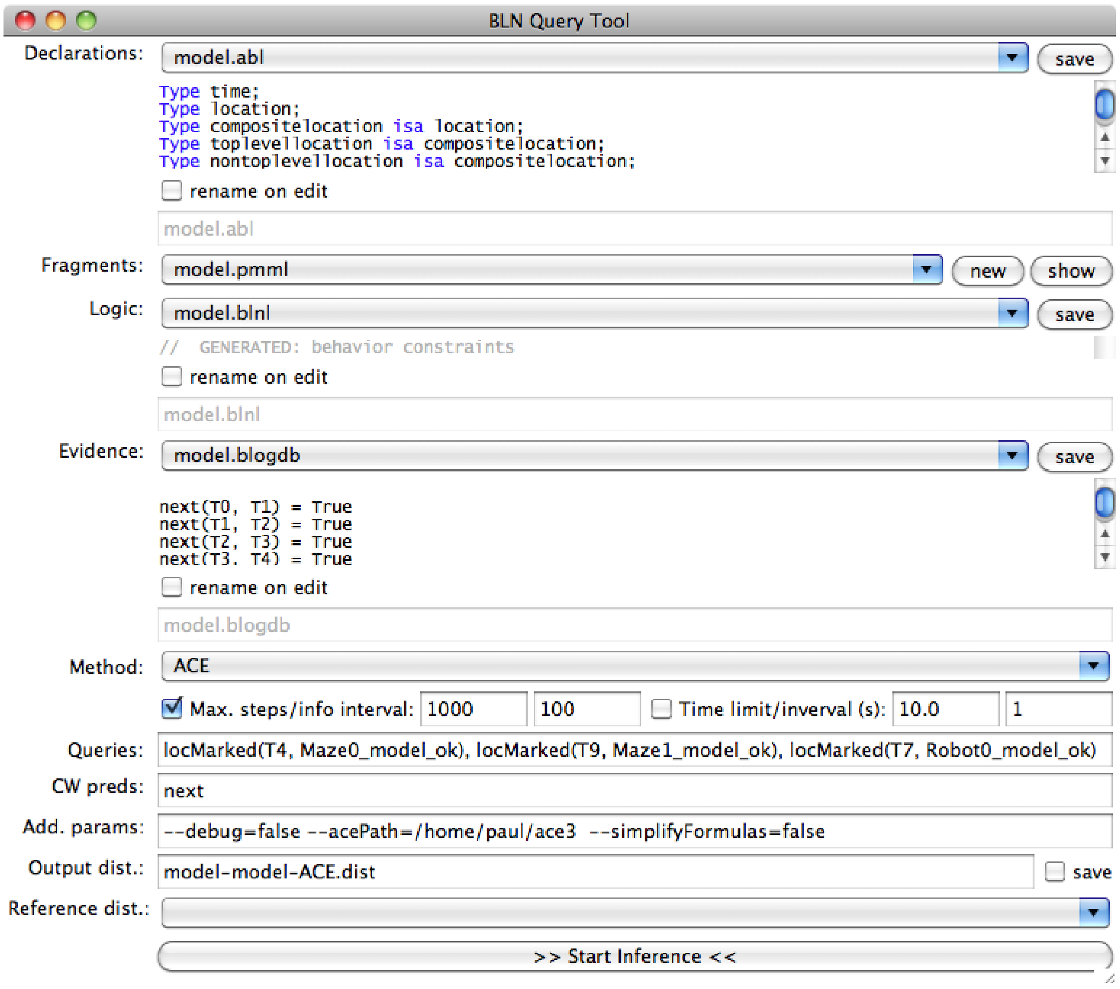
```
phca2bln.py --schedule=<file containing $\mathcal{P}$>
     --observations=<file containing $\mathbf{o}^{0:t}$>
     --buildPath=<path to store output at>
     <path to phca file>
```

For a concrete PHCA model, the call would for example be

```
phca2bln.py --schedule=operations.schedule
     --observations=observations.obs
     --buildPath=./output
     model.hca
```

The translator creates as output four files, storing respectively the definitions $\mathcal{D}$, the fragments $\mathcal{F}$, the logical formulas $\mathcal{L}$ and the knowledge base DB. The above example call would result in the four files `model.abl`, `model.pmml`, `model.blnl` and `model.blogdb` in the local directory `output`. These four files can now be fed into the BLN framework to compute success probabilities, for example with Ace as backend inference engine.

One tool in particular of this framework is `blnquery`, which allows to load arbitrary models and compute probabilities for arbitrary goals. It offers a GUI, of which a screenshot is shown in figure A.1. The four model files are chosen from the drop-down selectors “Declarations”, “Fragments”, “Logic” and “Evidence”. The tool automatically recognizes all models in the working directory, typically the one from which the tool was started. The selector “Method” allows to an algorithm or external engine, such as Ace. Finally, in the text field “Queries”, the goals can be specified as comma-separated queries for the probability of bln predicates being true. For plan assessment, the *locMarked*() predicate is important: To specify, for instance, the goal (l, t), the query locMarked(T$<t>$, $<$name of $l>$) must be specified. The goal queries for our main example described in section 2.1.3 are shown in the screenshot.



Declarations: model.abl

```
Type time;
Type location;
Type compositelocation isa location;
Type toplevellocation isa compositelocation;
Type nontoplevellocation isa compositelocation;
```

☐ rename on edit

model.abl

Fragments: model.pmml

Logic: model.blnl

```
//  GENERATED: behavior constraints
```

☐ rename on edit

model.blnl

Evidence: model.blogdb

```
next(T0, T1) = True
next(T1, T2) = True
next(T2, T3) = True
next(T3, T4) = True
```

☐ rename on edit

model.blogdb

Method: ACE

✓ Max. steps/info interval: 1000 100 ☐ Time limit/inverval (s): 10.0 1

Queries: locMarked(T4, Maze0_model_ok), locMarked(T9, Maze1_model_ok), locMarked(T7, Robot0_model_ok)

CW preds: next

Add. params: --debug=false --acePath=/home/paul/ace3 --simplifyFormulas=false

Output dist.: model-model-ACE.dist ☐ save

Reference dist.:

Figure A.1.: Screenshot of the BLN tool by Dominik Jain that allows, among other things, to compute the probability of arbitrary BLN predicates being true (see sections 3.7 and 5.3).

B. PHCA Models

B.1. Model for Cognitive Factory Example

Here we present the input plan $\mathcal{P}$, the observations and the graphical and code description of the PHCA model we used to compute results for our main example described in section 2.1.3. The following variable names in the code have been changed for graphical representation (star indicates an arbitrary sequence of strings):

MILL*: Machining*.

MAZE0-LINK-HOLES/ASSEMBLY-LINK-HOLES: Holes.

PFORCE: Force.

*CMD: Cmd.

*WORKER: Worker.

```
NoProduct, NoComponent, 0: MAZE0-WORKER=OK
NoProduct, NoComponent, 0: MAZE1-WORKER=OK
NoProduct, NoComponent, 0: ROBOT0-WORKER=OK
NoProduct, NoComponent, 0: ASSEMBLY-LINK-HOLES=OK
NoProduct, NoComponent, 0: ASSEMBLY-CMD=NOCOMMAND
Maze0, Mill0,           0: MILL0-CMD=MILL
Maze1, Mill1,           0: MILL1-CMD=MILL

NoProduct, NoComponent, 1: ROBOT0-WORKER=OK
NoProduct, NoComponent, 1: ASSEMBLY-LINK-HOLES=OK
NoProduct, NoComponent, 1: MILL0-CMD=NOCOMMAND
Maze1, Mill1,           1: MILL1-CMD=MILL
Maze0, Assembly,        1: ASSEMBLY-CMD=ASSEMBLE-COVER

NoProduct, NoComponent, 2: ROBOT0-WORKER=OK
NoProduct, NoComponent, 2: MILL0-CMD=NOCOMMAND
Maze1, Mill1,           2: MILL1-CMD=MILL
Maze0, Assembly,        2: ASSEMBLY-CMD=ASSEMBLE-PINS

NoProduct, NoComponent, 3: ROBOT0-WORKER=OK
Robot0, Mill0,          3: MILL0-CMD=MILL
Maze1, Mill1,           3: MILL1-CMD=MILL

NoProduct, NoComponent, 4: MAZE0-WORKER=OK
Robot0, Mill0,          4: MILL0-CMD=MILL
NoProduct, NoComponent, 4: MILL1-CMD=NOCOMMAND
NoProduct, NoComponent, 4: ASSEMBLY-CMD=NOCOMMAND

NoProduct, NoComponent, 5: MAZE0-WORKER=OK
```

```
NoProduct , NoComponent , 5: MAZE1 - WORKER = OK
NoProduct , NoComponent , 5: MILL0 - CMD = NOCOMMAND
NoProduct , NoComponent , 5: MILL1 - CMD = NOCOMMAND
NoProduct , NoComponent , 5: ASSEMBLY - LINK - HOLES = OK
Robot0 , Assembly ,       5: ASSEMBLY - CMD = ASSEMBLE - ROBOT

NoProduct , NoComponent , 6: MAZE0 - WORKER = OK
NoProduct , NoComponent , 6: MAZE1 - WORKER = OK
NoProduct , NoComponent , 6: MILL0 - CMD = NOCOMMAND
NoProduct , NoComponent , 6: MILL1 - CMD = NOCOMMAND
Maze1 , Assembly ,        6: ASSEMBLY - CMD = ASSEMBLE - COVER

NoProduct , NoComponent , 7: MAZE0 - WORKER = OK
NoProduct , NoComponent , 7: ROBOT0 - WORKER = OK
NoProduct , NoComponent , 7: MILL0 - CMD = NOCOMMAND
NoProduct , NoComponent , 7: MILL1 - CMD = NOCOMMAND
Maze1 , Assembly ,        7: ASSEMBLY - CMD = ASSEMBLE - PINS

NoProduct , NoComponent , 8: MAZE0 - WORKER = OK
NoProduct , NoComponent , 8: ROBOT0 - WORKER = OK
NoProduct , NoComponent , 8: MILL0 - CMD = NOCOMMAND
NoProduct , NoComponent , 8: MILL1 - CMD = NOCOMMAND
NoProduct , NoComponent , 8: ASSEMBLY - CMD = NOCOMMAND

NoProduct , NoComponent , 9: MAZE0 - WORKER = OK
NoProduct , NoComponent , 9: MAZE1 - WORKER = OK
NoProduct , NoComponent , 9: ROBOT0 - WORKER = OK
NoProduct , NoComponent , 9: MILL0 - CMD = NOCOMMAND
NoProduct , NoComponent , 9: MILL1 - CMD = NOCOMMAND
NoProduct , NoComponent , 9: ASSEMBLY - CMD = NOCOMMAND
NoProduct , NoComponent , 9: ASSEMBLY - LINK - HOLES = OK
```

Listing B.1: Input plan $\mathcal{P}$ for the cognitive factory example from section 2.1.3.

```
PFORCE__0=NONE
PFORCE__1=NONE
PFORCE__2=NONE
PFORCE__3=HIGH
```

Listing B.2: Input observations for the cognitive factory example from section 2.1.3.

```
VARIABLE - DOMAIN - TYPE - DEFINITIONS

    VARIABLE - DOMAIN - TYPE COMPONENT - STATUS (OK FAULTY)
    VARIABLE - DOMAIN - TYPE FORCE - STATUS (NONE NORMAL HIGH)
    VARIABLE - DOMAIN - TYPE MILL - COMMAND (MILL NOCOMMAND)
    VARIABLE - DOMAIN - TYPE ASSEMBLE - COMMAND (ASSEMBLE - COVER ASSEMBLE - PINS
                                               ASSEMBLE - ROBOT NOCOMMAND)


VARIABLE - DEFINITIONS

    VARIABLE MILL0 - CMD OF - TYPE CONTROL WITH - RANGE MILL - COMMAND
    VARIABLE MILL1 - CMD OF - TYPE CONTROL WITH - RANGE MILL - COMMAND
    VARIABLE MILL0 - C OF - TYPE DEPENDENT WITH - RANGE COMPONENT - STATUS
    VARIABLE MILL1 - C OF - TYPE DEPENDENT WITH - RANGE COMPONENT - STATUS
    VARIABLE ASSEMBLY - CMD OF - TYPE CONTROL WITH - RANGE ASSEMBLE - COMMAND
    VARIABLE MAZE0 - WORKER OF - TYPE CONTROL WITH - RANGE COMPONENT - STATUS
    VARIABLE MAZE0 - LINK - HOLES OF - TYPE DEPENDENT WITH - RANGE COMPONENT - STATUS

    VARIABLE MAZE1 - WORKER OF - TYPE CONTROL WITH - RANGE COMPONENT - STATUS
    VARIABLE MAZE1 - LINK - HOLES OF - TYPE DEPENDENT WITH - RANGE COMPONENT - STATUS
    VARIABLE ROBOT0 - WORKER OF - TYPE CONTROL WITH - RANGE COMPONENT - STATUS
    VARIABLE ASSEMBLY - C OF - TYPE DEPENDENT WITH - RANGE COMPONENT - STATUS
    VARIABLE ASSEMBLY - LINK - HOLES OF - TYPE DEPENDENT WITH - RANGE COMPONENT - STATUS
    VARIABLE PFORCE OF - TYPE OBSERVABLE WITH - RANGE FORCE - STATUS

```

```
MODEL-DEFINITIONS

(root COMPOSITE
    (CHILDREN
            ((maze0-model COMPOSITE
           (CHILDREN
                (STARTING (maze0-model-ok PRIMITIVE
                   (BEHAVIOR-CONSTRAINT (AND (OR (MAZE0-LINK-HOLES = OK)))) ))
                ((maze0-model-faulty PRIMITIVE
                   (BEHAVIOR-CONSTRAINT (AND (OR (MAZE0-LINK-HOLES = FAULTY)))) ))
            )
                         (TRANSITIONS
                (TRANS-FROM-TO-GUARD-PROB maze0-model-ok maze0-model-ok
                    (AND (MAZE0-WORKER = OK)) 1.0)
                 (TRANS-FROM-TO-PROB maze0-model-faulty maze0-model-faulty 1.0)

                (TRANS-FROM-TO-GUARD-PROB maze0-model-ok maze0-model-faulty
                     (AND (MAZE0-WORKER = FAULTY)) 1.0))
        ))

        ((maze1-model COMPOSITE
           (CHILDREN
                (STARTING (maze1-model-ok PRIMITIVE
                   (BEHAVIOR-CONSTRAINT (AND (OR (MAZE1-LINK-HOLES = OK)))) ))
                ((maze1-model-faulty PRIMITIVE
                   (BEHAVIOR-CONSTRAINT (AND (OR (MAZE1-LINK-HOLES = FAULTY)))) ))
            )
                         (TRANSITIONS
                (TRANS-FROM-TO-GUARD-PROB maze1-model-ok maze1-model-ok
                    (AND (MAZE1-WORKER = OK)) 1.0)
                 (TRANS-FROM-TO-PROB maze1-model-faulty maze1-model-faulty 1.0)

                (TRANS-FROM-TO-GUARD-PROB maze1-model-ok maze1-model-faulty
                     (AND (MAZE1-WORKER = FAULTY)) 1.0))
        ))

        ((robot0-model COMPOSITE
            (CHILDREN
                (STARTING (robot0-model-ok PRIMITIVE))
               ((robot0-model-faulty PRIMITIVE))
            )
                         (TRANSITIONS
                (TRANS-FROM-TO-GUARD-PROB robot0-model-ok robot0-model-ok
                     (AND (ROBOT0-WORKER = OK)) 1.0)
                 (TRANS-FROM-TO-PROB robot0-model-faulty robot0-model-faulty 1.0)

                (TRANS-FROM-TO-GUARD-PROB robot0-model-ok robot0-model-faulty
                     (AND (ROBOT0-WORKER = FAULTY)) 1.0))
        ))

        ((mill0-model COMPOSITE
            (CHILDREN
                (STARTING (mill0-idle PRIMITIVE
                     (BEHAVIOR-CONSTRAINT (AND (OR (MILL0-C = OK)))) ))
                ((mill0-mill PRIMITIVE
                     (BEHAVIOR-CONSTRAINT (AND (OR (MILL0-C = OK)))) ))

                ((mill0-fail-drillbroken PRIMITIVE
                    (BEHAVIOR-CONSTRAINT (AND (OR (MILL0-C = FAULTY)))) ))
            )

            (TRANSITIONS
                (TRANS-FROM-TO-GUARD-PROB mill0-idle mill0-mill
                     (AND (MILL0-CMD = MILL)) 0.99)
                (TRANS-FROM-TO-GUARD-PROB mill0-idle mill0-idle
                     (AND (MILL0-CMD = NOCOMMAND)) 1.0)
                (TRANS-FROM-TO-GUARD-PROB mill0-idle mill0-fail-drillbroken
                    (AND (MILL0-CMD = MILL)) 0.01)

        (TRANS-FROM-TO-GUARD-PROB mill0-mill mill0-mill
             (AND (MILL0-CMD = MILL)) 0.99)
        (TRANS-FROM-TO-GUARD-PROB mill0-mill mill0-idle
             (AND (MILL0-CMD = NOCOMMAND)) 1.0)
        (TRANS-FROM-TO-GUARD-PROB mill0-mill mill0-fail-drillbroken
           (AND (MILL0-CMD = MILL)) 0.01)

        (TRANS-FROM-TO-PROB mill0-fail-drillbroken mill0-fail-drillbroken
           1.0)
    )
))

((mill1-model COMPOSITE
    (CHILDREN
        (STARTING (mill1-idle PRIMITIVE
             (BEHAVIOR-CONSTRAINT (AND (OR (MILL1-C = OK)))) ))
        ((mill1-mill PRIMITIVE
             (BEHAVIOR-CONSTRAINT (AND (OR (MILL1-C = OK)))) ))

        ((mill1-fail-drillbroken PRIMITIVE
            (BEHAVIOR-CONSTRAINT (AND (OR (MILL1-C = FAULTY)))) ))
    )

    (TRANSITIONS
        (TRANS-FROM-TO-GUARD-PROB mill1-idle mill1-mill
            (AND (MILL1-CMD = MILL)) 0.99)
        (TRANS-FROM-TO-GUARD-PROB mill1-idle mill1-idle
             (AND (MILL1-CMD = NOCOMMAND)) 1.0)
        (TRANS-FROM-TO-GUARD-PROB mill1-idle mill1-fail-drillbroken
            (AND (MILL1-CMD = MILL)) 0.01)

        (TRANS-FROM-TO-GUARD-PROB mill1-mill mill1-mill
             (AND (MILL1-CMD = MILL)) 0.99)
        (TRANS-FROM-TO-GUARD-PROB mill1-mill mill1-idle
             (AND (MILL1-CMD = NOCOMMAND)) 1.0)
        (TRANS-FROM-TO-GUARD-PROB mill1-mill mill1-fail-drillbroken
             (AND (MILL1-CMD = MILL)) 0.01)

        (TRANS-FROM-TO-PROB mill1-fail-drillbroken mill1-fail-drillbroken
             1.0)
    )
))

((assembly-model COMPOSITE
    (CHILDREN
        (STARTING (assembly-idle PRIMITIVE
             (BEHAVIOR-CONSTRAINT (AND (OR (PFORCE = NONE)))) ))
       ((assembly-cover PRIMITIVE
            (BEHAVIOR-CONSTRAINT (AND (OR (PFORCE = NONE)))) ))
       ((assembly-robot PRIMITIVE
            (BEHAVIOR-CONSTRAINT (AND (OR (PFORCE = NONE)))) ))
        ((assembly-pins PRIMITIVE
            (BEHAVIOR-CONSTRAINT
                    (AND (OR (ASSEMBLY-LINK-HOLES = FAULTY)
                             (ASSEMBLY-C = FAULTY) (PFORCE = NORMAL))
                         (OR (ASSEMBLY-LINK-HOLES = OK) (PFORCE = HIGH))
                         (OR (ASSEMBLY-C = OK) (PFORCE = HIGH)))) ))
        (STARTING (assembly-status-model COMPOSITE

             (CHILDREN
                (STARTING (assembly-status-ok PRIMITIVE
                 (BEHAVIOR-CONSTRAINT (AND (OR (ASSEMBLY-C = OK)))) ))
                 ((assembly-status-faulty PRIMITIVE
                 (BEHAVIOR-CONSTRAINT (AND (OR (ASSEMBLY-C = FAULTY)))) ))
             )
            (TRANSITIONS
            (TRANS-FROM-TO-PROB assembly-status-ok assembly-status-ok
                 0.9995)
            (TRANS-FROM-TO-PROB assembly-status-ok assembly-status-faulty
                         0.0005)
                (TRANS-FROM-TO-PROB assembly-status-faulty assembly-status-faulty
                            1.0)
                    )
                ))
            )

            (TRANSITIONS
                (TRANS-FROM-TO-GUARD-PROB assembly-idle assembly-idle
                    (AND (ASSEMBLY-CMD = NOCOMMAND)) 1.0)
                (TRANS-FROM-TO-GUARD-PROB assembly-idle assembly-cover
                    (AND (ASSEMBLY-CMD = ASSEMBLE-COVER)) 1.0)
             (TRANS-FROM-TO-GUARD-PROB assembly-idle assembly-pins
                    (AND (ASSEMBLY-CMD = ASSEMBLE-PINS)) 1.0)
                (TRANS-FROM-TO-GUARD-PROB assembly-idle assembly-robot
                    (AND (ASSEMBLY-CMD = ASSEMBLE-ROBOT)) 1.0)

                (TRANS-FROM-TO-GUARD-PROB assembly-cover assembly-cover
                    (AND (ASSEMBLY-CMD = ASSEMBLE-COVER)) 1.0)
                (TRANS-FROM-TO-GUARD-PROB assembly-cover assembly-pins
                    (AND (ASSEMBLY-CMD = ASSEMBLE-PINS)) 1.0)
                (TRANS-FROM-TO-GUARD-PROB assembly-cover assembly-idle
                    (AND (ASSEMBLY-CMD = NOCOMMAND)) 1.0)

                (TRANS-FROM-TO-GUARD-PROB assembly-pins assembly-pins
                    (AND (ASSEMBLY-CMD = ASSEMBLE-PINS)) 1.0)
                (TRANS-FROM-TO-GUARD-PROB assembly-pins assembly-idle
                    (AND (ASSEMBLY-CMD = NOCOMMAND)) 1.0)

                (TRANS-FROM-TO-GUARD-PROB assembly-robot assembly-robot
                    (AND (ASSEMBLY-CMD = ASSEMBLE-ROBOT)) 1.0)
                (TRANS-FROM-TO-GUARD-PROB assembly-robot assembly-cover
                    (AND (ASSEMBLY-CMD = ASSEMBLE-COVER)) 1.0)
                (TRANS-FROM-TO-GUARD-PROB assembly-robot assembly-idle
                    (AND (ASSEMBLY-CMD = NOCOMMAND)) 1.0)
            )
        ))


    )

    (TRANSITIONS
   )
)
```

Listing B.3: PHCA description code for the cognitive factory example from section 2.1.3.

Figure B.2.: PHCA model used in the kitchen example in section 2.2.

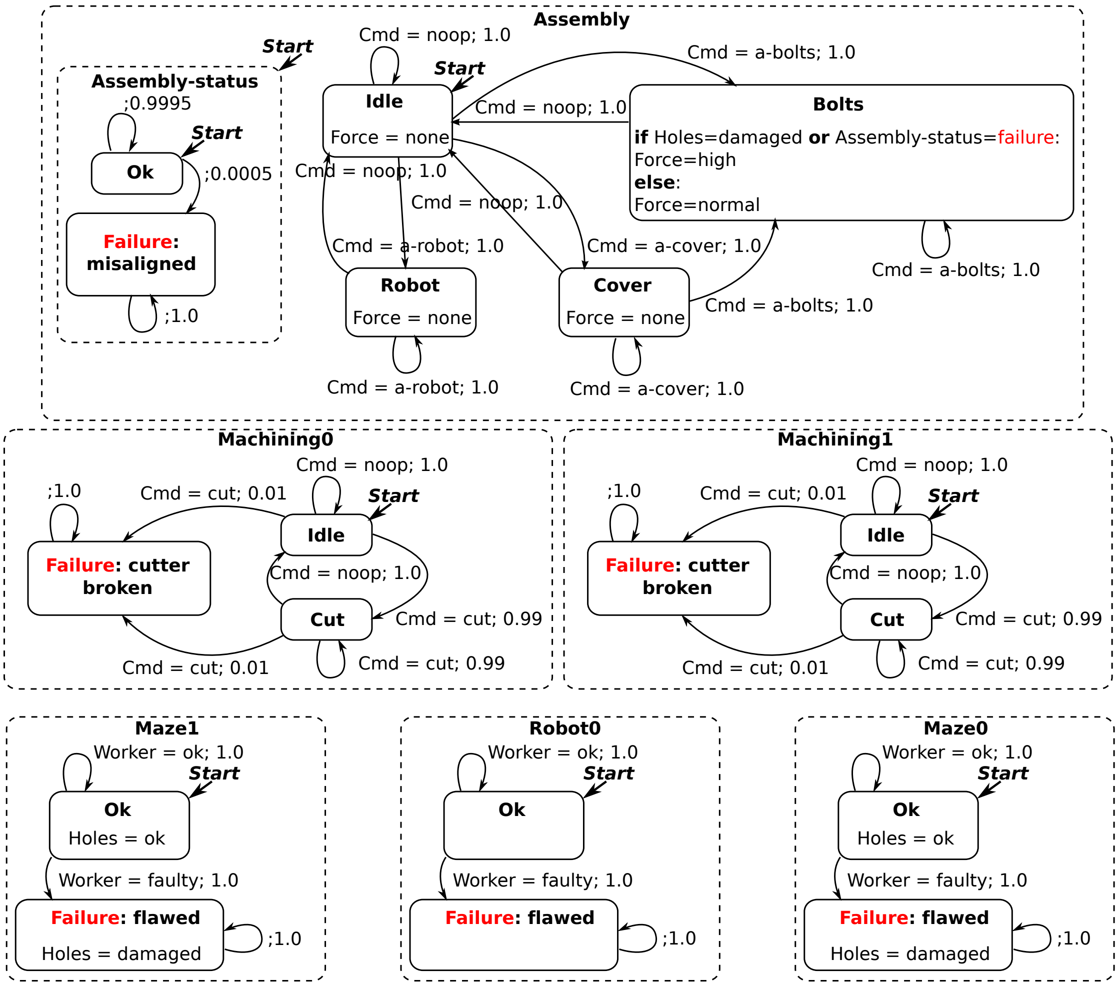


Figure B.1.: The PHCA model for the example instance from section 2.1.3.

B.2. Model for Household Robot Example

Here we show the model for the kitchen example described in section 2.2. Its graphical representation is shown in figure B.3, its code in listing B.6. Note that, due to the simplifications we made, the commands that tell the robot to move, place or pick are merely place holders. The implementation requires to give some sort of command.

```
NoProduct , NoComponent , 0: PLATE0 - LINK - CMD=OK
NoProduct , NoComponent , 0: PLATE1 - LINK - CMD=OK
NoProduct , NoComponent , 0: CUP0 - LINK - CMD=OK
NoProduct , NoComponent , 0: CUP1 - LINK - CMD=OK
```

```
NoProduct, NoComponent, 0: PLATE0-LINK-CMD-2=OK
NoProduct, NoComponent, 0: PLATE1-LINK-CMD-2=OK
NoProduct, NoComponent, 0: CUP0-LINK-CMD-2=OK
NoProduct, NoComponent, 0: CUP1-LINK-CMD-2=OK
NoProduct, NoComponent, 0: ROBOT-CMD=NOCOMMAND
Plate0,    Left-arm,    0: LEFT-ARM-CMD=PICKUP
Plate0,    Right-arm,   0: RIGHT-ARM-CMD=PICKUP
NoProduct, NoComponent, 0: TABLE-CMD=NOCOMMAND

NoProduct, NoComponent, 1: PLATE1-LINK-CMD=OK
NoProduct, NoComponent, 1: CUP0-LINK-CMD=OK
NoProduct, NoComponent, 1: CUP1-LINK-CMD=OK
NoProduct, NoComponent, 1: PLATE1-LINK-CMD-2=OK
NoProduct, NoComponent, 1: CUP0-LINK-CMD-2=OK
NoProduct, NoComponent, 1: CUP1-LINK-CMD-2=OK
Plate0,    Left-arm,    1: LEFT-ARM-CMD=NOCOMMAND
Plate0,    Right-arm,   1: RIGHT-ARM-CMD=NOCOMMAND
NoProduct, NoComponent, 1: ROBOT-CMD=MOVE
NoProduct, NoComponent, 1: TABLE-CMD=NOCOMMAND

NoProduct, NoComponent, 2: PLATE1-LINK-CMD=OK
NoProduct, NoComponent, 2: CUP0-LINK-CMD=OK
NoProduct, NoComponent, 2: CUP1-LINK-CMD=OK
NoProduct, NoComponent, 2: PLATE1-LINK-CMD-2=OK
NoProduct, NoComponent, 2: CUP0-LINK-CMD-2=OK
NoProduct, NoComponent, 2: CUP1-LINK-CMD-2=OK
NoProduct, NoComponent, 2: ROBOT-CMD=NOCOMMAND
NoProduct, NoComponent, 2: LEFT-ARM-CMD=PLACE
NoProduct, NoComponent, 2: RIGHT-ARM-CMD=PLACE
Plate0,    Table,       2: TABLE-CMD=SCAN

NoProduct, NoComponent, 3: PLATE1-LINK-CMD=OK
NoProduct, NoComponent, 3: CUP0-LINK-CMD=OK
NoProduct, NoComponent, 3: CUP1-LINK-CMD=OK
NoProduct, NoComponent, 3: PLATE0-LINK-CMD-2=OK
NoProduct, NoComponent, 3: PLATE1-LINK-CMD-2=OK
NoProduct, NoComponent, 3: CUP0-LINK-CMD-2=OK
NoProduct, NoComponent, 3: CUP1-LINK-CMD-2=OK
NoProduct, NoComponent, 3: LEFT-ARM-CMD=NOCOMMAND
NoProduct, NoComponent, 3: RIGHT-ARM-CMD=NOCOMMAND
NoProduct, NoComponent, 3: ROBOT-CMD=MOVE
NoProduct, NoComponent, 3: TABLE-CMD=NOCOMMAND

NoProduct, NoComponent, 4: PLATE0-LINK-CMD=OK
NoProduct, NoComponent, 4: PLATE1-LINK-CMD=OK
NoProduct, NoComponent, 4: CUP0-LINK-CMD=OK
NoProduct, NoComponent, 4: CUP1-LINK-CMD=OK
NoProduct, NoComponent, 4: PLATE0-LINK-CMD-2=OK
NoProduct, NoComponent, 4: PLATE1-LINK-CMD-2=OK
NoProduct, NoComponent, 4: CUP0-LINK-CMD-2=OK
NoProduct, NoComponent, 4: CUP1-LINK-CMD-2=OK
NoProduct, NoComponent, 4: ROBOT-CMD=NOCOMMAND
Cup0,      Left-arm,    4: LEFT-ARM-CMD=PICKUP
Cup1,      Right-arm,   4: RIGHT-ARM-CMD=PICKUP
NoProduct, NoComponent, 4: TABLE-CMD=NOCOMMAND

NoProduct, NoComponent, 5: PLATE0-LINK-CMD-2=OK
NoProduct, NoComponent, 5: PLATE1-LINK-CMD-2=OK
NoProduct, NoComponent, 5: PLATE0-LINK-CMD=OK
NoProduct, NoComponent, 5: PLATE1-LINK-CMD=OK
NoProduct, NoComponent, 5: CUP0-LINK-CMD-2=OK
NoProduct, NoComponent, 5: CUP1-LINK-CMD=OK
Cup0,      Left-arm,    5: LEFT-ARM-CMD=NOCOMMAND
Cup1,      Right-arm,   5: RIGHT-ARM-CMD=NOCOMMAND
NoProduct, NoComponent, 5: ROBOT-CMD=MOVE
NoProduct, NoComponent, 5: TABLE-CMD=NOCOMMAND

NoProduct, NoComponent, 6: PLATE0-LINK-CMD=OK
NoProduct, NoComponent, 6: PLATE1-LINK-CMD=OK
```

```
NoProduct, NoComponent, 6: PLATE0-LINK-CMD-2=OK
NoProduct, NoComponent, 6: PLATE1-LINK-CMD-2=OK
NoProduct, NoComponent, 6: CUP0-LINK-CMD-2=OK
NoProduct, NoComponent, 6: CUP1-LINK-CMD=OK
NoProduct, NoComponent, 6: LEFT-ARM-CMD=PLACE
Cup1,      Right-arm,   6: RIGHT-ARM-CMD=NOCOMMAND
NoProduct, NoComponent, 6: ROBOT-CMD=NOCOMMAND
Cup0,      Table,       6: TABLE-CMD=SCAN

NoProduct, NoComponent, 7: PLATE0-LINK-CMD=OK
NoProduct, NoComponent, 7: PLATE1-LINK-CMD=OK
NoProduct, NoComponent, 7: PLATE0-LINK-CMD-2=OK
NoProduct, NoComponent, 7: PLATE1-LINK-CMD-2=OK
NoProduct, NoComponent, 7: CUP0-LINK-CMD-2=OK
NoProduct, NoComponent, 7: CUP1-LINK-CMD=OK
NoProduct, NoComponent, 7: LEFT-ARM-CMD=NOCOMMAND
NoProduct, NoComponent, 7: RIGHT-ARM-CMD=NOCOMMAND
NoProduct, NoComponent, 7: ROBOT-CMD=NOCOMMAND
Cup1,      Table,       7: TABLE-CMD=SCAN

NoProduct, NoComponent, 8: CUP0-LINK-CMD=OK
NoProduct, NoComponent, 8: CUP0-LINK-CMD-2=OK
NoProduct, NoComponent, 8: CUP1-LINK-CMD-2=OK
NoProduct, NoComponent, 8: PLATE0-LINK-CMD=OK
NoProduct, NoComponent, 8: PLATE1-LINK-CMD=OK
NoProduct, NoComponent, 8: PLATE0-LINK-CMD-2=OK
NoProduct, NoComponent, 8: PLATE1-LINK-CMD-2=OK
NoProduct, NoComponent, 8: LEFT-ARM-CMD=NOCOMMAND
NoProduct, NoComponent, 8: RIGHT-ARM-CMD=NOCOMMAND
NoProduct, NoComponent, 8: ROBOT-CMD=MOVE
NoProduct, NoComponent, 8: TABLE-CMD=NOCOMMAND

NoProduct, NoComponent, 9: CUP0-LINK-CMD=OK
NoProduct, NoComponent, 9: CUP1-LINK-CMD=OK
NoProduct, NoComponent, 9: CUP0-LINK-CMD-2=OK
NoProduct, NoComponent, 9: CUP1-LINK-CMD-2=OK
NoProduct, NoComponent, 9: PLATE0-LINK-CMD=OK
NoProduct, NoComponent, 9: PLATE1-LINK-CMD=OK
NoProduct, NoComponent, 9: PLATE0-LINK-CMD-2=OK
NoProduct, NoComponent, 9: PLATE1-LINK-CMD-2=OK
Plate1,    Left-arm,    9: LEFT-ARM-CMD=PICKUP
Plate1,    Right-arm,   9: RIGHT-ARM-CMD=PICKUP
NoProduct, NoComponent, 9: ROBOT-CMD=NOCOMMAND
NoProduct, NoComponent, 9: TABLE-CMD=NOCOMMAND

NoProduct, NoComponent, 10: CUP0-LINK-CMD=OK
NoProduct, NoComponent, 10: CUP1-LINK-CMD=OK
NoProduct, NoComponent, 10: CUP0-LINK-CMD-2=OK
NoProduct, NoComponent, 10: CUP1-LINK-CMD-2=OK
NoProduct, NoComponent, 10: PLATE0-LINK-CMD=OK
NoProduct, NoComponent, 10: PLATE0-LINK-CMD-2=OK
Plate1,    Left-arm,    10: LEFT-ARM-CMD=NOCOMMAND
Plate1,    Right-arm,   10: RIGHT-ARM-CMD=NOCOMMAND
NoProduct, NoComponent, 10: ROBOT-CMD=MOVE
NoProduct, NoComponent, 10: TABLE-CMD=NOCOMMAND

NoProduct, NoComponent, 11: CUP0-LINK-CMD=OK
NoProduct, NoComponent, 11: CUP1-LINK-CMD=OK
NoProduct, NoComponent, 11: CUP0-LINK-CMD-2=OK
NoProduct, NoComponent, 11: CUP1-LINK-CMD-2=OK
NoProduct, NoComponent, 11: PLATE0-LINK-CMD=OK
NoProduct, NoComponent, 11: PLATE0-LINK-CMD-2=OK
NoProduct, NoComponent, 11: LEFT-ARM-CMD=PLACE
NoProduct, NoComponent, 11: RIGHT-ARM-CMD=PLACE
NoProduct, NoComponent, 11: ROBOT-CMD=NOCOMMAND
Plate1,    Table,       11: TABLE-CMD=SCAN

NoProduct, NoComponent, 12: CUP0-LINK-CMD=OK
NoProduct, NoComponent, 12: CUP1-LINK-CMD=OK
```

```
NoProduct, NoComponent, 12: CUP0-LINK-CMD-2=OK
NoProduct, NoComponent, 12: CUP1-LINK-CMD-2=OK
NoProduct, NoComponent, 12: PLATE0-LINK-CMD=OK
NoProduct, NoComponent, 12: PLATE0-LINK-CMD-2=OK
NoProduct, NoComponent, 12: PLATE1-LINK-CMD-2=OK
NoProduct, NoComponent, 12: LEFT-ARM-CMD=NOCOMMAND
NoProduct, NoComponent, 12: RIGHT-ARM-CMD=NOCOMMAND
NoProduct, NoComponent, 12: ROBOT-CMD=NOCOMMAND
NoProduct, NoComponent, 12: TABLE-CMD=NOCOMMAND

NoProduct, NoComponent, 13: CUP0-LINK-CMD=OK
NoProduct, NoComponent, 13: CUP1-LINK-CMD=OK
NoProduct, NoComponent, 13: PLATE0-LINK-CMD=OK
NoProduct, NoComponent, 13: PLATE1-LINK-CMD=OK
NoProduct, NoComponent, 13: CUP0-LINK-CMD-2=OK
NoProduct, NoComponent, 13: CUP1-LINK-CMD-2=OK
NoProduct, NoComponent, 13: PLATE0-LINK-CMD-2=OK
NoProduct, NoComponent, 13: PLATE1-LINK-CMD-2=OK
NoProduct, NoComponent, 13: LEFT-ARM-CMD=NOCOMMAND
NoProduct, NoComponent, 13: RIGHT-ARM-CMD=NOCOMMAND
NoProduct, NoComponent, 13: ROBOT-CMD=NOCOMMAND
NoProduct, NoComponent, 13: TABLE-CMD=NOCOMMAND
```

Listing B.4: Input plan $\mathcal{P}$ for the kitchen example in section 2.2.

```
RFID__0=NOSIGNAL
RFID__1=NOSIGNAL
RFID__2=NOSIGNAL
RFID__3=NOSIGNAL
```

Listing B.5: Observations for the kitchen example in section 2.2.

```
VARIABLE-DOMAIN-TYPE-DEFINITIONS
    VARIABLE-DOMAIN-TYPE COMPONENT-STATUS (OK FAULTY)
    VARIABLE-DOMAIN-TYPE ITEM-STATUS (OK LOST)
    VARIABLE-DOMAIN-TYPE TABLE-COMMAND (SCAN NOCOMMAND)
    VARIABLE-DOMAIN-TYPE ROBOT-COMMAND (MOVE NOCOMMAND)
    VARIABLE-DOMAIN-TYPE ROBOT-ARM-COMMAND (PICKUP PLACE NOCOMMAND)


    VARIABLE-DOMAIN-TYPE RFID-STATUS (NOSIGNAL RECEIVED)
    VARIABLE-DOMAIN-TYPE ARM-STATUS (OK FAULTY)

VARIABLE-DEFINITIONS
    VARIABLE PLATE0-LINK-CMD OF-TYPE CONTROL WITH-RANGE COMPONENT-STATUS
    VARIABLE PLATE0-LINK-CMD-2 OF-TYPE CONTROL WITH-RANGE COMPONENT-STATUS
    VARIABLE PLATE1-LINK-CMD OF-TYPE CONTROL WITH-RANGE COMPONENT-STATUS
    VARIABLE PLATE1-LINK-CMD-2 OF-TYPE CONTROL WITH-RANGE COMPONENT-STATUS
    VARIABLE CUP0-LINK-CMD OF-TYPE CONTROL WITH-RANGE COMPONENT-STATUS
    VARIABLE CUP0-LINK-CMD-2 OF-TYPE CONTROL WITH-RANGE COMPONENT-STATUS
    VARIABLE CUP1-LINK-CMD OF-TYPE CONTROL WITH-RANGE COMPONENT-STATUS
    VARIABLE CUP1-LINK-CMD-2 OF-TYPE CONTROL WITH-RANGE COMPONENT-STATUS

    VARIABLE PLATE0-LINK-ITEM OF-TYPE DEPENDENT WITH-RANGE ITEM-STATUS
    VARIABLE PLATE1-LINK-ITEM OF-TYPE DEPENDENT WITH-RANGE ITEM-STATUS
    VARIABLE CUP0-LINK-ITEM OF-TYPE DEPENDENT WITH-RANGE ITEM-STATUS
    VARIABLE CUP1-LINK-ITEM OF-TYPE DEPENDENT WITH-RANGE ITEM-STATUS

    VARIABLE TABLE-LINK-ITEM OF-TYPE DEPENDENT WITH-RANGE ITEM-STATUS
    VARIABLE TABLE-LINK-CMD OF-TYPE DEPENDENT WITH-RANGE COMPONENT-STATUS
    VARIABLE LEFT-ARM-LINK-CMD OF-TYPE DEPENDENT WITH-RANGE COMPONENT-STATUS
    VARIABLE RIGHT-ARM-LINK-CMD-2 OF-TYPE DEPENDENT WITH-RANGE COMPONENT-STATUS
    VARIABLE TABLE-CMD OF-TYPE CONTROL WITH-RANGE TABLE-COMMAND
    VARIABLE ROBOT-CMD OF-TYPE CONTROL WITH-RANGE ROBOT-COMMAND
    VARIABLE LEFT-ARM-CMD OF-TYPE CONTROL WITH-RANGE ROBOT-ARM-COMMAND
    VARIABLE RIGHT-ARM-CMD OF-TYPE CONTROL WITH-RANGE ROBOT-ARM-COMMAND
    VARIABLE RFID OF-TYPE OBSERVABLE WITH-RANGE RFID-STATUS
```

```


MODEL-DEFINITIONS
(root COMPOSITE
    (CHILDREN

        ((plate0-model COMPOSITE
            (CHILDREN
                (STARTING (plate0-model-ok PRIMITIVE (BEHAVIOR-CONSTRAINT (AND (OR (PLATE0-LINK-ITEM = OK)))) ))
                ((plate0-model-faulty PRIMITIVE (BEHAVIOR-CONSTRAINT (AND (OR (PLATE0-LINK-ITEM = LOST)))) ))
            )
            (TRANSITIONS
                (TRANS-FROM-TO-GUARD-PROB plate0-model-ok plate0-model-ok
                    (AND (PLATE0-LINK-CMD = OK) (PLATE0-LINK-CMD-2 = OK)) 1.0)
                (TRANS-FROM-TO-PROB plate0-model-faulty plate0-model-faulty 1.0)
                (TRANS-FROM-TO-GUARD-PROB plate0-model-ok plate0-model-faulty
                    (OR (PLATE0-LINK-CMD = FAULTY) (PLATE0-LINK-CMD-2 = FAULTY)) 1.0)
            )
        ))

        ((plate1-model COMPOSITE
            (CHILDREN
                (STARTING (plate1-model-ok PRIMITIVE (BEHAVIOR-CONSTRAINT (AND (OR (PLATE1-LINK-ITEM = OK)))) ))
                ((plate1-model-faulty PRIMITIVE (BEHAVIOR-CONSTRAINT (AND (OR (PLATE1-LINK-ITEM = LOST)))) ))
            )
            (TRANSITIONS
                (TRANS-FROM-TO-GUARD-PROB plate1-model-ok plate1-model-ok
                    (AND (PLATE1-LINK-CMD = OK) (PLATE1-LINK-CMD-2 = OK)) 1.0)
                (TRANS-FROM-TO-PROB plate1-model-faulty plate1-model-faulty 1.0)
                (TRANS-FROM-TO-GUARD-PROB plate1-model-ok plate1-model-faulty
                    (OR (PLATE1-LINK-CMD = FAULTY) (PLATE1-LINK-CMD-2 = FAULTY)) 1.0)
            )
        ))

        ((cup0-model COMPOSITE
            (CHILDREN
                (STARTING (cup0-model-ok PRIMITIVE (BEHAVIOR-CONSTRAINT (AND (OR (CUP0-LINK-ITEM = OK)))) ))
                ((cup0-model-faulty PRIMITIVE (BEHAVIOR-CONSTRAINT (AND (OR (CUP0-LINK-ITEM = LOST)))) ))
            )
            (TRANSITIONS
                (TRANS-FROM-TO-GUARD-PROB cup0-model-ok cup0-model-ok
                    (AND (CUP0-LINK-CMD = OK) (CUP0-LINK-CMD-2 = OK)) 1.0)
                (TRANS-FROM-TO-PROB cup0-model-faulty cup0-model-faulty 1.0)
                (TRANS-FROM-TO-GUARD-PROB cup0-model-ok cup0-model-faulty
                    (OR (CUP0-LINK-CMD = FAULTY) (CUP0-LINK-CMD-2 = FAULTY)) 1.0)
            )
        ))
        ((cup1-model COMPOSITE
            (CHILDREN
                (STARTING (cup1-model-ok PRIMITIVE (BEHAVIOR-CONSTRAINT (AND (OR (CUP1-LINK-ITEM = OK)))) ))
                ((cup1-model-faulty PRIMITIVE (BEHAVIOR-CONSTRAINT (AND (OR (CUP1-LINK-ITEM = LOST)))) ))
            )
            (TRANSITIONS
                (TRANS-FROM-TO-GUARD-PROB cup1-model-ok cup1-model-ok
                    (AND (CUP1-LINK-CMD = OK) (CUP1-LINK-CMD-2 = OK)) 1.0)
                (TRANS-FROM-TO-PROB cup1-model-faulty cup1-model-faulty 1.0)
                (TRANS-FROM-TO-GUARD-PROB cup1-model-ok cup1-model-faulty
                    (OR (CUP1-LINK-CMD = FAULTY) (CUP1-LINK-CMD-2 = FAULTY)) 1.0)
            )
        ))



        ((robot COMPOSITE

            (CHILDREN

                (STARTING (left-arm COMPOSITE
                    (CHILDREN
                        (STARTING (left-arm-ok PRIMITIVE (BEHAVIOR-CONSTRAINT (AND (OR (LEFT-ARM-LINK-CMD = OK)))) ))
                        ((left-arm-faulty PRIMITIVE (BEHAVIOR-CONSTRAINT (AND (OR (LEFT-ARM-LINK-CMD = FAULTY)))) ))
                    )

                    (TRANSITIONS
                        (TRANS-FROM-TO-PROB left-arm-ok left-arm-ok 0.999)
                        (TRANS-FROM-TO-PROB left-arm-faulty left-arm-faulty 1.0)

                        (TRANS-FROM-TO-PROB left-arm-ok left-arm-faulty 0.001)
                    )
                ))
                (STARTING (right-arm COMPOSITE
                    (CHILDREN
                        (STARTING (right-arm-ok PRIMITIVE (BEHAVIOR-CONSTRAINT (AND (OR (RIGHT-ARM-LINK-CMD-2 = OK)))) ))
                        ((right-arm-faulty PRIMITIVE (BEHAVIOR-CONSTRAINT (AND (OR (RIGHT-ARM-LINK-CMD-2 = FAULTY)))) ))
                    )

                    (TRANSITIONS
                        (TRANS-FROM-TO-PROB right-arm-ok right-arm-ok 0.99)
                        (TRANS-FROM-TO-PROB right-arm-faulty right-arm-faulty 1.0)

                        (TRANS-FROM-TO-PROB right-arm-ok right-arm-faulty 0.01)
                    )
                ))

            )

            (TRANSITIONS
            )
        ))


        ((table COMPOSITE

            (CHILDREN
                (STARTING (table-idle PRIMITIVE (BEHAVIOR-CONSTRAINT (AND (OR (RFID = NOSIGNAL))
                                                                          (OR (TABLE-LINK-CMD = OK))))
                ))
                ((table-detect PRIMITIVE (BEHAVIOR-CONSTRAINT
                                            (AND (OR (RFID = RECEIVED) (TABLE-LINK-ITEM = LOST))
                                                 (OR (RFID = NOSIGNAL) (TABLE-LINK-ITEM = OK))
                                                 (OR (TABLE-LINK-CMD = OK))
                                            )
                                         )
                ))
            )

            (TRANSITIONS

                (TRANS-FROM-TO-GUARD-PROB table-idle table-idle (AND (TABLE-CMD = NOCOMMAND)) 1.0)
                (TRANS-FROM-TO-GUARD-PROB table-idle table-detect (AND (TABLE-CMD = SCAN)) 1.0)
                (TRANS-FROM-TO-GUARD-PROB table-detect table-detect (AND (TABLE-CMD = SCAN)) 1.0)
                (TRANS-FROM-TO-GUARD-PROB table-detect table-idle (AND (TABLE-CMD = NOCOMMAND)) 1.0)

            )

        ))

    )

    (TRANSITIONS
    )
)
```

Listing B.6: PHCA description code for the kitchen example in section 2.2.

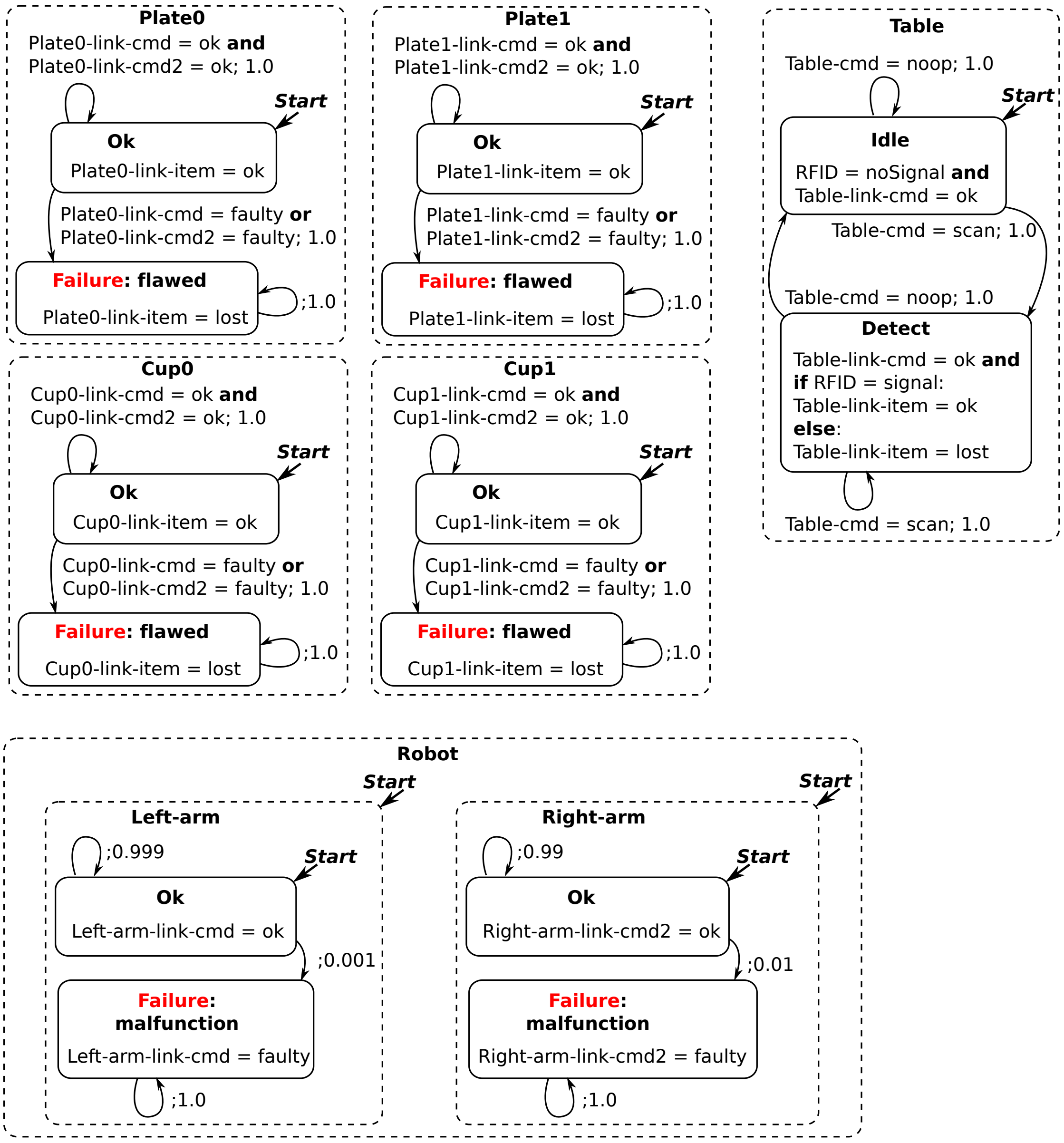


Plate0
Plate0-link-cmd = ok and Plate0-link-cmd2 = ok; 1.0
Start
Ok
Plate0-link-item = ok
Plate0-link-cmd = faulty or Plate0-link-cmd2 = faulty; 1.0
Failure: flawed
Plate0-link-item = lost
;1.0
Plate1
Plate1-link-cmd = ok and Plate1-link-cmd2 = ok; 1.0
Start
Ok
Plate1-link-item = ok
Plate1-link-cmd = faulty or Plate1-link-cmd2 = faulty; 1.0
Failure: flawed
Plate1-link-item = lost
;1.0
Table
Table-cmd = noop; 1.0
Start
Idle
RFID = noSignal and Table-link-cmd = ok
Table-cmd = scan; 1.0
Table-cmd = noop; 1.0
Detect
Table-link-cmd = ok and if RFID = signal: Table-link-item = ok else: Table-link-item = lost
Table-cmd = scan; 1.0
Cup0
Cup0-link-cmd = ok and Cup0-link-cmd2 = ok; 1.0
Start
Ok
Cup0-link-item = ok
Cup0-link-cmd = faulty or Cup0-link-cmd2 = faulty; 1.0
Failure: flawed
Cup0-link-item = lost
;1.0
Cup1
Cup1-link-cmd = ok and Cup1-link-cmd2 = ok; 1.0
Start
Ok
Cup1-link-item = ok
Cup1-link-cmd = faulty or Cup1-link-cmd2 = faulty; 1.0
Failure: flawed
Cup1-link-item = lost
;1.0
Robot
Start
Left-arm
;0.999
Start
Ok
Left-arm-link-cmd = ok
;0.001
Failure: malfunction
Left-arm-link-cmd = faulty
;1.0
Start
Right-arm
;0.99
Start
Ok
Right-arm-link-cmd2 = ok
;0.01
Failure: malfunction
Right-arm-link-cmd2 = faulty
;1.0

Figure B.3.: PHCA model for the kitchen example in section 2.2.

Nomenclature

$(s^{t_i}_{\Pi_U}, m^{t_i})$	Set of so-called discrete flow constraints. A discrete flow constraint in $\mathcal{F}_d$ is the discrete abstraction of a corresponding flow constraint (i.e. differential equation) in $\mathcal{F}$.
(s^{t}_{U}, m^{t})	HyPHCA hybrid state.
A_{df}	Discrete flow PHCA model.
$\dot{U}_{lvl}$	Derivative of the fill level of the filling station's silo. I.e. this encodes the change rate per time unit.
$\mathcal{E}_{\mathcal{P}}$	Execution adaptation function. Simulates the execution of a PHCA model.
G_i	A goal a technical system should achieve. In manufacturing plants, the goal to finish a certain product, indexed by i.
$\langle(p, c, t, a)\rangle_j$	Sequence of tuples forming a plan, in manufacturing a schedule. p identifies a product to be worked by a station or factory component c, which should perform action a at time t.
$\mathcal{C}$	PHCA set of finite domain constraints. Consists of behavior constraints of locations and guard constraints of tranistions.
$\mathcal{D}$	Set of definitions for a Bayesian logic network.
$\mathcal{F}$	In the context of Bayesian logic networks and probabilistic reasoning: Set of fragments for a Bayesian logic network. In the context of hybrid models and HyPHCA: Set of continuous flow constraints, i.e. constraints over real-valued variables in the form of linear ordinary differential equations.
$\mathcal{F}_d$	Set of discrete flow constraints. A discrete flow constraint in $\mathcal{F}_d$ is the discrete abstraction of a corresponding flow constraint (i.e. differential equation) in $\mathcal{F}$.
$\mathcal{L}$	Set of first-order logical formulas for a Bayesian logic network.

$\mathcal{R} = (X, D, C)$	A constraint net, consisting of a set of variables, a set of domains and a set of constraint functions.
$\mathcal{R}_B = (X, D, G, P)$	A Bayesian net, consisting of a set of variables, a set of domains for these variables, a graph representing conditional dependencies and a set of conditional probability functions.
$\mathcal{T}$	PHCA set of transitions.
HA	Hybrid PHCA model.
$St(\mathrm{M})$	Set of system behaviors/trajectories, i.e. sequences of system states (time frame not specified).
$St^t(\mathrm{M})$	Set of states for system model M at time t.
fR	The constant fill rate of the filling station's silo.
Cmd	PHCA command variables.
Dep	PHCA dependent variables.
ω_{fail}	Lower threshold for success probability, if it is below ω_{fail}, something must be done, e.g. re-planning.
ω_{success}	Upper threshold for success probability, if it is above ω_{fail}, success is assumed to not be jeopardized.
$\mathcal{B} = (\mathcal{D}, \mathcal{F}, \mathcal{L})$	A Bayesian logic network consisting of a set of definitions, a set of fragments and a set of first-order logical formulas.
$\mathcal{P}$	Sequence of operation steps, a plan, a technical system shall execute to achieve some goals.
M_{PHCA}	PHCA model.
$M_{\text{PHCA}}^{\mathcal{P}}$	PHCA model adapted to the execution of a plan $\mathcal{P}$. The model is the result of execution function $\mathcal{E}_{\mathcal{P}}$.
M_{PHCA}^{N}	PHCA model that explicitly represents N time steps, i.e. where locations, transitions, etc. are represented explicitly for all the N time steps.
Π	PHCA set of finite variables, if not math. product.

Π^t	Set of PHCA variables at time t.
$\Pi_{U'}$	Discrete versions of the real-valued HyPHCA variables U', which are created when translating a HyPHCA to a discrete flow PHCA.
Π_U	Discrete versions of the real-valued HyPHCA variables U, which are created when translating a HyPHCA to a discrete flow PHCA.
Σ	PHCA locations.
Σ_c	PHCA composite locations.
Σ_p	PHCA primitive locations.
$Pr(\mathcal{G}_i \mid \mathbf{o}^{0:t})$	Success probability for product indexed by i and given observations $\mathbf{o}^{0:t}$.
$\theta(t)$, m^t	Marking at time point t, part of trajectory θ.
Θ	Alternate notation for the set of all trajectories of a PHCA.
Υ	Function that translates a PHCA model into some generic problem representation, e.g. a constraint net or a Bayesian logic net.
Υ_{BLN}	Function that translates a PHCA to a Bayesian logic network.
Υ_{COP}	Function that translates a PHCA model into a constraint net.
C	Set of constraint functions.
D	Set of domains for model variables.
$D_{X_\Sigma}, D_\Pi, D_{X_{\text{Exec}}}$	Sets of domains for the constraint net variables encoding PHCA locations X_Σ and PHCA variables Π, respectively, and the set of domains for constraint net variables X_{Exec} used to encode the structure and transition semantics of PHCA.
G_λ	A grid cell, i.e. a rectangular part of the state space of a HyPHCA.
k	Number of system behaviors/trajectories to compute. More general, number of solutions to enumerate in decreasing order with respect to their objective value.
N	Number of time steps considered simultaneously for plan assessment, i.e. the size of the time horizon spanning past and future.
O	PHCA observation variables.

P	Set of conditional probability functions.
p_l	Lower bound on success probability.
p_u	Upper bound on success probability.
$P_\Xi(\Xi_i)$	PHCA probability distribution over intial markings Ξ_i.
p_l^*	Stochastic lower bound of success probability.
$P_T[l_i]$	PHCA prob. dist. over transition functions T.
p_u^*	Stochastic upper bound of success probability.
U	Set of all HyPHCA real-valued variables.
U'	Set of all HyPHCA real-valued variables right after a transition was taken. That is, if some variable U_i has value a, and taking a transition changes that value to b, then variable U_i' takes value b.
U_i	Continuous, e.g. real-valued, variable.
U_{lvl}	The real-valued fill level of the filling station's silo.
X	Set of model variables. In context of COP: set of COP variables. In context of BLN: set grounded BN random variables.
X^t	Set of system variables (COP variables, grounded BN random variables) that encode the model state at time t.
X^t_{Exec}	Set of auxiliary variables used to encode PHCA structure and its transition semantic over discrete time steps.
X^t_Σ	Set of constraint net variables that correspond to PHCA locations.
X_{U_i}	Discrete-valued variable, in a discrete flow PHCA the discrete abstraction of variable U_i of the corresponding HyPHCA.
G_i	Set of all goals for some plan $\mathcal{P}$.
em	Example model, the model used for the running example introduced in chapter 2. Also used for evaluation.
fm$x$	Factory model 1, 2 or 3, used for the evaluation.
sm	Satellite model.

List of Abbreviations

General Abbreviations

AI	Artificial intelligence.
BLN	Bayesian logic network.
BN	Bayesian network.
BOM	Bill of materials.
CAD	Computer aided design.
CAPP	Computer aided process planning.
CDF	Cumulative distribution function.
CIM	Computer-integrated manufacturing.
COP	Constraint optimization problem.
CPT	Conditional probability table.
CoTeSys	Cognition for technical systems.
CRAM	Cognitive robot abstract machine.
DLR	Deutsches Zentrum für Luft- und Raumfahrt.
dfPHCA	Discrete flow PHCA.
HMM	Hidden Markov model.
HyPHCA	Hybrid PHCA.
k-N-TWF	Time window filtering with k most probable trajectories and N time steps.
MAP	Most probable a posteriori hypothesis.

MN	Mixed network.
MPE	Most probable explanation. Also known as MLE, most likely explanation.
PCOP	Probabilistic constraint optimization problem. In this work, most of the time a COP is a PCOP, therefore we use COP and PCOP interchangeably.
PDDL	Planning domain description language.
PHCA	Probabilistic hierarchical constraint automaton/automata.
PLC	Programmable logic controller.
POMDP	Partially observable Markov decision processes.
RFID	Radio-frequency identification.
RMPL	Reactive model-based programming language.
STRIPS	Stanford Research Institute Planning System.
TUM	Technische Universität München.
VSCP	Valued constraint satisfaction problem.
WCSP	Weighted constraint satisfaction problem.

Abbreviations in Bibliography

AAAI	The association for the advancement of artificial intelligence. Also used for the associated conference.
AAMAS	International conference on autonomous agents and multiagent[sic] systems.
ACM	Association for computing machinery.
ASWEC	Australian software engineering conference.
ECAI	European conference on artificial intelligence.
ETFA	IEEE international conference on emerging technologies and factory automation.
FMSB	Formal methods in systems biology.
HSCC	International conference on hybrid systems: computation and control.
ICAPS	International conference on automated planning and scheduling.
ICCA	IEEE international conference on control and automation.
ICRA	IEEE international conference on robotics and automation.
ICSE	International conference on software engineering.
IJCAI	International joint conference on artificial intelligence.
ISRR	The international symposium of robotics research.
IV	IEEE intelligent vehicles symposium.
KI	Annual conference on artificial intelligence in Germany. KI stands for German "Künstliche Intelligenz".
LNCS	Lecture notes in computer science. Published by Springer.
SAC	ACM symposium on applied computing.
SFM	International school on formal methods for the design of computer, communication and software systems.
UAI	Conference on uncertainty in artificial intelligence.