Distributed Mode Estimation Through Constraint Decomposition

by

Henri Badaro

Submitted to the Department of Aeronautics and Astronautics
in partial fulfillment of the requirements for the degree of

Master of Science in Aeronautics and Astronautics

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2010

Author .....
Department of Aeronautics and Astronautics
August 23, 2010

Certified by .....
Brian C. Williams
Professor
Thesis Supervisor

Accepted by .....
Eytan H. Modiano
Associate Professor of Aeronautics and Astronautics
Chair, Committee on Graduate Students

Distributed Mode Estimation
Through Constraint Decomposition

by

Henri Badaro

Submitted to the Department of Aeronautics and Astronautics
on August 23, 2010, in partial fulfillment of the
requirements for the degree of
Master of Science in Aeronautics and Astronautics

Abstract

Large-scale autonomous systems such as modern ships or spacecrafts require reliable monitoring capabilities. One of the main challenges in large-scale system monitoring is the difficulty of reliably and efficiently troubleshooting component failure and deviant behavior. Diagnosing large-scale systems is difficult because of the fast increase in combinatorial complexity. Hence, efficient problem encoding and knowledge propagation between time steps is crucial. Moreover, concentrating all the diagnosis processing power in one machine is risky, as it creates a potential critical failure point. Therefore, we want to distribute the online estimation procedure. We introduce here a model-based method that performs robust, online mode estimation of complex, hardware or software systems in a distributed manner. Prior work introduced the concept of probabilistic hierarchical constraint automata (PHCA) to compactly model both complex software and hardware behavior. Our method, inspired by this previous work, translates the PHCA model to a constraint representation. This approach handles a more precise initial state description, scales to larger systems, and to allow online belief state updates. Additionally, a tree-clustering of the dual constraint graph associated with the multi-step trellis diagram representation of the system makes the search distributable. Our search algorithm enumerates the optimal solutions of a hard-constraint satisfaction problem in a best-first order by passing local constraints and conflicts between neighbor sub-problems of the decomposed global problem. The solutions computed online determine the most likely trajectories in the state space of a system. Unlike prior work on distributed constraint solving, we use optimal hard constraint satisfaction problems to increase encoding compactness. We present and demonstrate this approach on a simple example and an electric power-distribution plant model taken from a naval research project involving a large number of modules. We measure the overhead caused by distributing mode estimation and analyze the practicality of our approach.

Thesis Supervisor: Brian C. Williams
Title: Professor

Acknowledgments

I would like to express my gratitude to the people who supported and helped me through to the completion of this thesis.

I am grateful to my advisor and thesis supervisor, Brian Williams, for his key insights and encouragements on my research, his critical help on technical writing, and for his efforts to make the thesis deadline.

I would like to thank and congratulate my parents, my brother, and the rest of my family, whose constant love, support, and friendship allowed me to achieve what I have achieved so far, including this Master's thesis.

I would also like to thank all my friends and labmates who helped me during the research, writing, completion, and printing phases of this thesis, in particular: Shannon, Alborz, Andreas, Cristi, David, Hiro, Hui, Julie, Patrick, Ted, Yi. Thank you to Alison for the logistics, Dave, Mike and Paul for the the meetings, Bobby, Larry, Seung and Stephanie for their company in the lab.

I would finally like to thank my friends from MIT and the ones whom I knew before, for the endless conversations, for the off-campus moments, the great fun, and the support. They made my stay here so valuable and unforgettable, in the dorms, Cambridge, and remotely from other places and time zones. We will stay in touch by way of aeronautics, TCP/IP or UDP.

This research was sponsored by a grant from the Office of Naval Research through the Johns Hopkins University Applied Physics Laboratory, contract number N00014-08-C-0681.

Contents

List of Figures

Definitions and Examples

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

Large-scale autonomous systems such as modern ships and spacecraft require reliable monitoring capabilities. Unmanned vehicles are not the only devices to employ autonomy. As air, sea, and ground vehicles become more complex, they also require a high level of autonomy to assist the users, pilots, or operators.

For instance, in a large modern ship, complex systems ensure the proper distribution of water, power, heat, air, and information. There may be several reasons to limit the amount of human intervention in these systems. Humans, particularly when under pressure, are error-prone. Moreover, some critical zones of a vehicle may not be accessible or can be dangerous due to steam, high voltage, and water leaks. Therefore, robust autonomy capabilities are crucial to increase vehicle survivability where human intervention is not possible or responsive enough.

Autonomous embedded systems, such as distribution systems in intelligent vehicles, may fail in various predictable or unexpected ways. Therefore, a robust autonomy capability requires reliable monitoring methods. Moreover, if relevant diagnosis and troubleshooting can be performed, more complex and capable systems can be used while maintaining the same level of survivability.

System diagnosis consists of identifying sets of working and faulty components, and their probability. An autonomous diagnosis engine analyzes commands and obser-

vations of the system in order to output diagnoses over time. Autonomous diagnosis allows to detect problems without human intervention, hence increasing safety.

Yet, if the diagnosis engine fails, the whole mission is in jeopardy. Therefore, centralizing the diagnosis puts the system at risk: by concentrating all the computing and decision making into one place, a critical failure point is created. A way to solve this problem is to distribute the diagnosis process. Furthermore, if this process is distributed across several machines, one can still duplicate each of them. If each cluster (computer, or processor) is redundant, the whole system fails only when two paired clusters fail, whereas if a centralized (monolithic) system is duplicated, two failures are sufficient to endanger the computational process.

1.2 Challenges and Required Capabilities

Autonomous monitoring of embedded systems, such as the ones previously mentioned, raises a number of challenges, which can be addressed by extending the current state of the art in distributed diagnosis [15, 19, 13, 36, 21, 14].

The first challenge comes from the complexity of the system state space, in particular if a system consists of both software and hardware components. The state space representing the possible nominal and unknown behaviors grows much faster than the number of components. Therefore, a monitoring process may not be able to enumerate exhaustively all possible diagnoses in a reasonable amount of time. For these complex systems, an efficient problem encoding may be a crucial step in making the search of component failures tractable.

The second major challenge concerns the dynamic behavior of the systems we are monitoring. These embedded systems evolve over time, and the components may be in various states across the period of observation. However, starting a new estimation process at every time step without keeping track of previous intermediate results is both time consuming and inefficient. A system state is estimated from a history of previous observations. Therefore, in the case of long experiments, it is interesting to update the belief state while keeping as many previous estimates and as much

knowledge of the system as possible.

Finally, a third important challenge has to do with the distribution of the monitoring process. If we want to solve the diagnosis problem online in a distributed way, we need to decompose it and decentralize it onto several computing clusters. Therefore, we need a decomposable problem framing, paired with a distributed optimal solver. One of the main difficulties of performing a robust diagnosis lies in the fact that we do not only look for a single most likely diagnosis, (as this would not be robust to diagnosis error). We must compute a whole set of optimal solutions in a distributed way, while limiting the amount of communication between the clusters.

Distribution is an important step for improving monitoring reliability. The additional robustness associated with distribution can be interpreted in several ways. First, in the case where a computation node is lost, a distributed and decentralized diagnosis algorithm may still be able to work in a degraded mode, providing most likely diagnoses to the best of the knowledge of the remaining cluster. This is indeed the case for our approach: it can still provide diagnoses even if some computers or some observations are not available. Yet, we do not deal with cases where failing means of communication lead to corrupted message transmission.

Second, if each computer is duplicated in a distributed monitoring process, the only way to make the whole computation fail is to lose two identical nodes. The loss of two non-paired nodes does not affect the monitoring ability. On the other hand, if a large computer centralizing all monitoring is duplicated, the loss of two machines destroys all the monitoring capability. Therefore, in order to reach the same level of tolerance to computer failure as in a distributed process, it may be necessary to add more than one large backup computer in the centralized version. Hence, if the online estimation process is distributed onto several processors, then the same level of robustness to processor or communication failure requires less redundancy.

Finally, with a suitable encoding, estimation may be dispatched locally and gathered to determine the overall system state. This permits the diagnosis of embedded systems where several subsystems are integrated from different manufacturers, in the sense that this distributed algorithm could be adapted to perform several partial local

mode estimations specific to each component in order to achieve global monitoring.

1.3 Approach and Innovations

In order to provide the three main required capabilities introduced in Section 1.2, we propose a method to estimate in a distributed manner the system state evolution over a certain time window. We also propose a way to update this belief of the system state when new observations are made. Hence, we introduce a capability to monitor a mixed hardware and software large-scale system over long time periods, by distributing the required processing power on a set of computers.

For that purpose, we improve and extend a model-based approach to diagnosis that uses Probabilistic Hierarchical Constraint Automata (PHCA) modeling to perform mode estimation over a window of N time steps [27]. This method handles multiple faults, novel faults, intermittent faults, and delayed symptoms. We increase the scalability of this method by encoding the estimation problem as an Optimal Constraint Satisfaction Problem (OCSP) containing only hard constraints and costs on some single variable assignments [42]. An increased scalability allows us to solve larger problems and hence deal with complex systems with mode components.

Every time step, new observations of the system are made, and new commands are sent. The state of the system evolves and its diagnosis must be updated accordingly. If diagnoses are performed over a moving, finite, time window of operations, a belief state update consists of an update of the estimated state trajectories of the system. In order to allow an efficient belief update at each new time window, we introduce a dummy decision variable into this OCSP. This variable summarizes the previous most-likely finite horizon diagnoses along with their probabilities. Depending on the new observations, some of these previous hypotheses will be extended to produce updated estimations on the most likely system state trajectories.

Most importantly, the distribution of the mode estimation process is ensured by a distributed optimal solver that enumerates solutions to the aforementioned OCSP in a best-first order, hence allowing us to compute the K most likely diagnoses at

each time step. This method uses the optimal solver OPSAT in each cluster [42], and the clusters work on a tree-decomposed version of the dual constraint graph of the OCSP. Clusters communicate by passing messages informing their neighbors of their partial solution candidates or by sending conflicts to signal dead-ends in the search. This algorithm is inspired by a previous message passing method that uses only soft-constraints [29], but our method focuses on increased scalability by using hard constraints and by introducing conflict handling.

To summarize, our approach has three sources of innovation:

1. distributed optimal hard-constraint problem solving,
2. efficiency of a problem encoding, and
3. the ability to update the belief state of a system online.

This approach will be presented on simple examples and on a realistic power distribution system example detailed in Chapter 3. This latter large-scale model contains many components that can fail in unexpected ways. It is taken from a larger naval project, where robust autonomy is crucial for survivability, and to reliably add additional functionality to embedded systems.

1.4 Outline

The rest of this thesis is organized as follows.

In Chapter 2, we first describe relevant prior work in the fields of distributed diagnosis, distributed optimal constraint solving, and model-based estimation. We also discuss how this work tackled some of the challenges mentioned in this introduction.

We then present a problem statement and a motivating large-scale power distribution system example in Chapter 3. This example is a real-world scenario that justifies the choice of methods and the orientation of this research to achieve distributable, robust mode estimation.

The subsequent three chapters explain our approach, innovations, and technical solutions. Our approach to distributed model-based diagnosis consists of two phases.

First, a model of the system is used to frame the diagnosis problem into a standard optimal constraint satisfaction problem. This problem is decomposed with an off-the-shelf tree-decomposition method, in order to produce a tree of connected subproblems. Then, the second phase performs online diagnosis by solving the tree of subproblems in a distributed manner. The distributed search of solutions outputs a set of most-likely state trajectories depending on observations and commands. The distributed search over the coarse-grain decomposed problem consists of sequences of local optimal solution enumerations, and communications of these solutions between neighbor subproblems by message-passing. Additionally, the state estimates may be updated at each time step. Empirical results show that model-based diagnosis can be performed on large-scale systems and that our diagnosis engine can detect unpredicted failures. Moreover, the overhead on processing time of a distributed search against a centralized search is small, even if it is measured on a simulated distributed optimal constraint solver.

Chapter 4 proposes a novel distributed search method based on tree-decomposition of a dual hard-constraint graph. Chapter 5 deals with an improved model and problem encoding of probabilistic hierarchical constraint automata, which allows us to update the mode estimates of a system more efficiently than in prior work. Next, Chapter 6 introduces implementation insights and empirical results for the distributed mode estimation method.

Finally, Chapter 7 concludes this thesis, and presents possible areas of future work.

Chapter 2

Related Work

In Section 1.2 we introduced three challenges that arise from autonomous, distributed diagnosis of large-scale embedded systems, and we presented three features that a diagnosis executive could deliver in order to tackle these three challenges.

First, the executive should run on a decentralized set of computers, without jamming the network with excessive intermediate result communication. In Section 2.1, we describe related work in distributed diagnosis and distributed optimal constraint solving. The objective of this distribution is to avoid centralizing all the computing power at one single point.

Second, we want to diagnose mixed software and hardware embedded systems that may disclose unexpected behavior. In Section 2.2 we describe methods often used to represent models of embedded systems and to encode the diagnosis problem, in the field of model-based diagnosis. These methods mainly use probabilistic and hierarchical automata in order to model finite-domain systems with a discrete stochastic state evolution.

Finally, mode estimation should be able to keep as much information as possible, while updating the mode estimates at each new discrete time step. In Section 2.3, we discuss previous methods used to propagate knowledge of the system behavior and previous estimates over time when new observations are made and new commands are sent. As the number of possible state evolutions is very large, monitoring executives are often not able to keep track of the whole evolution history, therefore an improved,

more efficient problem framing is needed.

2.1 Distributed Diagnosis and Constraint Solving

In this section, we present work related to distributed diagnosis. Many diagnostic methods rely on a constrained optimization process. Hence, distributed diagnosis can be achieved by decomposing the online problem solving. This is why we also recall techniques commonly used to decompose and solve general constrained optimization problems. Finally, we focus on more specialized methods for dealing with distributed discrete optimal constraint satisfaction.

2.1.1 Previous Work in Distributed Diagnosis

One of the most direct ways to perform distributed diagnosis is to use and modularity of the problem framing. To perform decomposition, this kind of approach can be found in expert diagnostic systems [19]. For the systems we care about, expert system diagnosis offers a less powerful diagnostic capability than the more modern model-based approach. However, by nature, expert diagnosis can perform large-scale simple distributed diagnoses with a small overhead as compared to a centralized version.

Expert diagnosis uses knowledge about the system to efficiently associate observations to diagnoses [19]. This knowledge can be based on a series of simulations or benchmarks. Additionally, humans can provide the system with knowledge about the different failure scenarios. Case-based and rule-based reasoning systems are particularly useful solutions when a model of the system is not known. Moreover, rule-based systems are capable of exploiting a large amount of parallelism, due to the nature of the constrained problems involved [15]. These monitoring systems are based on lists of simple implication rules for checking whether the behavior of components is consistent with predefined descriptions, and so the truth of each of these implications can be verified independently on different machines.

Simple rule systems are standard paradigms for space systems. When scaling on large systems, the rule sets may become too large to maintain or to debug. Moreover,

powerful as it may be, a knowledge-based approach does not deal with unexpected or unpredictable behaviors of system components. Hence, in cases where a complete expertise is required, model-based diagnosis has been proposed in order to increase robustness to unexpected behavior. Model-based diagnosis uses a model of the system and the environment along with observations to determine if the estimated behavior is consistent with a *nominal* or an *unknown* situation.

In spite of the practicality of rule-based diagnosis when it comes to distributed computation, it may not apply for our usage, as we want to determine the probabilities of each diagnosis.

Most previous distributed model-based diagnosis methods were conducted in the field of multi-agent systems.

Fabre *et al.* [13] present a distributed diagnosis method for large, discrete-event, dynamic systems. The motivating idea of this paper is to deal with systems that are too large to be handled as a whole. The method proposes that local supervisors provide partial local monitoring, in a framework related to Markov fields theory and Bayesian networks. A global system is *factored* into smaller subsystems that communicate through common variables. Then a monitoring executive fully uses concurrency properties to identify independent behaviors, so as to limit the problem complexity. Additionally, this method is fully fast, decentralized, and it is robust to message passing latency. This method however cannot output a set of most-likely state trajectories, it stops when one consensus on a globally-consistent diagnosis is reached between all the neighbors. In order to provide robustness to our system, we cannot limit ourselves to one single diagnosis per time step.

Roos *et al.* [36] analyze the influence of spatial and semantical knowledge distribution while performing multi-agent diagnosis. Multiple agents, each having their own local constraints, interact with each other, creating other interaction constraints. Hence, Ross *et al.* conclude that the sole knowledge of information in partial problems is not sufficient to decide what a global situation looks like. As we also notice in this thesis, when solving decomposed optimal constraint satisfaction problems, that the way knowledge and interactions are distributed among components has a substantial

influence on the difficulty to reach consensus between subproblems. Therefore, as pointed out by Roos *et al.* [36], though theoretically possible, multi-agent or distributed diagnosis are not necessarily always practically feasible.

Because of the aforementioned spatial knowledge distribution issues, other methods have analyzed the influence of the topology of a distributed system on the distributed diagnosis paradigms. This insight is particularly valuable in networked embedded systems In these systems, the topology of the system network determines the shape of a diagnosis engine. Hence, Kurien, Koutsoukos and Zhao in [21] use model-based diagnosis along with network topology analysis to perform consistency-based diagnosis. This method produces distributed constraint satisfaction problems shaped by the network topology. Furthermore, this diagnosis approach is fast. Yet, it does not take into account the probability of component failure. Instead, it returns diagnoses that are consistent with the observations. Hence, it may overestimate the likelihood of a multiple failure for instance, if this multiple failure explains a deviant device behavior.

Finally, another active field of research is based on Petri net representation, such as in [14] by Genc and Lafortune. Petri nets can model the possible evolution of discrete-event systems, and they are backed up with a theoretical mathematical model that allows many computational optimizations. In [14], this abstraction method is used to perform online model-based fault diagnosis. In this method, one Petri net is decomposed into two entangled nets, which lead to a distributed monitoring processing. The theory behind Petri nets remains valid and can still be used in each of the sub-nets after the decomposition; therefore, the problem decomposition and distribution is computationally efficient. This method assumes that the Petri nets models of the system can be decomposed into two place-bordered Petri nets satisfying certain conditions, so that the two resulting diagnosers can exchange messages about the occurrence of observable events.

2.1.2 General Distributed Constraint Optimization Methods

Many model-based diagnostics processes can be framed as constraint optimization problems [10, 39, 40]. A probabilistic model of the software and hardware components of the system along with observations can be used to monitor the system behavior by computing the most likely states. These are determined from the optimal feasible solutions to the estimation problem.

We assume in this work that the variable domains are discrete, as well as the divisions of time into synchronous time steps. This influences the problem representation and the solution method. Many systems, such as digital electric circuits or sequential mechanical devices, operate over a finite number of discrete modes, and thus can be well represented by a discrete model. Moreover, the behavior of sequential devices whose components which modes evolve at similar synchronous speeds can be adequately captured with discrete time steps. However, very non-linear dynamical systems, such as heat machines or turbulent fluid circuits, are unlikely to be modeled with discretized steps.

Our problem may be framed as a standard mixed integer-linear program [5] that can be distributed with a method like Dantzig-Wolfe decomposition [8] (Column Generation) adapted to MILPs. Although a MILP framing is not the most convenient for our purposes, it is worth mentioning, as some notions in the decomposition method can be found in later more specialized approaches, such as master problems (objective function to optimize), coupling constraints, projections, or cutting planes (which are a kind of conflicts).

A more specialized and adapted framework is mixed linear-logic programming. This kind of problems can be solved using Bender's decomposition [3, 18] (row generation), which shares many similarities with the Dantzig-Wolfe method, including the generation of candidate solutions and cutting planes. More recently, Jégou, Terrioux and Pinto [20, 32] have presented computationally efficient and practical distribution algorithms for optimal constraint satisfaction problems, adapted to fast, distributed diagnosis. However all these methods directly allow to find a single solution.

2.1.3 Distributed Optimal Constraint Satisfaction

In general, constraint optimization problems (with soft constraints) can be solved by distributed constraint optimization methods (DCOP). Many available algorithms have been optimized for various particular problems. For instance the multiple knapsack problem or the distributed graph coloring problems can indeed be solved efficiently by DCOP algorithms, but the generic DCOP algorithms have higher average complexity than their specialized counterparts. They are either exponential in memory space (Distributed Pseudotree Optimization Procedure in [31]) or exponential in the number of messages exchanged between agents (Asynchronous Backtracking in [30], Asynchronous Partial Overlay in [22], No-Commitment Branch and Bound in [6]).

If a large amount of memory is available, DPOP may look promising in our case, as it bounds the need for communication between agents, and it can solve OCSP problems efficiently. However it only allows to find the optimal solution of a problem. It is indeed possible to find the next optimal solutions by negating all the previous assignments inside new constraints. However this kind of constraint involving all the variables would affect the efficiency of the decomposition by creating a fully connected pseudo-tree. Moreover, constraints are expressed in terms of a set of possible partial assignments. Thus a negative constraint is transcribed as the set of all the other possible partial assignments in the scope of the negation. This enumeration leads to an exponential number of possibilities, and if all the variables are involved, the constraint description is too large to be represented.

We chose a method similar to the one described by Sachenbacher and Williams in [37] that can compute a set of optimal solutions to an optimal soft-constraint satisfaction problem (valued CSP). In this thesis, we use hard constraints instead of soft constraints because our online mode estimate update method is based on a hard-constraint encoding. Additionally, we inspired from this previous method and increased its scalability on larger problems in order to adapt it to our needs, by using a coarse-grain problem decomposition.

Our approach applies a tree-clustering method [11] to the dual graph of hard constraints in order to produce a tree of partially correlated sub-problems. We then use a conflict-based best-first enumeration of the solution to compute globally consistent trajectories in the transition space by passing messages and conflicts between neighbor sub-problems. It is a more specialized technique than the ones in the previous subsection to solve discrete problems. And as we restrict our encoding to hard constraints, we manage to increase the efficiency of our method compared to the one by Sachenbacher and Williams [37].

2.2 Model Representation and Problem Encoding

In a physical system, some faulty component behaviors may not be immediately visible when failure occurs. For instance, it is not possible to distinguish a relay that is *stuck-closed* from a properly working *closed* relay until it is sent a command to toggle open. Hence, symptoms of a failure may arise some time after the failure actually occurs. This is why we need to estimate the state of a system over a history of observations (finite time window), as explained later in this section.

Moreover, we need to be able to accurately represent the possible evolutions of the components of the system. Thus, we need a diagnosis problem formulation that takes into account the probabilistic (stochastic) behavior of a system. We want to take into account the probability of failure of each component because it is not computationally possible to compute all the possible diagnoses at every time step. Hence, we choose to enumerate the most-likely diagnoses, maximizing the probability that the accurate diagnosis is in the set of enumerated solutions. That is why we need to take probabilities into account in our model representation.

In the model-based monitoring framework, several automaton-based representations have been used to model a system. When dealing with dynamic systems, these models are usually encoded into intermediate representations such as a trellis diagram.

Probabilistic Concurrent Constraint Automata (PCCA) were first introduced in the Livingstone and Titan mode estimation frameworks [41, 40]. They are used to rep-

resent plant models by modeling each component as a concurrent automaton. These components can interact with each other and their behaviors is not deterministic. PCCA could not model software behavior.

In order to offer a more expressive modeling than PCCA, previous work introduced a mode estimation method framing a compact Probabilistic, Hierarchical, Constraint Automaton (PHCA) representation into a soft constraint optimization problem [28]. However it did not allow an efficient use of complex transitions or the definition of an initial state, both desirable to achieve a much more accurate modeling.

Additionally, this method introduced a finite-time-horizon mode estimation. N-step time window problem encoding introduced in this previous work is a necessary feature that we are keeping in our approach. Finite time window mode estimation diagnoses systems by keeping track of observations and commands over a finite history of time steps.

In this thesis, we effectively propose a method for an efficient, PHCA-based encoding of complex transitions that mix probability distributions with constraint guards. We also propose a way to deal with initial state representation. Instead of specifying a single possible initial state as in the previous work, we can specify a probability distribution over a set of initial states. Each initial state corresponds to a certain set of modes for each component of the system. Hence, we intend to represent as precisely as possible a probabilistic evolution of a model while remaining tractable.

In the method presented in this thesis, the system representation as a probabilistic hierarchical constraint automaton handles complex transitions and specifies an initial probability distribution over the possible system states. This encoding is then converted into a more compact multi-step trellis diagram, then framed as an optimal constraint satisfaction problem. These notions will be more formally defined in later chapters. In short, the trellis diagram represents all the possible sequences of transitions between feasible states from an initial to a final time step.

Automata representations are not the only way to model systems in the framework of model-based diagnosis. For instance, the model-based approach in [1] by Armant, Dague and Simon that deals with distributed consistency-based diagnosis is focused

on theory representations. These encodings aim at providing efficient extractions of minimal diagnoses.

2.3 Online Mode Estimate Update

If the systems to diagnose are static, like the ones covered by the Sherlock-style mode estimation [10], there is no need to update the mode estimate. In this thesis, we are dealing with dynamic systems that evolve over discrete time steps, and we want to integrate new observations each time step to monitor the system evolution.

The approach of Martin, Chung and Williams in [23] improves the accuracy of transition probability computation compared to the regular PHCA estimation by using observation functions, whereas this thesis focuses on a tractable simplified representation and encoding of transitions. This paper extends the Best-First Belief State Enumeration (BFSE) with the Best-First Belief State Update method (BFBSU) [24] by eliminating a simplification concerning the observation probability distribution. The improvements in their work and the improvements in this thesis are not mutually exclusive, but we chose to focus on the scalability of the computation for large models, such as the one presented in the next chapter.

As compared to Mikaelian, Williams and Sachenbacher in [28] who propose to recompute the mode estimates, or belief state, for a whole time window at every new time step, in this thesis we keep track of the previous state trajectories to extend them by one time step every new time step. We first compute enabled transitions, then append some previously computed trajectories at each time step so that a predefined number of most-likely estimates can be computed. Therefore, our approach needs two passes of constraint optimization, as explained in Chapter 5, one to compute transitions, and one to decide which trajectories can be extended according to the new observations.

Now that we have presented some background and related work about distributed mode estimation, we will more precisely state the problem that we tackle in this thesis in Chapter 3.

Chapter 3

Problem Formulation

This chapter formally describes the problem of distributed mode estimation of discrete dynamic systems. Model-based diagnosis of embedded systems can be achieved through a method called mode estimation. *Mode estimation* considers that system components can probabilistically transition between several behaviors, or *modes*. Its objective is to determine the hidden *state* of the system in the form of the set of component modes. Estimation accuracy is key to troubleshooting failures, should they occur, with the best likelihood possible. For instance, a broken relay and a broken power supply may both explain a system failure; and if power supplies are known to be more reliable than relays, without additional information, the diagnosis: "the relay is broken" may be more likely than: "the power supply is broken".

We first present the objective (Section 3.1) and the inputs and outputs to our mode estimation problem (Section 3.2). Then in Section 3.3, we discuss the assumptions and hypotheses of the problem, and we propose a motivating example (Section 3.4).

As specified earlier in our approach in Chapter 1, the diagnosis problem is formulated internally into a series of a constraint optimization problem. Hence, distributed diagnosis is achieved by solving this constraint problem in a distributed manner. Therefore, Section 3.5 formally defines the kind of optimal constraint problem we need to decompose and solve.

3.1 Objective

Mode estimation capabilities may be added to systems such as a power distribution circuit in order to perform diagnosis. More precisely, we want to perform distributed *N-step*, *K-best* mode estimation through the tree-decomposition of the dual constraint graph of the diagnosis problem.

Let us consider a simple electronic gate with three modes: *wire*, *inverter*, and *unknown*. Each of these modes corresponds to a behavior. In the *wire* mode, the input equals the output, whereas in the *inverter* mode, the output and the input have opposite boolean values. Finally, the last mode, *unknown* is unconstrained. It represents any possible situation, such as the case where the output is stuck to 1 or 0. This gate can receive a sequence of commands to switch between nominal modes, and commands may fail with a certain probability. The objective of distributed mode estimation is to be able to evaluate the most likely possible evolutions of this circuit, depending on the failure probabilities of the model, and observations. A set of diagnoses is updated at each time step, and the computation power needed to solve this diagnosis problem is distributed on several computers.

In the next section, we define more formally the inputs and outputs of the monitoring problem.

3.2 Model-Based Diagnosis Problem

The main input to our model-based diagnosis system consists of a probabilistic plant model example (3.2.1), a set of commands sent at each time step to the plant (3.2.2), and observations sent at each time step by the plant (3.2.3). Other user parameters are also fed to the system: the size of the diagnosis time window (number of steps of the time window, 3.2.4), and the number of candidate diagnoses to compute at each belief state update (3.2.5).

The main output is a diagnosis in the form of a set of most likely state trajectories (3.2.6). For instance, for a circuit made of two relays, a state trajectory may be: $\{p =$

$0.3 : (relay_1 \text{ is open}, relay_2 \text{ is closed})^{time_1}, (relay_1 \text{ is closed}, relay_2 \text{ is broken})^{time_2}\}$.

The objective of this thesis is to present a distributed method that computes a set of most likely state trajectories. To that extent, we choose to frame this problem into an optimal constraint satisfaction problem and solve this problem in a distributed manner. An OCSP is a hard-constraint satisfaction problem in proposition state logic with an associated objective function. OCSP's will be formally defined in Section 3.5.

Definition 1 (Model-Based Diagnosis Problem)
A model-based diagnosis problem is a quintuplet

$$(Plant\_Model, Commands, Observations, N, K)$$

, where

1. $Plant\_Model$ represents the set of system components and their interactions (Subsection 3.2.1);
2. $Commands = \{c_1, ..., c_{numberOfCommands}\}$ models the set of commands sent to the system;
3. $Observations = \{o_1, ..., c_{numberOfObservations}\}$ contains the sensor data (Subsection 3.2.2);
4. N is the size of the observation history used to monitor the system (Subsection 3.2.4); and
5. K is the number of state trajectories to compute (Subsection 3.2.5). □

3.2.1 Input: Probabilistic Plant Model

The core of a model-based diagnosis method is a plant model representation. In this thesis, we deal with probabilistic representations, as we look for the most likely diagnoses. The plant model describes all the components (also called modules), their interactions, inputs and outputs, and their respective probabilistic transitions.

Definition 2 (Plant Model)

A plant model is a pair $(Components, Interactions)$, where

1. $Components = \{a_1, ..., a_{numberOfComponents}\}$ is a set of components (Definition 3); and

2. *Interactions* is a set of global constraints involving state variables shared by different components, or global variables. □

For instance, the connection between two relays may be represented by the following constraint: $Input(Relay_2) = Output(Relay_1)$.

We suppose that each component can only act in a finite number of ways, each of which is called a *mode*. A mode is associated with a constraint representing a component behavior. For instance, a `Closed` mode for a relay would be represented by a constraint such as *input* = *output*. In the particular case of a multithreaded-software representation, if each thread corresponds to a mode, the software model may be in several modes simultaneously. We say that several modes are `marked` at the same time. Thus, the state of a system is determined by the set of marked modes, or *marking*.

Definition 3 (Component)

A component is a triplet $(Modes, State\_Variables, Transitions)$, where

1. $Modes = \{m_1, ..., m_{numberOfModes}\}$ is the set of all the component modes;

2. $State\_Variables$ is a state of variables internal to the component; and

3. $Transitions$ is the set of probabilistic guarded transitions of the component (Definition 4). □

Each mode, or internal parameter can be modeled with discrete finite variables, and constraints are subsets of *allowed* combinations of variable assignments. Therefore, the whole internal state of the system is represented by a set of discrete variable assignments.

For instance, the `Closed` mode of a relay may be modeled by the following constraint: $input = output$.

The interactions between modules can be modeled by shared variables or global constraints whose *scope* involve private variables of different components. For instance, in a circuit where a generator output is connected to a relay input, a global constraint such that $output(generator) = input(relay)$ can be used.

Each component can *transition* at each time step between different modes. For each marking, different transition scenarios are possible, they are represented by different *branches* of a transition. At most one branch can be enabled, and the distribution probability for the branch choice is fixed in the model. Then, different sets of guarded markings correspond to each branch: at the next time step, the markings of the chosen branch are enabled if their respective *constraint guards* `hold` at the current time step.

Definition 4 (Transition)
A transition is a triplet $(Origin, Branches)$, where

1. $Origin$ is a mode assignment $mode = value$ triggering the constraint; and
2. $Branches = \{b_1, ..., b_{numberOfBranches}\}$ is a set of probabilistic branches (Definition 5). The sum of the branches probabilities is not greater than 1. □

For instance, a relay in a closed mode may have two transition branches, one nominal, likely transition element, and one not nominal, unlikely transition element.

Definition 5 (Branch)
A branch is a triplet $(Probability, Guarded\_Markings)$, where

1. $Probability$ is the probability that the branch is enabled if the transition is triggered; and
2. $$Guarded\_Markings = \{gm_1, gm_{numberOfGuardedMarkings}\}$$
   is such that $gm_i = (guard_i, marking_i)$. If the constraint $Guard_i$ holds at a certain time step, then the $Marking_i$ of modes holds at the next time step. □

For instance, the nominal transition branch of a relay can contain two sets of guarded destinations: $(command = cmd\_open, \{relayisopen\})$, and $(command = cmd\_close, \{relayisclosed\})$.

In this thesis, we represent plant models with an automaton abstraction, a PHCA model introduced by Williams [39]. We define PHCA and present our related problem encoding later in Chapter 5.

In Subsections 3.2.2 and 3.2.3, we go into the details of the way to represent commands and observations.

3.2.2 Input: Commands

In a model-based representation, one can model user or controller commands sent to a plant as assignments to variables in the plant model. These assignments can be described at each time step with unary constraints of the type

$$command\_variable = value$$

. Command variables are often present in transition constraint guards.

3.2.3 Input: Observations

An *observation* is an assignment to an observable variable, that can be read or measured on a plant. Like commands, observations are modeled by unary constraints. They are added at each time steps to the theory of the model-based encoding, in the form

$$observable\_variable = value$$

. In this thesis, we consider that there is no uncertainty on observations. Hence, if no observation is made on a certain variable, we consider that the observation occurred if the corresponding assignment is consistent with the model.

3.2.4 Parameter: Number of Steps

The number of steps N is the size of the time window. We need to compute the most-likely trajectories instead of the most likely states at each time step in order to detect delayed symptoms. Hence, the estimations are based on a certain history of observations [27]. We could theoretically base the computation on all previous states, but a fixed window size appeared to be a trade off between problem complexity and accuracy. In our method, we want to be able to move the time window at every new time step, updating the belief states by appending the state trajectories.

3.2.5 Parameter: Number of Candidates

Computing the most-likely trajectory based on a history of N time steps increases the ability to detect delayed symptoms. Now, in order to increase the probability to find an exact diagnosis, we compute a set of K most-likely trajectories. Hence, if the most likely estimate is does not represent the actual state trajectory of the system, a set of the following most-likely candidates have a high probability to contain the actual state evolution of the system. In our problem framing, the parameter K represents the *depth* of the candidate search. If at least K trajectories are consistent with the theory, then K candidates are generated. Otherwise, the problem output consists of all the possible trajectories.

Additionally, at each time step, the state estimates can be extended depending on new observations, commands. and previous estimates. Hence, by computeing a set of optimal trajectories instead of a single trajectory estimate, we decrease the probability to end up without any estimate, if the previous most-likely diagnosis proves to be erroneous.

3.2.6 Output: State Trajectories

The output of our mode estimation algorithm is a list of *state trajectories* ordered by decreasing probability. Each state is a full assignment of mode variables at a certain time step. A trajectory is a sequence of consecutive states inside a certain

time window. The number of candidate trajectories is bounded by K (Subsection 3.2.5) and the length of the time window is N, the number of time steps that the window covers (Subsection 3.2.4).

Definition 6 (State Trajectory)
Let $M = \{m_1, m_{numberOfModes}\}$ be the set of mode variables, N the size of the estimation time window. A state trajectory is a pair $(p, trajectory)$, with:

- p the trajectory probability, and
- $trajectory = (state^i, ..., state^{i+N-1})$.

Each state is indexed by a time step and it is in the form:
$state^i = (m_1^i = v_1^i, ..., m_{numberOfModes}^i = v_{numberOfModes}^i)$, with v_j^i mode values. □

We have introduced the inputs and outputs to our estimation problem. Now we will describe more precisely the scope of this problem.

3.3 Assumptions

In this section, we state the context of the estimation problem we want to solve, and we present our assumptions about the behavior of the system we want to monitor.

This thesis focuses on mode estimation of discrete systems. This means that the behavior of each system component, including the internal behavior, can be modeled by finite and discrete-domain variables. Moreover, we assume that we know a model of the system beforehand; we do not need to infer it from experiments. We model system evolution with synchronous transitions. This means that our monitoring method either applies to synchronous systems, or that time step divisions are small compared to characteristic transition times.

From a probabilistic perspective, we assume that individual components are independent. *A fortiori*, the probability of failure of one component does not depend on the state of any of the other components. We also assume that the behavior of the system is Markovian. In other words, the state of the system at a certain time

step only depends on its state at the immediately preceding time step, and not on the whole history. Finally, we assume the rules that govern the system do not change over time: the system model is invariant by a shift in time.

Now that we have stated our assumptions (3.3) and have described our inputs and outputs (3.2), we can provide a better idea of the problem we are solving with an example.

3.4 Realistic Example

A plant model is the main input to the diagnosis problem, so we present a plant model example to illustrate this concept. In this section we introduce an example power distribution circuit taken from a naval vehicle system diagnosis [35], and the approach we will use to design a credible and practical monitoring scenario. This example will be detailed throughout the thesis and used to illustrate our methods. The example reflects a real-world situation, consisting of a large-scale system involving various types of components interacting with each other.

The power distribution plant model example presented here contains components that can receive commands to toggle between different modes. These mode transitions are probabilistic, in the sense that mode changes have a certain probability not to obey commands, and this deviant evolution is unpredictable. (For instance, if an unlikely, ill-nominal transition branch is enabled, a component may transition into an unknown mode, no matter which command it received.) The description of the circuit consists of the different components, their possible behaviors and their dynamics.

3.4.1 Component Description

The circuit we want to monitor is composed of four types of devices: generators (also called power supplies), relays, power panels, and wires. By switching relays between passing and blocking states, the system is able to power some set of power panels with some set of generators. A power panel is powered by a power supply if they are connected by a sequence of closed relays. Figure 3-1 displays the structure of

the circuit. Power supplies and relays may receive commands to power `on` and `off`, or `close` and `open`, respectively, as shown in Figure 3-2. These components have a certain probability to fail and they can be `reset`.

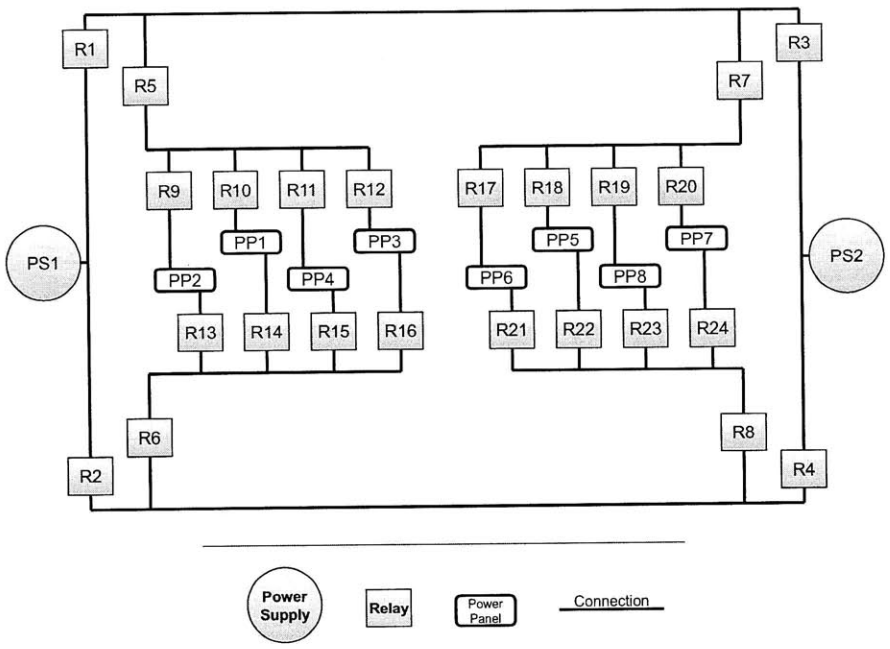


Figure 3-1: Redundant Power Distribution System Model

The modes of the different components are described as follows.

- A power supply can be in a `Powered` ($output = 1$), `Not Powered` ($output = 0$), or `Unknown` mode.
- A relay can be in a `Closed` ($input = output$), `Open`, or `Unknown` mode.
- A power panel can be in a `Nominal` ($power = (input_1|input_2)$) or `Unknown` mode. Power panels model all the external devices that are powered by the power distribution system, such as pumps, or A/C systems.

Our model rules out problems of faulty conduction or delivery capacity of wires: perfect wires instantly provide full power when they are connected to a powered generator, and the power state is a boolean value. The circuit has multiple redundancies,

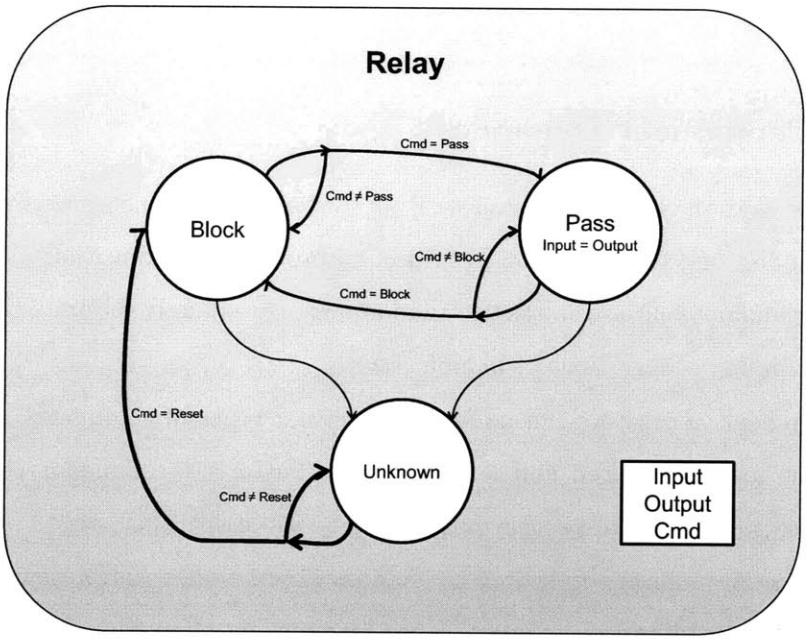


Figure 3-2: Model of the behavior of a relay

because generators are connected to multiple relays and power panels have two inputs, so there may be multiple relay switching scenarios that power the same set of power panels with diverse generator configurations. In this model, we exclude the possibility of short-circuit occurrences, supposing that generators are properly protected and grounded. Moreover, each component is always in a well-defined mode, and we suppose that the mode transitions are synchronous.

The next subsection describes our model for the dynamics of the circuit components, including when each of them can receive an individual command every time step.

3.4.2 Component Dynamics

We suppose that the different modules of our system evolve synchronously between different modes, and this evolution is subject to probabilistic transitions. As transitions of each component are subject to randomness, we can only estimate the system dynamics through observations of hidden states.

In our model, generators can nominally receive 2 types of commands, switching between the modes `Powered` and `Not Powered`. `On` or `Off` commands change the values of the generators' respective outputs at the successive time steps.

Relays are commanded to switch to `Pass` or `Block` modes, and a power panel is considered powered if one of its inputs is fed by a current. However, if a generator, a relay, or a power panel does not exhibit a nominal behavior, that is, does not switch correctly to the commanded mode, it is in an `unknown`, off-nominal mode.

It is possible to exit off-nominal modes by sending the `Reset` command to a given component. The `Reset` commands revert components to their default mode, which are `Not Powered` or `Block`.

3.4.3 Monitoring Problem

The monitoring problem in this example consists of determining the most likely mode configuration of each component, and should a failure happen, it determines which

components are most likely to be in an unknown mode. We suppose that the probability of each failure is independent between components, and that the probabilistic evolution of the circuit is a memoryless process. This means that the outcome of a probabilistic transition choice is not influenced by previous random choices: the associated random variables are independent.

We need to model commands sent to the system and observations obtained from the system. These observations correspond to a series of power measurements and commands over several time steps. Observable variables have to be determined before the experiment is started. They can be for instance the measurements of power at the connections of some of the components, such as power panels or generators. Observable quantities can be related to the position of power probes on a physical simulation.

In this thesis, a plant model is represented by a PHCA, introduced in previous work [39], then encoded into a constraint optimization problem with the method described in Chapter 5. The type of optimization problem we use is defined in Section 3.5.

3.5 Optimal Constraint Problem

One of the main challenges in this thesis is to solve an Optimal (Hard-) Constraint Satisfaction Problem in a distributed manner. This section defines OCSP's. The algorithm described in Chapter 4 presents a method to solve OCSP's in a distributed manner. The definition of OCSP below is analogous to the one presented by Williams and Ragno in [33]. This definition of OCSP's differs from the standard valued CSP's definitions by the fact that all the valued soft constraints of OCSP's are unary.

Definition 7 (Optimal Constraint Satisfaction Problem)
An Optimal Constraint Satisfaction Problem is a sextuplet:

$$(State\_Variables, Domains, Constraints, Decision\_Variables, Costs, Valuation)$$

with the following properties.

- *State_Variables* is a list $\{V_1, ..., V_n\}$ of state variables, each of which has a domain of values $D_i = \{a_1, ..., a_{m_i}\}$ of values in *Domains*. We use discrete, finite domains in this work.

- *Constraints* is a set $\{C_1, ..., C_m\}$ of hard constraints that the variable assignments must respect. A constraint specifies subsets of allowed partial assignments for some of the variables. In other words, the set of partial assignments is defined by:

$$Constraint(C) \subseteq \{ \bigotimes_{V_i} D_i \mid V_i \in Scope(C) \}$$

  for each constraint C. ($\bigotimes$ is the Cartesian product symbol.) The set of variables affected by a constraint is called the *scope* of the constraint.

- *Valuation* is a valuation structure on a set E, $(E, \preceq, \oplus, \bot, \top)$. E is a set of costs. It is totally ordered by $\preceq$, $\top$ and $\bot$ are respectively maximum and minimum elements of $(E, \preceq)$, and $\oplus$ is an associative, commutative, and monotone (for $\preceq$) operator with neutral element $\bot$ and absorbing element $\top$. It defines the objective function of the OCSP that has to be optimized under the constraints C.

- *Decision_Variables* is a subset of *State_Variables* whose individual assignments have a cost (see definition of *Costs*).

- *Costs* associates each individual assignment of decision variables to a cost. The valuation structure describes how to compute the value of an objective function by associating with the operator $\oplus$ (product, sum, max...) the costs of all assignments. □

For mode estimation, maximize over the probability of diagnoses; hence, we could use

$$Valuation = ([0, 1], \geq, \cdot, 1, 0)$$

but we choose a slightly different but equivalent representation detailed in Chapter 5.

An OCSP involves a set of *state variables*, each of which can be assigned a value in their domain. Some of these variables are called *decision variables*, which means that each value in each of their respective domains is associated with a cost, which depends on the probabilistic nature of the model in the case of mode estimation. An associative operator computes the total cost for each global assignment to the decision variables.

The objective of the optimization is to *determine the global assignment with the optimal cost*, while respecting constraints that may only make some partial assignment legal.

In our encoding of a monitoring problem, the costs of decision variable values are negative real numbers (obtained from logarithms of probabilities), the associative function is the sum (+), and the comparison function is the real number regular comparison operator ($\geq$). We maximize the total cost of an assignment, which is the likelihood of our solution. The feasible set is defined by a list of disjunctive clauses that must all be true, which is more compact than an explicit definition of allowed sets of partial assignments. We explain this encoding in our mode estimation case and explain how to use it for online monitoring in Chapter 5.

The trellis diagram structure used in this work corresponds to an optimization problem that may be solved in a distributed manner. While some distributed optimization methods under constraints already exist (as presented in Chapter 2), they are most often not well suited to solving for a set of optimal feasible solutions, rather than a single solution.

Now that the problem is formally defined, we can present our distributed approach to solve OCSP's in Chapter 4, and our framing of a diagnosis problem into an OCSP in Chapter 5.

Chapter 4

Distributed Optimal Constraint Satisfaction Problem Solving

In Chapter 3, we defined a discrete Optimal Constraint Satisfaction Problem (OCSP) as a constraint satisfaction problem in which costs are associated with the assignments of some individual decision variables. In this chapter, we propose a method that: solves OCSPs in a distributed manner, and enumerates solutions in a best first order.

The approach we use divides the initial problem into smaller sub-problems, solves them in a distributed manner, and then combines their solutions. The merged solution is identical to the solution of the original problem. Here, we use the term 'distributed' to mean a system where each sub-problem has its own dedicated processing and local memory. We cannot afford to distribute only the processing power without also distributing the storage (memory), because we would like to limit the need for communication between computers. For example, an implementation could distribute the subproblems over networked computers or processes in a single computer. To enforce the consistency of variable assignments between different subproblems and to share information such as conflicts, we employ a simple message passing system.

The purpose of this approach is to provide the algorithmic basis for robust, distributed mode estimation. Distributing processing increases the robustness of mode estimation against processor or communication failure. Rather than monolithically duplicating the monitoring system, our approach allows alternate ways to achieve

computation or communication fault tolerance. For instance, a duplication of each subproblem's computational unit is able to handle more than a single failure if at least one unit of each subproblem remains healthy. Meanwhile, enumerating multiple solutions increases the likelihood of identifying the correct mode. As described in the problem statement in Chapter 3, finding a single optimal solution is not sufficient for reliable mode estimation if it does not match the actual system state.

4.1 Method Overview

The objective of this chapter is to present a method to solve an OCSP in a distributed manner. The input of the problem is an OCSP and a positive integer K. As explained in Subsection 3.2.5, K is the size of the leading set of solutions that the solver is looking for. If the number of feasible solutions to the OCSP is less than K, then the algorithm returns all possible solutions. The output of the method is a list of solutions to the OCSP sorted by decreasing utility, for example likelihood.

The distributed optimal constraint satisfaction method consists of two stages.

1. First, the problem is decomposed into a tree of connected OCSP subproblems.

2. Next, distributed search is performed by enumerating solutions at each node of the tree, and stitching them by passing messages on the edges, so that solutions of the initial OCSP problem can be enumerated in a best-first order.

4.2 Problem Decomposition

The first stage of the distributed search is a decomposition of the OCSP into subproblems. The objective of this decomposition is to create a tree of subproblems such that the *composition* of all subproblems conveys the same meaning as the original OCSP. Each subproblem is an OCSP and the edges represent consistency constraints between each subproblem.

The problem decomposition involves five steps.

1. First, we build a binary dual constraint graph of the CSP (4.2.1).

2. Next, we perform a standard joint-tree clustering on the graph, producing an acyclic graph of partially-coupled CSP's (4.2.2).

3. After some simplifications (4.2.3),

4. this graph becomes a rooted tree by picking a root as a centroid of the acyclic graph (4).

5. Finally, we transform these CSP's into OCSP's by adding costs such that globally optimal solutions can be retrieved by merging solutions from OCSP's in each node (4.2.5).

4.2.1 Constraint Graph

One of the main challenges in a distributed optimization process is to decompose the problem. A problem decomposition may be based on the structure of the problem. The structure of constraint satisfaction problems is traditionally represented by constraint graphs [34, 11]. Previous work on soft-constraint optimization problems use primal constraint graphs associated with tree-clustering to decompose the solution computation [37]. This thesis presents a method to decompose and solve optimal hard-constraint satisfaction problems by using a dual constraint graph to represent the CSP component of the problem.

Let us consider a constraint graph associated with a constraint satisfaction problem. Each node corresponds to a variable and an edge connects two variables if and only if at least one constraint involves these variables together. This is called the *primal constraint graph* (Figure 4-1). Primal graphs are used in the decomposition method that inspired the approach presented in this section [27].

A first observation is that when a constraint affects more than two variables, we encounter a cycle in the primal graph. When there is a cycle in a constraint graph, it cannot be solved by arc-consistency. Hence, cycles of subproblems are hard to solve

in a distributed manner as they require a lot of communication between subproblems. We will develop this argument later in this subsection.

Definition 8 (CSP Primal Dual Constraint Graph)
Let $\{Variables, Domains, Constraints\}$ be a CSP. The primal constraint graph associated with this CSP is an undirected, binary graph defined as $\{Nodes, Edges\}$ such that:

1. $Nodes = \{(v, Domains[v]) \mid v \in Variables\}$, and
2. $Edges = \{(\{m, n\}, c) \mid m, n \in Nodes,\ c \in Constraints, \{var(m), var(n)\} \subseteq scope(c)\}$. □

Example 1 (Example of CSP Primal Graph)
In Figure 4-1, we represent the primal graph associated with a constraint satisfaction problem involving five variables: $\{a, b, c, d, e\}$, and four constraints which respective scopes are (a, b, c), (a, b), (b, c, d), and (a, e). The associated constraint graph contains six binary edges (Figure 4-1). □

In the context of a distributed search algorithm for solving CSP's, cycles tend to make the distribution processes inefficient, due to increased communication between subproblems. If a graph is acyclic, arc-consistency between nodes ensures global consistency, therefore, the need for communication between nodes is reduced. In addition to this issue, it is worth noting that the search complexity of typical search algorithms is exponential with respect to the size of the cycles while it is polynomial with respect to the depth of the tree [34].

While Sachenbacher and Williams used a primal constraint graph to solve its valued constraint satisfaction problem [37], we chose to work with a dual graph, because we want to ensure that each node contains at least one constraint. A dual constraint graph can be interpreted as the constraint graph from the dual constraint problem.

One can see a soft-constraint satisfaction problem as the dual of a hard-constraint satisfaction problem by using a constraint-variable duality paradigm. Hence, there

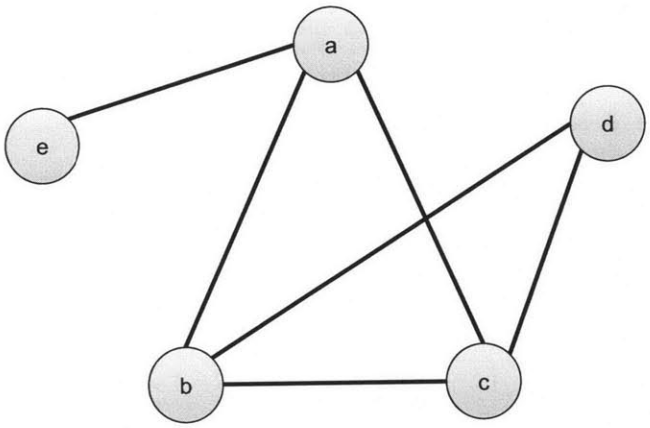


Figure 4-1: Primal Graph Example

are some similarities between the method presented in this thesis, that uses hard constraints and a dual graph, and the previous method, that uses soft constraints and a primal graph. A soft-constraint satisfaction problem defines preferences by putting weight on the different outcomes of the constraints, whereas in our case, costs depend on each of the decision variable assignments. If we consider that in the case of valued-constraint satisfaction problems, the objective is to assign a *value* (a tuple) to each constraint, while maintaining variable consistency between all the constraint assignments, one can reformulate the problem as a hard-constraint satisfaction problem with weighted variable domains. With this transformation, consistency constraints (equality) between tuples plays the role of hard constraints (as defined in Chapter 3).

Using standard terminology, let us call the set of variables affected by a constraint the *scope of the constraint*, and call the set of values that a variable can take a *variable domain*. The *relation* of a constraint is the set of allowed assignments to variables in its scope. More formally, it is a subset of the cross-product of the domains of the variables in its scope which satisfy the constraint. With this terminology, we can define the dual graph in terms of the primal graph. The domain of the dual variable associated with a given primal constraint is the relation of the primal constraint. The dual constraints enforce the fact that a primal variable, shared in the scope of several primal constraints, should take the same value in all the constraints, thus the dual constraints are consistency constraints between elements of the dual variables.

The following paragraphs present the formal definition of a CSP binary dual constraint graph, based on the definition of a dual constraint hypergraph. The OCSP problem decomposition described in the current section is based on a decomposition of the CSP binary dual constraint graph.

We now define the dual constraint graph in both its hypergraph form and binary form, which we will use, and illustrate variable-constraint duality in a figure.

Definition 9 (CSP Dual Constraint Hypergraph)
Let $\{Variables, Domains, Constraints\}$ be a CSP. The Dual constraint hypergraph associated with this CSP is a hypergraph defined as $\{Nodes, Hyperedges\}$ such that:

$Nodes = \{(c) \mid c \in Constraints\}$, and

$Hyperedges = \{(h, v, d) \| v \in Variables,\ d = Domains[v],\ and\ h = \{n \mid n \in Nodes, v \in scope(n)\}\}$. In other words, there is one hyperedge per variable, which connects all nodes that use the variable in a constraint. □

Definition 10 (CSP Dual Binary Constraint Graph)

Let $\{Variables, Domains, Constraints\}$ be a CSP. The Dual constraint graph associated with this CSP is an undirected, binary graph defined as $\{Nodes, Edges\}$ such that:

$Nodes = \{(c) \mid c \in Constraints\}$, and

Edges are defined as the set of binary edges spanning the hyperedges of the dual constraint hypergraph, and they are labeled with the sets of variables in common between the scopes of adjacent nodes. □

Example 2 (Illustration of Constraint-Variable Duality)

For instance, Figure 4-2 and Figure 4-3 respectively show the constraint hypergraph and the dual binary constraint graph of a simple constraint satisfaction problem. Notice that the edges are labeled by the sets of shared variables between the constraints contained in the nodes. For example, nodes with scopes $\{a, b, c\}$ and $\{b, c, d\}$ are connected with an edge labeled $\{b, c\}$, which is the intersection between the two aforementioned scopes. □

From now on, we will refer to dual binary constraint graphs as dual constraint graphs, or simply dual graphs.

Elaborating from the earlier discussion, there are several advantages to working with a dual constraint graph.

In a primary constraint graph, every constraint that involves more than 2 variables leads to a cycle. Dual graphs avoid the production of a cycle every time a constraint is more than binary. More importantly, since we are dealing with constraint satisfaction problems with costs on the values of decision variables, associating nodes with constraints means each node is a self-contained sub-problem.

Given an OCSP, the computation of the binary dual constraint graph associated with the OCSP's CSP component is the first step of the problem decomposition

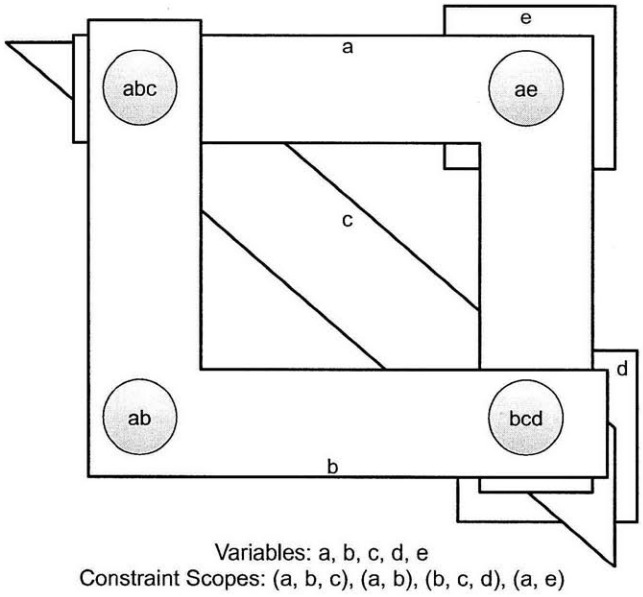


Figure 4-2: Dual Constraint Hypergraph Example

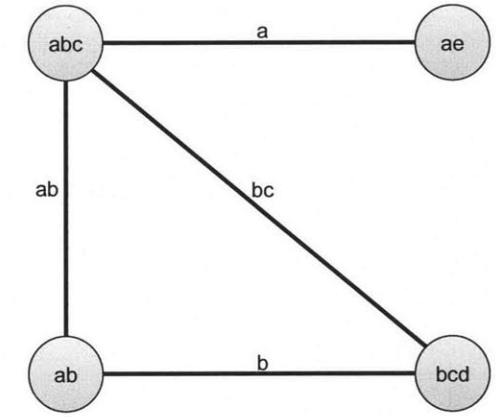


Figure 4-3: Variable-Constraint Duality Example

method we present in this section (Algorithm 1).

Algorithm 1 Constraint Graph Building Algorithm

▷ The input of this algorithm are an OCSP and a standard tree-clustering heuristic.

graph ← *newgraph*
hyperedges ← ∅
for *variable in problem* **do**
    hyperedges[*variable*] ← ∅
end for
for *constraint in problem* **do**
    addNode(*constraint*, *graph*)
    for *variable in scope*(*constraint*) **do**
        hyperedges[*variable*] ← {*Node*(*constraint*)} ∪ *hyperedges*[*variable*]
    end for
end for
for *hyperedge* ∈ *hyperedges* **do**
    addHyperedge(*hyperedge*, *graph*)
end for ▷ *graph* is now the dual constraint hypergraph associated with the OCSP

Subsection 4.2.2 presents the characteristics of the tree-decomposition method we use to compute a clustered, acyclic dual constraint graph.

4.2.2 Tree Clustering into an Acyclic Graph

The objective of the current Section, 4.2, is to present our method to decompose an OCSP problem. In Subsection 4.2.1, we showed how a dual constraint graph can represent the structure of a CSP. In this subsection, we justify and present tree clustering, the second step in the decomposition. Applied on a constraint graph, tree clustering produces an acyclic graph by clustering nodes. When two nodes are merged (4.2.2), a node ends up containing one or more constraints, the scope of a merged node is the union of the scope of the constraints of the original node, and each binary edge represents an external consistency condition between two nodes.

Example 3 (Example of Tree-Clustering)

In the example Figure 4-4, we can consider that we have a problem with 4 constraints,

$C1, C2, C3$ and $C4$, where the first three share variables, but $C4$ only shares variables with $C1$, like in Figure 4-1 and Figure 4-3. Subproblems which share variables are correlated, in the sense that the assignments to these shared variables must be consistent in each node. The first graph is a binary dual constraint graph. It shows the correlations between constraints. The first three nodes of this graph form a cycle. By gathering $C2$ and $C3$, two nodes are clustered. This clustering makes the second graph acyclic. In this clustered graph, each node is a subproblem, and the edge represents a correlation between these subproblems: $C1, C2$, and $C3$ share a variable. □

A decomposition of the CSP is obtained by mapping the resulting clustered constraint graph into a tree of CSP subproblems. This section justifies and presents the characteristics of this decomposition.

To limit the need for communication without sacrificing computational efficiency, we decide to use a tree decomposition over a constraint graph, after mapping the problem on a graph structure [34], in a way inspired by the decomposition in [37]. The main characteristic of the graph on which the distribution process is based is that it is acyclic. Therefore it can be analyzed as a tree of subproblems. Additionally, a tree clustering enforces the fact that for any given variable v, the set of nodes containing v is connected. Consequently, if the initial problem contains completely decoupled components, the clustering will lead to a forest instead of a tree.

In this thesis, we use a standard bucket-tree decomposition method like the one presented by Dechter in [11]. The tree produced by a clustering method depends on the structure of the problem we want to solve. (For instance, in the cases when the constraint graph is complete, a clustering method may produce a graph with very few nodes and the decomposition may be impractical.) We must make sure that all constraints and all possible results of the initial problem are induced in the set of tree-decomposed sub-problems of the reformulation. Additionally, as we solve an optimal constraint satisfaction problem, the different costs have to appear in the decomposed version of the problem. The decomposition of costs is presented later in Subsection 4.2.5.

Algorithm 2 presents the outline of an OCSP decomposition process. First, the dual constrain graph associated with the underlying CSP. Then, a standard tree-clustering method is applied on the graph, returning a forest of acyclic graphs. The resulting nodes in the acyclic graph are labeled by sets of constraints. The content of each node is then updated to contain a standalone OCSP. Finally, the addition of costs on the decision variable assignments is performed by the method *addCostsToNodes*. Its behavior is presented later in this section. The elements contained in each node in the graph output by the algorithm are:

1. constraint definitions,

2. variable definitions (domains), and

3. decision variable definitions.

Algorithm 2 Constraint Graph Building Algorithm

1: $treeCluster(graph, heuristic)$ ▷ A tree-decomposed binary dual constraint graph
2: **for** *node in graph* **do** ▷ Each node is updated to contain a standalone OCSP subproblem.
3: $localConstraints \leftarrow content(node)$
4: $localVariables \leftarrow variables \cup scope(localConstraints)$
5: $localDecisionVariables \leftarrow decisionVariables \cup scope(localConstraints)$
6: $content(node) \leftarrow (localVariables, localDecisionVariables, localConstraints)$
7: **end for**
8: $addCostsToNodes(graph)$ ▷ Costs are added to the decision variable descriptions.
9: **return** *graph*

To perform clustering, we have to find a suitable ordering (encoded as a *heuristic* in the tree-decomposition [11]) for the sub-problems so that it limits the amount of backtracking during the search. We could separate all variable assignments in the tree as in previous work by Sachenbacher and Williams [37], however we prefer to rely on a search library using a more specialized method to deal with the variables clustered inside a single sub-problem (with a powerful conflict-directed A* algorithm-based solver, OPSAT [42]). In the method presented in this thesis, each node contains

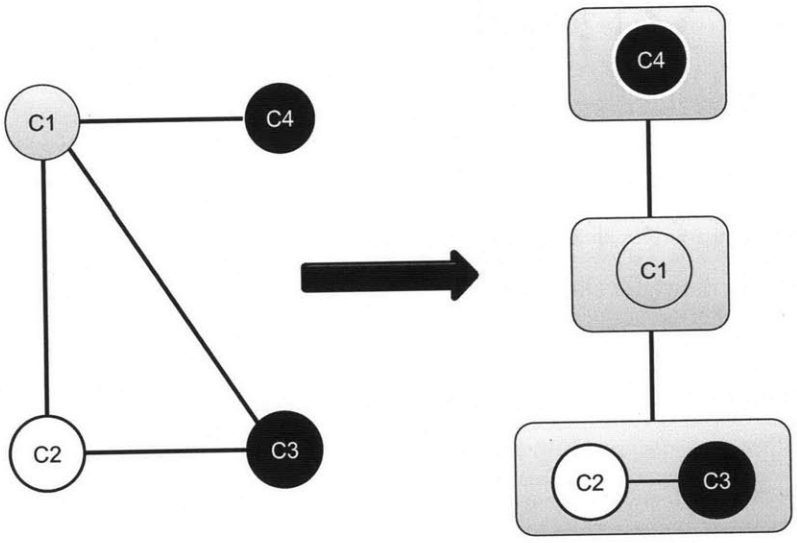


Figure 4-4: Tree Clustering Example

a full standalone subproblem with variables and constraints. We use a coarse grain approach to distribute OCSP solving, in which each subproblem is solved by conflict-directed A*. On the contrary, a fine grain DCOP approach would separate each variable in a process. Hence, we can take advantage of the power of a centralized conflict-directed A* inside a node, and we can use the algorithm presented in this chapter to deal with the distributed search inside the tree of subproblems. The fact that we are mixing these two approaches is a key to the efficiency of our proposed method.

Remark 1 (Choice of Tree Decomposition Methods)
The purpose of this thesis is not to evaluate the performance of a tree clustering algorithm, but to present a distributed constraint solving algorithm that takes advantage of the properties of a tree-decomposed dual-constraint graph.

Tree decomposition leads to a cycle-clustering operation on the constraint graph. Note that the depth of the tree here is distinct from the concept of *tree-width* introduced by Robertson and Seymour in [34], tree-width being the minimum over all the tree decompositions of the maximum size of a cluster minus one. An *optimal* tree decomposition would minimize the tree width but the computation of this number itself is NP-complete so we are not looking for an optimal clustering in this work. Any clustering that leads to the suppression of cycles and respects the properties of a bucket-tree can be used in our method. □

In the next section, we present a post-processing step on the tree that gathers and merges some nodes again if they share non-decision variables.

4.2.3 Graph Simplifications

In Subsection 4.2.2, we presented the tree-clustering stage of the OCSP decomposition. So far, the previous subsections had only approached the problem as a regular CSP. In this section, we introduce a simplification on the tree structure that is related to the OCSP nature of the problem.

In an OCSP, a solution only consists of assignments to the decision variables. Non-decision variables are not explicitly assigned. For every candidate solution, the solver only checks whether there exist assignments to non-decision variables that are consistent with both the theory and the candidate solution. Moreover, for one given solution to the OCSP, there may exist several sets of assignments to non-decision variables, and all these configurations only count as one single solution. Our simplification proposes to cluster nodes so that all non-decision variables become internal to subproblems contained in each node.

This is why we propose to gather neighbor nodes which share non-decision variables, and to merge their local problems (Algorithm 3). Thus, after the merging procedure, any shared variable can only be a decision variables. It ensures that shared will always be explicitly assigned while enumerating local candidate solutions. This is necessary in order to check if two candidate solutions in two nodes are consistent with each other so that they are merged together.

The previous paragraph proposed a way to deal with the cases when non-decision variables are shared between two neighbor nodes by merging these neighbors. Another solution is to transform all the shared variables into decision variables. This transformation implies that each value of a shared non-decision variable is assigned a neutral cost. We decided not to favor this solution for three reasons.

1. First, the efficiency of the local candidate solution enumeration algorithm is partly due to a limited number of decision variables to instantiate, as it only needs to check for internal consistency for all the non-decision variables. Moreover, adding costs with equal preference to the values of some variables ruins a search heuristic based on a best-first enumeration as it is not able to distinguish which hypothesis to test first.

2. The second reason is the artificial inflation of the state space: for the overall problem we only care about the decision variable assignments, and the configuration of the internal (non-decision) variables does not matter, so different internal assignment can lead to only one global solution.

3. Finally, from a practical point of view, the constraint graph of real-world problems is so large that its construction and the enumeration of all the edges would take too much time without a preprocessing consisting in merging the aforementioned nodes.

Algorithm 3 ensures that nodes that share non-decision variable be merged together. For each non-decision variable v, nodes whose scope contains v are gathered in a list, then clustered together inside a single node. Next, edges are updated accordingly.

Algorithm 3 Problem-Merging Algorithm

```
1: clusters ← ∅
2: for variable ∈ (variables \ decisionVariables) do
3:     cluster ← ∅
4:     for Node | variable ∈ scope(Node) do
5:         cluster ← cluster ∪ {Node}
6:     end for
7:     mergeCluster(cluster)          ▷ Merges all the nodes inside the cluster set
8: end for
```

4.2.4 Choice of a Root

Now that we have expressed the initial optimal constraint satisfaction problem as a tree of coupled subproblems, we need to choose a root. The root node defines the starting point of our distributed search algorithm. There are several ways to pick a root. One can choose a leaf, or conversely the highest-degree node, for instance. The complexity of backtrack search grows with the depth of the tree, hence we choose to select as the root the node that minimizes the depth of the tree. This corresponds to the *innermost* node of the acyclic graph.

Algorithm 4 takes a tree and returns a root node that minimizes the depth of the tree. It starts by looking at the leaves which are nodes with only one neighbor and progresses towards the interior until it reaches the root.

After a root is selected, we transform these CSP's into OCSP's by adding costs. The resulting tree of OCSP carries a decomposed version of the original OCSP with

Algorithm 4 Root-Choice Algorithm

```
1: procedure PICKROOT(Tree)
2:     if Tree = ∅ then
3:         return ∅
4:     else
5:         leaves ← ∅                                  ▷ This set is to contain the leaves.
6:         distFromLeaves ← ∅          ▷ Shortest distance from each node to a leaf.
7:         for node in tree do
8:             nodes ← ({node} ∪ nodes)
9:             if #neighbors(node) = 1 then
10:                leaves ← ({node} ∪ leaves)    ▷ Leaves are nodes with 1 neighbor.
11:            end if
12:        end for
13:        for node ∈ leaves do   ▷ Initialization of distFromLeaves for the leaves.
14:            distFromLeaves[node] ← 0
15:        end for
16:        root ← first(leaves)                        ▷ Arbitrary initial choice for a leaf,
17:        for node ∈ leaves do    ▷ to be updated to the farthest node from a leaf.
18:            queue ← {node}
19:            currentDistance ← 0 ancestor ← ∅
20:            while queue ≠ ∅ do ▷ queue is used to explore the tree, starting from
21:                currentNode ← pop(queue)       ▷ the leaves and going "inwards".
22:                for neighbor ∈ (neighbors(currentNode) \ ancestor) do
23:                    push(neighbor, queue) ▷ Nodes are added starting from outside.
24:                    if 1 + distFromLeaves[neighbor] < currentDistance then
25:                        currentDistance ← 1 + distFromLeaves[neighbor]
26:                    end if   ▷ distFromLeaves is updated during the exploration.
27:                    distFromLeaves[currentNode] ← currentDistance
28:                end for
29:                currentDistance ← currentDistance + 1
30:                ancestor ← {currentNode}
31:            end while
32:        end for
33:        maxDistance ← 0
34:        for node ∈ nodes do                  ▷ Picks the farthest node from any leaf.
35:            if maxDistance < distFromLeaves[node] then
36:                maxDistance ← distFromLeaves[node]
37:                root ← node                  ▷ argmax extraction in distFromLeaves
38:            end if
39:        end for
40:        return root
41:    end if
42: end procedure
```

the exact same constraints and solutions.

4.2.5 Adding Costs to Nodes

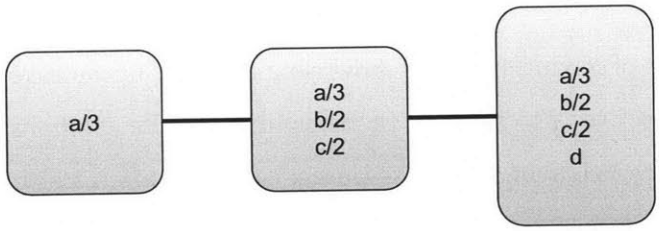
The purpose of this section is to describe how to decompose the objective function of an OCSP. The objective is to generate a tree-decomposed OCSP that has the same set of solutions as the centralized OCSP. This decomposition consists of a tree of OCSP subproblems. Moreover, we want the costs of the merged solutions to match their counterparts in the original monolithic problem. This section presents the reasons for our choice of splitting evenly the costs of shared decision variables between nodes.

In the case of a single or centralized problem, a simple summation of the individual assignment costs is enough to compute the total cost of a solution. On the other hand, one must be more careful when it comes to a set of decentralized problems. The issue is that some decision variables are repeated within several nodes. In this case, a straight summation of the costs of the solutions of each subproblem would double count some costs, producing an incorrect result.

Out of several possible and valid solutions to this question, we picked one that seems to be a good trade-off between simplicity and computation efficiency. We propose evenly dividing the cost of shared decision variables when they are present in several nodes. In Figure 4-5, we represent this repartition of cost between neighbors sharing variables. We symbolically describe the division of assignment costs by a fraction of variable names: $variable\_name/number\_of\_occurrence$.

Alternative methods have various problems. It is possible to let each variable be a decision variable at most once in the entire problem, but that way, consistency checks are made much harder; values must be projected from the subproblems on which they are explicitly computed into subproblems sharing the variables, and this constraint breaks ordinary best-first enumeration methods. Moreover, projections are constantly added and removed from neighbor subproblems, and the local optimal constraint satisfaction solver we are using does not allow constraint removal.

Another possibility would be to force the costs of the decision variable assignments to be zero in all but one of the nodes. However our local solver does not enumerate



Variables: a, b, c, d, e
Constraint Scopes: (a, b, c), (a, b), (b, c, d), (a, e)

Figure 4-5: Local Cost Computation

solutions as efficiently when non-discriminating costs are used. Then, a last option could be to keep costs as is in the subproblems but only count them once when merging candidate solutions. However that would ruin the pairwise best-first enumeration heuristic that we are about to present in Section 4.3.

The problem decomposition presented in this present section took an OCSP and decomposed it into a tree of coupled smaller OCSP's. The next section explains how our distributed method enumerates the optimal solutions of the original OCSP in a best-first order by passing messages between the nodes of the trees and computing on-demand solutions of the smaller OCSP's inside these nodes.

4.3 Distributed Problem Solving

The second stage of solving an OCSP is distributed search by message passing between nodes. The objective is to produce on-demand *candidate solutions* to subproblems and then to stitch neighbor solutions. Hence, communication inside the tree ensures that variable assignments are consistent in different nodes, and helps incrementally produce candidate solutions to larger subproblems, that are induced by the union of several nodes. At the end, search returns an ordered list of globally-optimal, feasible solutions to the initial OCSP along with their respective costs.

Our distributed search method involves several steps and algorithms.

1. First, we start by building a data structure on top of the tree, that gathers sets of nodes into pairs of subproblems. Each element of the `Pair` structure may be either a single node or another nested `Pair` of subproblems. This structure can be interpreted as a binary tree, as we explain in Subsection 4.3.1.

2. This pair-based data structure is used to enumerate optimal solutions by induction on the structure of the subproblems. An overview of the method is presented in Subsection 4.3.4.

3. We then present in Subsection 4.3.2 a best-first candidate solution enumeration algorithm that operates on a pair of subproblems to explain our distributed

search method. This algorithm works by expanding, on demand, the list of candidate solution of either of the two elements of the pair, and by trying to stitch candidates together, until the next pairwise-optimal solution is found.

4. The optimality or termination condition that ensures that a pairwise solution is indeed the i^{th} pairwise-optimal solution and (and commands to stop the search when the required i^{th} solution is found) is worth looking into inside a separate subsection (4.3.3).

5. Optionally, a conflict handling capability can be added to direct the search as shown in Subsection 4.3.5.

4.3.1 Binary Tree Structure

This subsection shows how we can gather nodes together to form subproblems so that the search of global solutions is seen as a sequence of choices in a binary tree.

Section 4.2 describes how to transform an OCSP instance into a tree of coupled OCSP problems. This set of problems satisfies the following properties.

1. If two nodes share a variable, then this variable is a decision variable.

2. For each variable v, the subset of nodes containing v is connected.

3. If local solutions to node subproblems are consistent, then a globally valid solution is generated by merging all local solutions, and the global cost of the global solution is the sum of the cost of each local solution.

The previous properties imply that nodes may be gathered so that a set of connected nodes can be seen from the rest of the tree as a subproblem, verifying the previous properties. Now we call *subproblem* the OCSP induced by merging a connected subset of nodes in the tree. Hence, we can recursively gather pairs of subproblems, starting by gathering pairs of nodes, until all the nodes are contained inside an outermost pair of subproblems.

Definition 11 gives a more formal definition of a `pair` of subproblems.

Definition 11 (Subproblem)

A `Subproblem` is either

1. a `pair` of `Subproblems`, or

2. a `Node`:

`Subproblem` := (`Subproblem`,`Subproblem`) | `Node`. □

For instance, consider the case where there are three subproblems, forming a tree, so there is no cycle. We call the central subproblem $SubProblem1$ and the two around it $SubProblem2$ and $SubProblem3$. $\{SubProblem1,\ SubProblem2\}$ is a subproblem $SubProblem0'$ from the point of view of $SubProblem3$, the local results of $SubProblem0'$ being stored next to those of $SubProblem1$. Hence, the global tree-decomposed problem may always be represented as a pair of subproblems. The same reasoning can easily be applied to any tree: by gathering subproblems together, we can apply a method analogous to the one for pair of subproblems.

Each subproblem can be a pair, or a node (Definition 11). Therefore, by induction, if we know how to enumerate pairwise optimal solutions, we can recursively apply the method to find the globally optimal solutions of a tree-decomposed problem.

A search for a global solution consists of:

1. candidate solution enumerations inside subproblems, and

2. pairwise candidate solution merging in order to enumerate pairwise solutions.

Thus, we can recursively create a virtual binary tree structure such that each `Pair` structure is the ancestor of its two subproblems. In this binary tree, the root represents the global tree-clustered OCSP problem, and each leaf is a node.

There may be several ways to gather nodes and subtrees into pairs of subproblems. In this thesis, we have chosen to work with a depth-first pairing, taking the root of each of the subtrees defined as the master-node for each pair (Figure 4-6). Hence, nodes only contain information concerning themselves, or their direct neighbors or

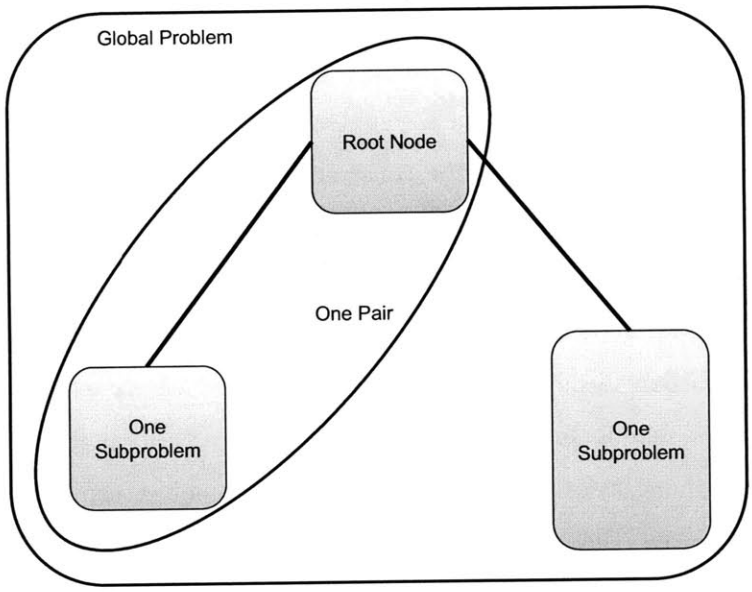


Figure 4-6: Double Structure Pairs-Subrees

pair partners. This choice of ordering, starting from the root node, allows us to make analogies with the projective model for the soft-constraint-based method in [37].

Now that a binary structure has been built on top of the tree of OCSP, we present an overview of our best-first enumeration algorithm on a pair of subproblems.

4.3.2 Pairwise-Optimal Solution Enumeration

Our objective is to find an algorithm to enumerate the solutions of a global problem, while looking at its decomposition into basic subproblems in a tree. First, let us observe how to evaluate the optimal solution of a pair of subproblems. We can assume that these problems have at least one variable in common.

A *candidate partial solution* is an assignment to all decision variables of a subproblem that is consistent with all the constraints of this subproblem. A pairwise solution is an assignment to all the decision variables of both subproblems satisfying all the constraints, and such that the assignments to shared variables is consistent. Recall that shared variables are decision variables (Section 4.2.3). This last condition is required, as decision variables are the only variables which are explicitly assigned during the solution search. Internal consistency checks are performed on non-decision variables when solving the OCSPs, but the solver does not return values for these variables. This is why consistency checks between two distinct subproblems are not possible if non-decision variables are shared.

We use an optimal solver, OPSAT [42], to enumerate locally the optimal candidate solutions of each node in a best-first order. From this result, we determine what the pairwise solutions are by comparing each pair of candidate solution coming from each node. In Figure 4-7, we see that a pair structure can ask one of its nodes to send a next optimal solution. A node contains a basic subproblem and can also store conflicts gathered from communications with its direct neighbors as explained later in Subsection 4.3.5 about conflict handling. If a node that is asked for its next solution has already enumerated all its candidates, it returns $\perp$ to the pair structure. A pair of subproblems leads to a valid pairwise solution if the shared variables have the same value in both nodes. The cost of a pairwise solution is the sum of the costs of both

subproblem solutions.

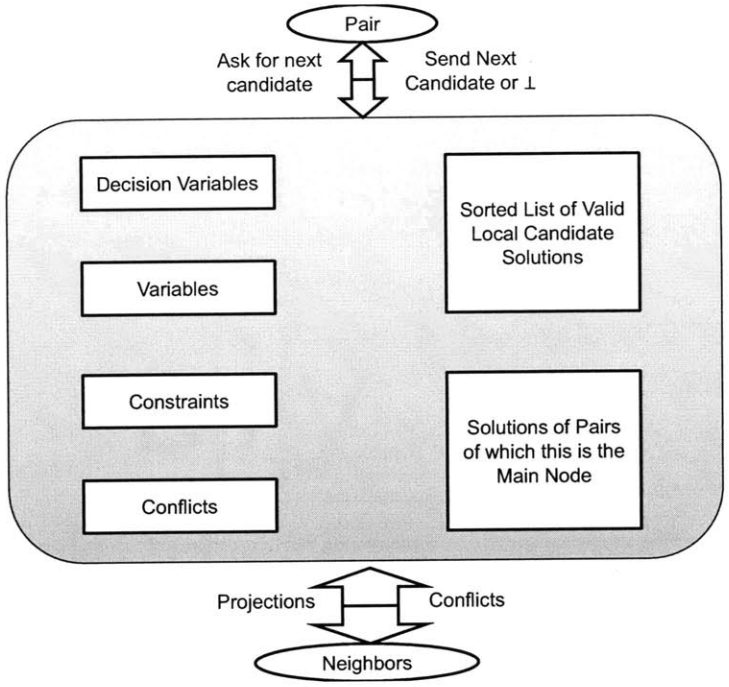


Figure 4-7: Node Communication

Example 4 (Candidate Solution Matching)

For instance, on Figure 4-8, the candidate solutions to a pair of subproblems are matched and joined together when they are consistent with each other. In the first example, the shared variables are b and c, and the projection of each candidate on the scope of the neighbor node is $\{b = Nominal, c = ON\}$. The cost of the two subproblem solution $\{a = OFF, b = Nominal, c = ON, d = Closed\}$ is the sum of the costs of both solutions. In the second example, we show how elements of two solution lists can be matched when they are consistent. □

One way to enumerate pairwise solutions is to enumerate all candidates in both subproblems, then produce all possible pairwise solutions by matching valid pairs of candidates. This method is very expensive and as Sachenbacher and Williams, we want to limit the number of candidate to compute, and therefore, we expand on

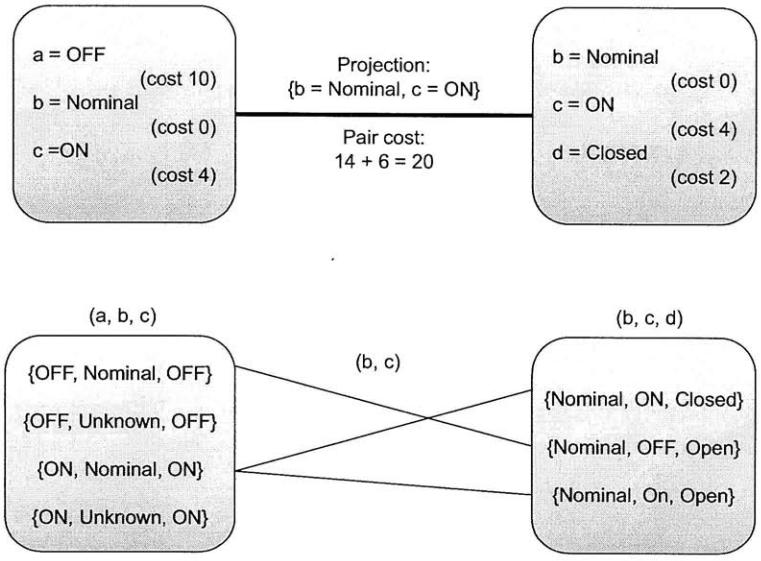


Figure 4-8: Valid Pair Solutions

demand the candidate list of each subproblem in order to generate optimal pairwise solutions [37]. At each iteration, one of the nodes of the pair can be asked to provide its next solution, until the pair can determine that it has effectively generated the i^{th} pairwise solution. This search is inspired by the A* algorithm. A heuristic determines which node to expand and when to stop the search, as presented in Section 4.3.3, like in the method presented by Sachenbacher and Williams [37].

By induction, we can consider that the reasoning presented in this section for a pair of nodes is also valid for a pair of subproblems.

4.3.3 Search Bound and Termination Condition

As we demonstrated in Subsection 4.3.2, a new pairwise valid solution is computed every time there is a matching between candidate solutions to the two subproblems of the pair. Let us suppose that the best solution to both subproblems match one another. This means that a pairwise optimal solution can be easily computed. This solution is composed of the union of the best candidate partial solutions of each subproblem. However, if the two best subproblem solutions do not match, the optimality criterion becomes non-trivial. Indeed, the second-best global solution is not necessarily the union of the best globally-feasible candidate partial solution of one problem and the second-best globally-feasible candidate partial solution of the other.

The same question is raised when we want to compute the i^{th} solution of a pair: even if i pairwise solutions have already been generated, we need to know if, in the future, a new solution can outrank one of the first i pairwise solutions or if the current i^{th} solution of the list is indeed the ith optimal pairwise solution.

We use an A*-like algorithm to prove when to stop the enumeration of new pairwise-valid solutions.

For clarity, let us name the two subproblems of the pair respectively $SubProblem1$ and $SubProblem2$ and the global problem $Pair$; we also suppose that one node, say the one containing $SubProblem1$, stores the pairwise solutions. While computing on-demand then matching partial solution candidates for $SubProblem1$ and $SubProblem2$, we produce a sorted list of solutions that are globally acceptable for

the initial constraint satisfaction problem *Pair*. The difficulty here is to determine when one of these solutions is at its final position in the list. In other words, we want to make sure that none of the future solutions will outrank a certain element of the list.

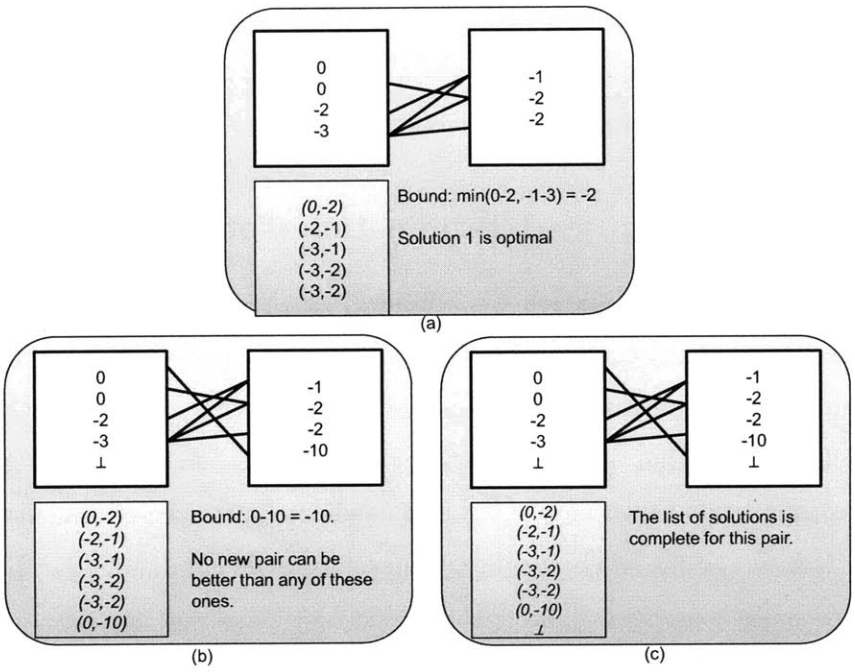


Figure 4-9: Best-First Enumeration Diagram

Example 5 (Best-First Enumeration)

For instance, in Figure 4-9, we represent a pair of nodes, and in each node we show the list of the costs of each candidate solution. The same pair is represented at 3 different instants. We represented the cases when candidates match one another (to create a pairwise solution) inside gray squares by connecting them with a line.

In the first case, we can see that any potentially new pairwise solution would not have a better cost than -2, therefore new pairwise solutions can appear between the current first and second solutions whose respective costs are -2 and -3.

On the contrary, in the second case where our bound indicates that no new solution

can have a better cost than -10, all the current pairwise solutions are at their final place, so we have computed the six best solutions of the pair.

In the last case, we see that the list of pairwise solutions is complete thanks to the $\perp$ symbol at the end. □

Algorithm 5 presents the condition under which a pair can determine if one of its nodes needs to be expanded in order to ensure that the beginning of the ordering of the pairwise candidates is definitive until a certain rank. This is the termination condition for the enumeration algorithm.

The procedure returns $\top$ if, and only if the search is not over in the case we need the $finalOrderingRank$ best pairwise solutions. To get this termination condition, we consider that expanding a node i cannot lead to a solution with a cost better than the one given by the matching (assuming it exists) between i's worst candidate solution and $j = i$'s optimal candidate solution. Therefore, if the $finalOrderingRank^{th}$ element of the pairwise solution list cannot be improved by computing a new candidate in either of the nodes, we can state that the search for the $finalOrderingRank^{th}$ pairwise solution is over. This condition determines when to stop the pairwise enumeration algorithm presented next in Subsection 4.3.4.

Algorithm 5 Best-First Enumeration Non-optimality Condition

```
1: procedure CONDITION(Pair)
2:     testCost ← cost(solutions[finalOrderingRank])
3:     heuristic1 ←
   cost(first(candidateSolutions1) + lastNon⊥(candidateSolutions2))
4:     heuristic2 ←
   cost(first(candidateSolutions2) + lastNon⊥(candidateSolutions1))
5:     heuristic ← best(heuristic1, heuristic2)
6:     if testCost ≽ heuristic) then
7:         return ⊥
8:     else
9:         return ⊤
10:    end if
11: end procedure
```

We have now defined the main tools and concepts we use to perform distributed

constraint solving. The next subsection presents the core algorithm for distributed, best-first solution enumeration.

4.3.4 Best-First Enumeration Overview

This Subsection presents the main enumeration algorithm for our distributed solver. It takes `Pair` of subproblems as an argument, and returns the next optimal solution. When applied to the root pair (containing all the nodes), the algorithm computes in a distributed manner and returns the next globally optimal solution to the original OCSP. The algorithm is recursive on the structure of the binary tree described in Subsection 4.3.1.

In Algorithm 6, the single argument *Pair* contains *solutions* (the list of solutions of the pair-level problem that have been computed so far), $candidateSolutions1$ and $candidateSolutions2$, $SubProblem1$ and $SubProblem2$, and $finalOrderingRank$. The lists of solutions are all sorted in a cost-wise best-first order. We also suppose we know that the $finalOrderingRank$ first candidates in *solutions* are in their final position. The algorithm returns the next-best solution, or an empty set if it does not exist. An invariant of this algorithm is that *solutions* always contains all the possible matchings that can be obtained from pairs of candidates respectively in $candidateSolutions1$ and $candidateSolutions2$.

Every time the algorithm decides to expand one subproblem, the most promising subproblem of the two is chosen, in the sense that its current worst candidate solution is better than the worst solution of its partner. Example 6 below runs step by step through the best-first enumeration algorithm.

The optimality of this algorithm is ensured by the heuristic presented in the previous section. The proof of optimality derives from the nature of the A* algorithm. Moreover, as Sachenbacher and Williams [37], we expand on-demand a minimal number of nodes during the search.

Example 6 (Best-First Enumeration Algorithm)
We run step by step the best-first enumeration algorithm through the example pre-

Algorithm 6 Pair-Level Best-First Enumeration 1/2

```
1: procedure NEXTSOLUTION(Pair) ▷ Indexing starts at 0 ▷ We are looking for the finalOrderingRank^th solution.
2:     if solutions ≠ ∅&last(solutions) = ⊥ then ▷ If all solutions are already computed,
3:         if length(solutions) = finalOrderingRank then
4:             return ⊥ ▷ All solutions were already enumerated,
5:         else ▷ Or returns the next solution in the list.
6:             finalOrderingRank ← finalOrderingRank + 1
7:             return solutions[finalOrderingRank]
8:         end if
9:     end if
10:    pb1 ← SubProblem1 pb2 ← SubProblem2 ▷ pb1 stores the node that will be expanded;
11:    cs1 ← candidateSolutions1 cs2 ← candidateSolutions2
12:    while #solutions ≤ finalOrderingRank do
13:        if last(cs1) ≤ last(cs2) then
14:            swap(pb1, pb2) swap(cs1, cs2) ▷ When pb2 is more promising, pb1 and pb2 are swapped.
15:        end if
16:        if last(cs1) ≠ ⊥ then
17:            candidate ← nextSolution(pb1)
18:        else
19:            solutions ← solutions ∪ {⊥}
20:            finalOrderingRank ← finalOrderingRank + 1
21:            return ⊥
22:        end if
23:        newSolutions ← cs2 × candidate ▷ newSolutions contains the set of new solutions produced by matching the new candidate against the candidates of the other subproblem.
24:        solutions ← insert(newSolutions, solutions)
25:    end while
```

Algorithm 7 Pair-Level Best-First Enumeration 2/2

```
26:     while condition(Pair) do
27:         if last(cs1) ≤ last(cs2)) then
28:             swap(pb1, pb2) swap(cs1, cs2)
29:         end if
30:         if last(cs1) ≠ ⊥ then
31:             candidate ← nextSolution(pb1)
32:         else
33:             solutions ← solutions ∪ {⊥}
34:             finalOrderingRank ← finalOrderingRank + 1
35:             return ⊥
36:         end if
37:         newSolutions ← cross(cs2, candidate)
38:         solutions ← insert(newSolutions, solutions)
39:     end while
40:     finalOrderingRank ← finalOrderingRank + 1
41:     return solutions[finalOrderingRank − 1]
42: end procedure
```

sented in Figure 4-10. The figure represents a pair of subproblems, and the costs of all the solutions. Let us call both subproblems A and B. We show in the steps below in which order solutions will be enumerated in each of these subproblems. Pairwise-consistent subproblems are connected by an edge.

- We compute the optimal solution in A and in B in order to have non-empty lists of solutions.
- The solutions do not match, so we expand the most promising, the one in A ($100 > 60$).
- We now have 2 solutions in A and 1 solution in B.
- A is still the most promising subproblem, we compute its next optimal solution, whose cost is 50.
- The latest computed solution in A matches one solution in B; one pairwise solution ($cost = 110$) is generated.

- This pairwise solution is optimal, but we cannot prove it yet: by enumerating the next solution in B, we can expect a pairwise solutions whose cost is equal to $100 + 60 = 160$.
- We expand the list in B, that contains 2 solutions now.
- We have a tie (50 *vs.* 50), so we can choose to expand A for instance, which produces a local solution with cost 40.
- We expand B, the cost of its worst solution is now 10, which proves that the first pairwise solution is indeed optimal.
- We then expand A, a new pairwise solution is computed, its cost is 30.
- next, we expand A, and receive the solution $\perp$: we have enumerated all the solutions of A, and now we can only enumerate solutions of B.
- The next solution in B produces a pairwise solution ($cost = 50$), that is better than the previous pairwise solution ($cost = 30$). However, we cannot prove yet that these pairwise solutions are the next-best solutions ($bound = 110$).
- When we expand the next solutions (costs 5 and 2), no new solutions are computed.
- Finally, B outputs $\perp$, all the solutions in both subproblems have been enumerated. We now know that the three computed pairwise solutions are optimal. Moreover, we know that there are no other pairwise solutions. □

This concludes the presentation of the main algorithm of our distributed optimal solver. This method may be extended by adding conflict handling capabilities.

4.3.5 Simple Conflict Handling

To speed-up the generation of solutions for the tree above, we can add some conflict handling to the search algorithm. A conflict follows from the idea of no-good memorization [38]. A conflict is used to eliminate inconsistent candidates of a node early,

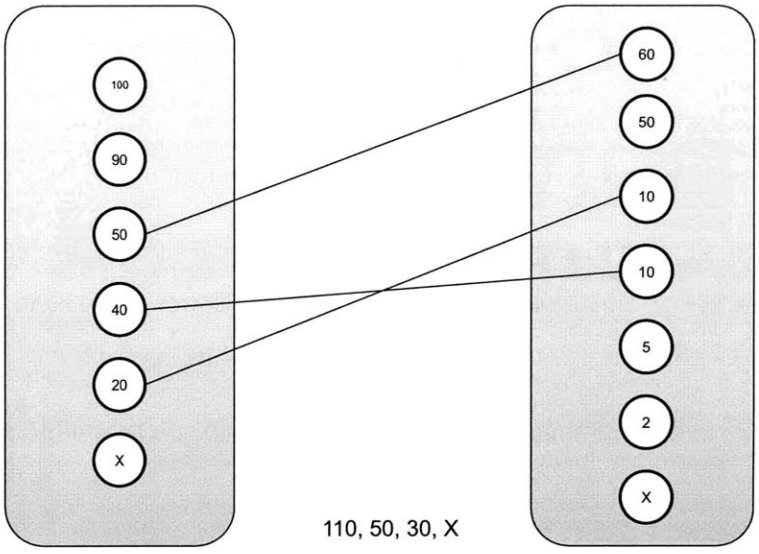


Figure 4-10: Pairwise Solution Matching Example: Computation of Costs

in order to not enumerate them. For example, the solver may deduce that a pair of variables (A, B) can never be assigned a pair of values (a, b), then integrate a new constraint $\not(A = a \ \& \ B = b)$ into the constraint set.

Every time a new candidate solution is computed for a node, its projection is tested for consistency against the neighbors of this node. In other words, each neighbor checks if the projection of the candidate solution on their scope is inconsistent with their local constraints and previous conflicts.

Hence, we try to limit the number of unnecessary candidate solution enumerations by direct consistency checks. The overhead of this additional test which potentially eliminates useless candidates is expected to be smaller than the one of fruitless optimal searches involving partial solutions that have no chance to be involved in any global solution. As a matter of fact, these consistency test are not optimization problems, so they do not require to generate any candidate, therefore they are performed quickly. Moreover, they can be run in parallel in each neighbor.

The implementation of the solver enumerating the local candidate solutions accepts an incremental addition of constraints (Example 7). Furthermore, it is able to check for consistency of a theory made of a set of constraints very quickly without needing to instantiate the actual value of the variables. We are taking advantage of these features in the conflict-assisted search. As soon as a local candidate solution is computed in a node, each of the neighbor nodes receives a projection of this candidate assignment on the respective shared variables. As a result, each of these neighbors can check if this hypothesis is consistent with their current constraints.

Example 7 (Simple Conflict Checking Method)
Figure 4-11 presents the different stages of the conflict checking method, from the point of view of a node.

First, a node is asked for its next solution. It computes its next candidate then broadcasts to all its neighbors the projections of this solution. Then, two cases can occur. In one case, the projections are consistent with the constraints of all the neighbors. Then, the candidate is added to the list of solutions. In the other case, one of the neighbors signals an inconsistency. Then, the node adds a new conflict to

its set of constraints. This conflict is the negation of the projection it had sent to the neighbor that signaled the inconsistency. □

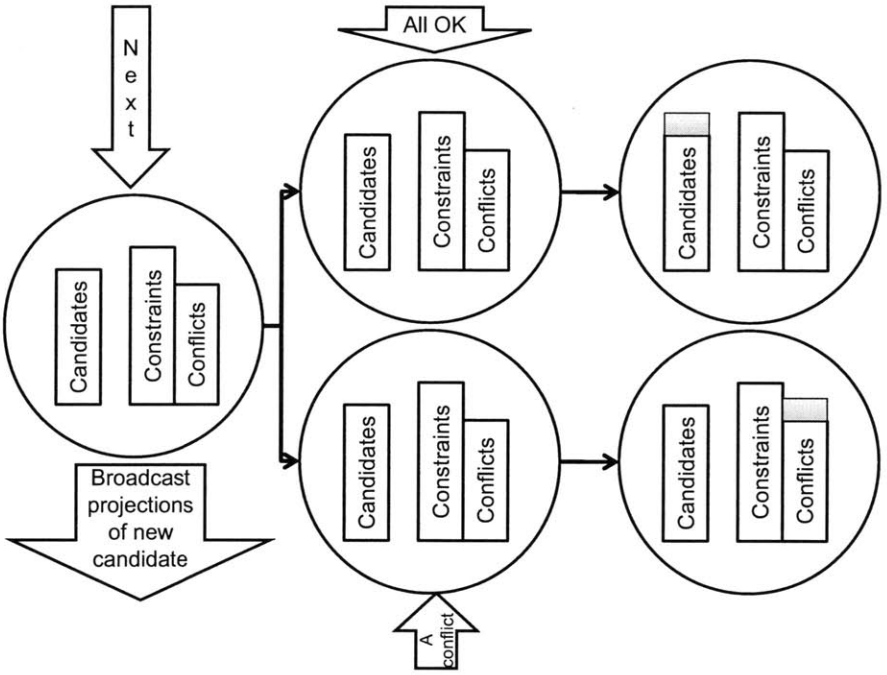


Figure 4-11: Simple Conflict Checking

To that extent, the nodes clone their current problem and add a series of constraints corresponding to the series of assignments. This duplicated problem is then solved as a satisfaction problem instead of an optimization problem which is much faster. If one of these neighbors signals an inconsistency, the current node knows that it should never enumerate a solution that would lead to the projection involved. Hence, it can add a conflict to its local theory, in the form of a new constraint denying the conjunction of variable assignments of the projection; this actual constraint is a single clause with negative literals negating each of the assignments of the incriminated projection.

Example 8 (Simple Conflict Checking)

Let N_1 and N_2 be two neighbor nodes, sharing variables b, and c, such that:

$Scope(N_1) = \{a, b, c\}$ and $Scope(N_2) = \{b, c, d\}$.

Let us assume that the problem in N_1 generates the following candidate: $\{a = 1, b = 2, c = 3\}$. To check if this solution is in direct conflict with the theory of the problem in N_2, N_1 sends N_2 the following projection: $\{b = 2, c = 3\}$.

If N_2 detects a contradiction of this projection with its constraints, it notifies N_1 which adds a *conflict* to its constraints. This conflict is a new constraint negating the projection sent to N_2: $Not(\{b = 2, c = 3\})$, also written with the clause: $(Not(b = 2) \| Not(c = 3))$.

Note that we do not need to add the conflict to N_2, as it is already implied by its set of constraints. □

This method only produces local conflicts that involve variables shared between pairs of nodes. Hence, these conflicts are not exhaustive (Algorithm 8). Contrary to a centralized search algorithm, here we only detect impossible assignment situations that occur for variables next to each other in the problem decomposition; the ordering of the variable is fixed in contrast to a conflict-directed A* [33, 42], or dynamic backtracking [12]. Nonetheless it is possible to extend the scope of the eliminations further than the node level, but a full arc-consistency check would imply explicitly assigning values in the neighbor nodes and propagating them until a leaf is reached, inside a backtracking loop. We would lose all the advantage of simple consistency checks.

In Algorithm 8 presenting conflict-directed enumeration, new candidates for the the problem in *Node* are generated and tested until they are not in contradiction with the *neighbors* of *Node*. First, at line 4, we check if all solutions have already been enumerated. Then the loop at line 8 continues the search until a plausible candidate is found, or until the last candidate has been enumerated. Depending on this outcome, the test at line 24 will either add the new candidate to the list *candidateSolutions*, or stop the search and add $\perp$ at the end of this list to signal that this node has been fully searched.

In order to eliminate candidates in conflicts with neighbors, every time a conflict is detected, the negation of the projection of the current candidate on the incriminated

neighbor is added to the *Node* theory line 19 in the form of a conflict. Then at each new enumeration, this new hypothesis can be taken into account by the local instance of OPSAT's conflict-directed A* search.

We have now completed the description of our distributed best-first solution enumeration algorithm. The reader may argue that other more direct methods can solve OCSP's in a distributed manner. For instance, one may choose to explore different assignments to a variable in parallel on different clusters. In the next section, we justify our choice not to choose this approach.

4.4 Difficulties of Exploring Different Variable Assignments in Parallel

The distributed search method proposed in the previous sections of this chapter uses concepts such as constraint graphs and tree-decomposition. One can wonder why we did not choose to naturally distribute a standard backtracking method by exploring several backtracking branches in parallel.

Simple backtracking is a very direct way to solve constraint satisfaction problems. One way to decompose a backtracking procedure is to evaluate in parallel different feasible assignments to a variable. If these sub-problems can be evaluated independently, this potentially leads to as many problem instances as the number of possible partial assignments. This multiplication of instances prevents this kind of algorithm from being scalable. Even if the method works for all problem instances, in practice, the complexity explosion is not limited to extreme cases. So if it is possible to use a distributed parallel exploration to solve very particular problem instances, to our knowledge, there does not seem to be an easy way to guess a priori the structure of the search tree of sub-problems. This complexity issue makes a parallel exploration method impractical without any efficiency improvements.

Moreover, in a tree of sub-problems representing potential assignments, many pieces are repeated in the different branches. Indeed, as variables receive candidate

Algorithm 8 Conflict-Handling Algorithm

```
1: procedure NEXTCANDIDATE(Node)
2:     problem ← problem(Node)
3:     candidateSolutions ← candidateSolutions(Node)
4:     if candidateSolutions ≠ ∅ & last(candidateSolutions) = ⊥ then
5:         return ⊥
6:     end if
7:     solutionFound ← ⊥
8:     while ⊤ do
9:         candidate ← solveNext(problem)      ▷ The current node generates a new candidate
10:        if candidate = ⊥ then               ▷ There are no more possible candidates
11:            solutionFound ← ⊤
12:        else
13:            solutionFound ← ⊤
14:            for neighbor ∈ neighbors(Node) do      ▷ Test for all neighbors, but stops if a conflict is found
15:                sharedVariables ← scope(Node) ∪ scope(neighbor)
16:                projection ← project(candidate, sharedVariables)
17:                if ¬consistent(problem(neighbor), projection) then      ▷ Test consistency against neighbors' theories
18:                    solutionFound ← ⊥
19:                    addConstraint(problem, ¬projection)      ▷ A conflict is added
20:                    break
21:                end if
22:            end for
23:        end if
24:        if solutionFound = ⊤ then
25:            candidateSolutions ← candidateSolutions ∪ {candidate}
26:            return last(candidateSolutions)      ▷ Returns the next candidate that is not in conflict with the neighbors
27:        end if
28:    end while
29: end procedure
```

values in a certain order in a backtracking loop, some variables can be assigned and unassigned with the same values many times during the search. A solution to this inefficiency could be to memorize some intermediate results. However this would require a central storage to accumulate and search information inside this knowledge base. Indeed, conflicts influencing the assignment to a variable need to be stored locally with respect to this variable. And if there are several instances of a given variable in the search tree, this means that we need the intermediate results to be stored so that they can be easily accessed *and* concurrently modified from each instance. Hence, a relevant distribution process cannot be achieved this way if we require search to be decentralized.

For instance, let us consider a problem with three variables (a, b, c), each of them having boolean domains. Let us put the following costs on assignments: $\top$ costs 0 and $\bot$ costs 1, and consider that we want to minimize the total sum of costs, under the constraint $(b \oplus c)$. With a parallel assignment method, one can test simultaneously on two clusters the assumptions $(a = \top)$ and $(a = \bot)$. We notice then that in both cases, the subproblems are identical: they contain only one constraint $(b \oplus c)$ and two variables b and c. So there is no use to repeating the same computation in both clusters. Though, the only way to avoid this duplication is to share a common memory between both clusters, which is not acceptable in our case.

The previous paragraphs of this section showed how impractical a parallel exploration of backtracking branches is when processor communication is limited. On the contrary, our method based on dual constraint graph clustering and best-first solution enumeration does not suffer the same limitations in this context.

4.5 Summary

To summarize, after a tree-clustering and a preprocessing stage on the initial problem, we are able to enumerate the best solutions with local message passing and optional conflict storing. The correctness of the solution is due to the tree structure of the decomposition that ensures that global consistency is implied by local consistency.

The optimality of the first best solutions computed is proven by the A*-like heuristic. This heuristic also minimizes the number of local candidate solutions that are computed on demand. We hence have a distributed algorithm to compute the K optimal solutions of an optimal hard-constraint satisfaction problem with costs on decision variable assignments.

Now that we have presented how to decompose a constraint satisfaction program into a tree of subproblems and solve it in a decentralized manner, let us show how to model systems that can be monitored with the technique above. We will use a probabilistic automaton-based representation that can be framed to an optimal hard-constraint satisfaction problem compatible with the aforementioned decomposition method.

Chapter 5

Modeling and Estimation Problem Framing

Many complex systems such as robots or aerospace, land or naval vehicles, consist of a mix of hardware and software components. These components are subject to unexpected malfunction and unpredictable behaviors. To ensure the robustness and the fault tolerance of complex systems, we need models that incorporate the risk of failure. When a sensor failure or an actuator malfunction is not detected properly by a monitoring system, a catastrophe may occur, particularly in critical systems.

In model-based diagnosis, mode estimation can be used to determine the failure states of software or hardware system components. Several modeling languages for encoding the behavior of a system have been introduced in prior work, as well as methods to estimate the system evolution over time.

In this chapter, we will recall prior work methods for system modeling and estimation, and present the solution we propose in order to allow a distributed, probabilistic mode estimation. The main innovations concern model representation based on previously introduced Probabilistic Hierarchical Constraint Automata, the mode estimation problem formulation, and a method to efficiently update the diagnosis at every time step (belief state update).

5.1 Model Representation

This section presents the modeling language we use in this thesis to represent a plant model, Probabilistic Hierarchical Constraint Automata (PHCA). The formal definition of a plant model was given in Chapter 3. A plant model is one of the main inputs to a mode estimation problem. We start by presenting previous work in the field of model representation, and justify our choice to use PHCA in this thesis. We then state the hypotheses that systems must verify in order for our model to be valid. Then, after defining formally the modeling language we use, we describe our innovations in terms of model representation.

5.1.1 Prior Work and Requirements

The need for robustness while monitoring model-based complex systems imposes some properties on the model. Accurate mode estimation requires an accurate model that can represent complex behaviors of hardware [9, 16, 12], software [7, 25, 26], or both [29]. While the plant model example that we are presenting only contains hardware components, we will consider in this thesis that we need to be able to represent mixed hardware and software systems in order to remain more general.

We distinguish two approaches to modeling with respect to the estimation method: consistency-based and probabilistic models. Some model-based monitoring techniques rely on consistency checks between a model representing the admitted evolutions of the system, and observations and measurements. In this case, any situation that is not in contradiction with the model is considered possible. This is for instance the case of the consistency-based estimation method presented in [12] by Dressler and Struss. In their work, faulty behaviors are detected by testing if the description of the behavior of faulty modes and the constraints implied by the observations are consistent. This approach is not adequate for the plant model that we are studying in this thesis. In the electric system that we presented in Chapter 3, the power distribution chain from one source to its destination involves several components: one generator, several relays, and one power panel. So a failure in the supply may involve one or several

components, and single failures are more likely than multiple failures in general. Yet, consistency-based methods would not sort the various failure explanations according to their likelihood, which does not allow to efficiently troubleshoot which components are the most likely to be responsible for a system malfunction.

In contrast to consistency-based diagnostic methods, other model-based diagnostic paradigms introduce probabilities in the model in order to compute the likelihood of each diagnosis. For instance, the Livingstone diagnostic engine [41] estimates the states of hardware components based on hidden Markov models that describe each component behavior as instances of nominal or failure modes. This hidden Markov model representation is encoded as Probabilistic Concurrent Constraint Automata (PCCA) [40, 10], which can represent hardware components.

The modeling method used in this thesis is based on Probabilistic Hierarchical Constraint Automata (PHCA) [39], which can be viewed as a superset of PCCA. Hierarchy was introduced in order to represent complex software behavior and handle situations were simultaneous transitions between states are triggered. Hierarchical automata had been previously used in embedded languages such as Statecharts [17] and Esterel [4]. Triggering several transitions simultaneously is very useful for software behavior modeling: a software execution may consist of several threads running at the same time. Hence, if we want to model a procedure launching several threads in parallel, we can do it by *transitioning* into the hierarchical automata that model each thread.

5.1.2 Hypotheses

The Probabilistic, Hierarchical Constraint Automaton model that we use is valid if the model verifies certain hypotheses about probabilistic independence. This section lists these conditions.

PHCA models are compact encodings of discrete Hidden Markov Models (HMMs). They take into account probabilistic system behavior, such as probabilistic component mode transitions. The objective of PHCA-based mode estimation is to estimate the evolution of component modes, which are hidden states of the system. The estimation

process is based on a series of observations and the PHCA model. PHCA models can accurately represent systems without memory (Markovian property) such that the probabilistic transitions of components are not correlated (failure independence). Both properties are explained in the following paragraphs.

1. We suppose that the evolution of the system we want to diagnose satisfies the Markov property. This means that the state of a system at a time step $n+1$ only depends on the state of the system at time n and of the commands sent between time steps n and $n+1$. This hypothesis implies that component transitions can be represented in terms of a stochastic choice, and constraint guards involving only variables from the current time step.

2. We suppose that probabilistic outcomes related to any subset of the components are independent. Consequently, the failure of one component does not affect the probability of transition into a failure state of an other component. This hypothesis implies that joint probabilities of events affecting two components are equal to the product of the probabilities of the events. This fundamental property is used in the estimation problem formulation in Section 5.2.

Now that we stated our hypotheses about the system behavior, we formally define Probabilistic, Hierarchical Constraint Automata in the following subsection.

5.1.3 Modeling Language

This subsection defines PHCA, the modeling language that we us in this thesis. A PHCA model is a set of hierarchical automata representing the possible probabilistic transitions between component modes. A PHCA model also represents the behavior of each mode with a set of constraints that hold when a mode is *marked*. The following definition was first introduced by Williams, Chung and Gupta [39]. We extend this definition and its semantics later in Subsection 5.1.4.

Systems are modeled as a set of individual PHCA components. A finite set of discrete modes of operation describing nominal or faulty behaviors is specified for

each component. These components communicate through shared variables, and transition guards and modes may be constrained by the modes of other components.

The following definition is a slightly updated version of the original one [39].

Definition 12 (Probabilistic, Hierarchical Constraint Automaton)
A PHCA is defined as a tuple

$$(L, V, C, \mathbf{P}_0, \mathbf{P}_T, \mathbf{P}_G, \mathbf{P}_O)$$

with the following properties.

- $L = L_p \sqcup L_c$ is the set of locations of the automaton, partitioned into primitive and composite locations. Primitive locations have no sub-functions, whereas composite locations are sub-automata. Locations of the automata represent modes of the system.

  A location can be either marked, or unmarked. A marking $m \in \mathcal{M}$ is a complete assignment to all locations; it defines the state of the system. A composite location is marked if and only if one of its children is marked. $\mathcal{M}$ is similar to the power set of L: $\mathcal{M} \equiv 2^L$.

- V is a set of finite-domain variables. Each variable $v \in V$ has a discrete, finite domain D_v.

  $$V = Observable\_Variables \cup Dependent\_Variables \cup Control\_Variables$$

  Let $\mathcal{P}$ be the set of partial assignments to variables in V.

- $C : L \rightarrow \mathcal{C}$ is a function that associates each location to a state constraint. In this thesis, constraints are expressed in first-order propositional logic.

  At each time step, if a location is marked, then its state constraint is enforced. If a location represents a given mode of the system, the associated state constraint describes the behavior of this mode.

- $\mathbf{P}_0$ is an optional probability distribution over allowed initial markings. This element describes the possible initial states of the system.

- $\mathbf{P}_T : L_p \times T \rightarrow [0,1]$ This element represents the probabilistic guarded transitions between different modes of the system. The actual PHCA implementations use a less abstract definition in terms of And-Or trees. At each time step, an automaton can transition from a marked primitive location to a set of locations with some probability: for each primitive location, $\mathbf{P}_T$ defines a probability distribution over transitions T.

- $\mathbf{P}_G : \mathcal{C} \times \mathcal{P} \times \mathcal{M} \rightarrow [0,1]$ defines *transition functions.* Transition functions give the probability that a transition guard condition in $\mathcal{C}$ is enabled given a partial assignment in $\mathcal{P}$ to control, observable, and dependent variables, and a marking in $\mathcal{M}$.

  In this thesis we make the following simplification: if a partial assignment and a marking exist such that the guard is satisfied, we state that the guard *holds* and the probability is 1; it is 0, otherwise. In other words, we only check if the state of the system is consistent with the transition guard.

- $\mathbf{P}_O : \mathcal{O} \times \mathcal{D} \times \mathcal{M} \rightarrow [0,1]$ defines *observation probability functions.* Given a state including a marking and a full assignment to decision variables, this function returns the probability that an observation is received. An observation is a full assignment to observable variables.

  In this thesis, we make the following simplification: observable variables are always clearly defined in the model, so that the value of observation functions is always 1 in case the observation is consistent with the state, 0 otherwise. □

For instance, Figure 3-2 presents a PHCA model for a relay including probabilistic guarded transitions between the three modes `Open`, `Closed` and `Unknown`.

Remark 2 (Specificities brought by Hierarchy)
In a PHCA, we represent components in a hierarchical way.

For instance, some components can contain different modules that may be enabled one after the other, or a computer program may run different threads. Thanks to the transition structure T, several locations can be marked at the same time inside a composite location. As location markings represent component modes, a component can be in several modes simultaneously. This means that PHCA can theoretically represent any possible combination of modes. In the case where only one location can be marked at any time step for a component, the hierarchical automaton may be replaced by a concurrent automaton in which modes are mutually exclusive. In Probabilistic, Concurrent Constraint Automata (PCCA) models introduced in previous work by Williams, Chung, and Gupta [39], each component mode is represented by one variable that can take several values. For instance, a mode variable in a relay model may have three values: `open`, `closed`, and `unknown`. □

The following subsection summarizes our innovations in the PHCA models, in terms of transitions, system initial state, and composite location initial state representations.

5.1.4 Innovation in the Modeling Capabilities

We introduced in the previous section a formal definition of PHCA. In this section, we define in more details our approach to transition modeling, and initial state representation. Next, in Section 5.2, which is about problem formulation, we explain how we encode these concepts into an OCSP, a form of hard-constraint optimization problem.

The formal PHCA definition in Subsection 5.1.3 introduces the set T of transition elements. In this thesis, we define transition elements $t \in T$ as sets of guarded branches. Hence, probabilistic transitions are represented by an Or-And tree: first, a probabilistic choice picks a set of branches, then each of these branches contains a guard and a destination marking. It allows to define complex transitions mixing probabilities and constraint guards.

Definition 13 (Transition Structure)

A probabilistic transition is a pair:

$$(origin, \{probability, transition\_element*\})$$

described as follows.

- *origin* is a primitive location. If it is marked at a given time step, then the transition is enabled and it may enforce a marking at the following time step.
- *probability* is the probability that a transition element is chosen if the transition is enabled. The choices are mutually exclusive; therefore, probabilities define a distribution and their sum is not greater than 1 (see PHCA definition).
- $transition\_element = branch*$ is a set of guarded branches. Each branch contains a constraint guard in a conjunctive normal form (set of disjunctive clauses), and a set of destination locations.

  If a branch is enabled and the constraint guard holds, then all the destination locations are marked at the following time step. This allows to mark multiple locations with a single transition. □

The previous definition makes clear the fact that several locations can be marked with a single transition. This was already the case in previous definitions of PHCA. It allows for instance the description of the `Always` construct, where a location marks itself and triggers a component at each time step [39]. However, the structure introduced below makes it easier to define other constructs, such as probabilistic intermittent Always loops for instance, where with a probability p, a dummy location marks itself, and with a probability $1-p$, the dummy location marks itself and a composite location.

We describe the possible initial states of a system by using structures that are similar to the one used for probabilistic transitions.

In the PHCA definition, we call the initial distribution over initial markings $\mathbf{P}_0$. This optional element describes the possible initial states of the system. For instance,

in a circuit, an initial state may be: `All the relays are open`. The structure describing the initial state probability distribution is analogous to the transition structure in Definition 13.

Definition 14 (Probability Distribution over Initial States)
A probability distribution over initial states is a set of pairs:

$$\{probability, initial\_marking*\}$$

described as follows.

- *probability* is the probability of an initial marking. The choices are mutually exclusive; therefore, probabilities define a distribution and their sum is not greater than 1.

- $initial\_marking \in \mathcal{M}$ is a marking.

  By analogy with the definition of transitions, we also represent an initial marking as a singleton: $\{branch\}$, with $branch = (\top, marking)$, an unguarded branch. Hence, an initial state description may be described as a probabilistic transition from a dummy state at $t = -1$, with unguarded branches. □

In the same way that we describe the initial state of a system, we may need to describe the initial behavior of a component when it is activated. If a component is described by a composite location, its initial behavior is described by a marking of the children of this location. Hence, when a transition triggers a composite location, it recursively marks the children locations corresponding to an initial state. To be general, we define a probability distribution on the initial (or default) markings of each composite location in the same way that we described the initial state distribution of a system in Definition 14.

Now that we have described and defined the plant model PHCA representation, we explain how we encode a PHCA-based mode estimation problem as an OCSP in Section 5.2.

5.2 Problem Formulation

In this section, we present our mapping of a PHCA *N-step* mode estimation problem into an OCSP. Previous work already introduced PHCA mode estimation using a soft-constraint problem framing [27]. We present a method that scales better on larger problems thanks to a more efficient problem encoding. We also propose a method to encode the structures, which is presented in Subsection 5.1.4.

The mode estimation problem framing consists of two main stages. As in previous work about mode estimation with transitions, we first encode the problem into an N-step trellis diagram that represents all possible state trajectories of the system. This encoding is in a standard OCSP format (5.2.2, 5.2.3). Next, we use this OCSP as the main input of the mode estimation process. A detailed problem framing is provided in Subsection 5.2.4.

5.2.1 Output of the Problem Formulation

As described in the problem statement in Chapter 3, the objective of mode estimation is to determine a set of K optimal state trajectory estimates of length N. Therefore, we need to formulate an OCSP representing N-step trajectories and extract K optimal solutions. However, if the final output consists of mode variable assignment estimates, the main part of the estimation problem determines which transitions are enabled at each time step. Therefore, we propose to cut the mode estimation problem framing into two sequential stages. The first one determines the outcomes of probabilistic choices, such as probabilistic transitions, or probabilistic initial states; then, a second phase computes which mode assignments correspond to these results. This decomposition makes problem framing and solving more efficient than previous methods, which mixed transition and mode assignment estimation in a single stage [40, 39, 27].

5.2.2 Trellis Diagram Expansion

This subsection highlights some properties of a trellis diagram. A trellis diagram presents all possible evolutions of a system state. A trellis is a directed, acyclic graph with nodes and edges. We are dealing in this thesis with an N-step trellis, therefore, it contains N sets of nodes representing modes. Each set (column) corresponds to one time step, and each node (row) corresponds to a system mode represented by a PHCA location. The edges represent possible transitions between nodes. Notice that each mode can be transitioned into from multiple previous states. Therefore, the probability that a mode is marked can be calculated by summing all transition probabilities that lead to this state. However, we are not looking here for the probability of each individual mode marking. We want to calculate the most-likely states (so the full markings with the highest probabilities). More precisely, we compute the most likely state trajectories over N steps, in order to deal with delayed symptoms [27].

A trellis diagram is an intermediate representation between a plant model and a constraint problem encoding. It is obtained by duplicating all the mode and internal variables for each time step. For instance, if an electric system contains a voltage, which is an observable variable, we use N variables to represent the evolution of the voltage measurement over the N step time window: $V^0, ..., V^{N-1}$. Variables representing transitions are only repeated $N-1$ times, as there are $N-1$ intervals between the N time steps. Constraints are also repeated, respectively N or $N-1$ times, if they are state constraints or guard constraints.

The next subsection details the mapping from a plant model to a trellis diagram.

5.2.3 Model to Constraint Problem Detailed Mapping

Given a PHCA model, this subsection explains how to encode an estimation problem into an OCSP. We have to expand all the variables, modes, and constraints as in the trellis diagram in order to encode the PHCA semantics into a standard optimal constraint framework.

In Chapter 3, we defined an optimal constraint satisfaction problem as a sextuplet:

$$(State\_Variables, Domains, Constraints, Decision\_Variables, Costs, Valuation)$$

and in this subsection, we specify these elements in the context of our mode estimation problem framing.

Valuation

Our objective is to extract the trajectories with the maximum probabilities. Therefore, decision variables are all associated with probabilistic choices, and $Valuation$ represents probabilities. We have two equivalent choices of valuations. We choose to express costs as logarithms of probabilities, therefore, we use the following definition:

$$Valuation = ([-\infty, 0], \leq, +, -\infty, 0)$$

that is valid, thanks to our assumption about probability independence (5.1.2).

Decision Variables

Decision variables correspond to probabilistic choices: probabilistic transitions, probabilistic default marking in a composite location, and probabilistic initial state. For each of the possible probabilistic choices, we associate a decision variable modeling the probability distribution on the possible choice outcomes. These three types of decision variables are described below in this subsection.

For instance, a decision variable is associated with each probabilistic transition choice. A transition choice with a high likelihood is modeled with a value which cost is lower than an unlikely choice.

Variables

In the same way that decision variables are duplicated, all the state variables, and their domains, are repeated and indexed by the N time steps: observable variables,

dependent variables, and control variables.

Locations and Markings

Indexed location variables are used to encode the possible markings. For a PHCA, a location variable has a binary domain: {marked, unmarked}. Location variables may be considered as boolean variables.

Location Constraints

Location constraints, also called state constraints, must be enforced when a location is marked. Therefore, for each time step, and each location, we store inside the OCSP *Constraints* element a set of constraints that encode: "if the location L is marked, then the location constraint of L c_L is enforced", or more formally, $L = \texttt{marked} \Rightarrow c_L$. This constraint must be expressed in a CNF form to be compatible with our solver.

Composite Marking

In the PHCA semantics, a composite location L0 is marked if, and only if one of its children L1, ... , Lk is marked. This constraint is expressed formally by the following equivalence: $L0 = \texttt{marked} \Leftrightarrow (\vee_{i=1..k} Lk = \texttt{marked})$. Now, we go into the details of constraints involving decision variables.

Transitions

Transition variables are the decision variables that determine which transition element of a probabilistic transition is chosen when the origin of the transition is marked. For instance, a transition T having 2 elements T_1 and T_2, both of probabilities 1/2, will lead to $N-1$ transition variables with 3 values: V_T^i, for $i \in 0..N-2$, and the respective domains of these variables are $\{v_{T_1}^i, v_{T_2}^i, v_{Noop}^i\}$. In this example, the assignment costs are respectively equal to $-log(2)$, $-log(2)$, and 0.

The value *Noop* represents the case when the transition is not enabled, *i.e.* when its origin is not marked. Therefore, for each transition, we create a constraint in

Constraints: $origin = \texttt{unmarked} \Leftrightarrow transition\_variable = Noop$. When the origin of the transition is marked, a transition value different from $Noop$ is chosen, and it enables all the corresponding transition branches. For each enabled branch, all the destination locations whose guard is consistent with the other variable assignments are marked at the following time step. Hence, we can represent the behavior of a transition element structure with an inter-temporal constraint of this form: $(T = v^i_{T_1} \wedge guard^i) \Rightarrow (destination^{i+1} = \texttt{marked})$.

Example 9 (Decision Variable Modeling a Probabilistic Choice)
Let `A` be a location with an outgoing transition `T` with two elements T_1 ($p = 0.9$) and T_2 ($p = 0.1$). To model this probabilistic choice, we introduce a decision variable v_T with weighted domain: $\{(v_{T_1} : log(0.9)),\ (v_{T_2} : log(0.1))\ (v_{T_0} = Noop : log(1) = 0)\}$. We add constraints that ensure that v_{T_1} or v_{T_2} are chosen if `A` is marked, and that v_{T_0} is chosen if `A` is unmarked. As a variable needs to be assigned a value, we had to introduce the additional $v_{T_0} = Noop$ value to the transition decision variable, to disable the transition when its origin `A` is not marked. □

Composite Transition Marking

The description of the transition constraint in the previous paragraph is valid when the destination is a primitive location. In case it is a composite location, it requires a slight change. As we cannot directly mark a composite location (we rather mark children of the composite location), we add a new variable for each time step and each composite location, called a *trigger variable*. A trigger variable can be marked or unmarked. When the trigger variable of a composite location is marked at a given time step, this means that one of the probabilistic initial markings of the composite location is enabled. Hence, when a transition reaches a composite location, we replace the name of the destination composite location by the one of its trigger variable inside the transition constraint.

By analogy to the method describing a probabilistic transition choice, for each composite location, one decision variable specifies which of the possible default initial markings have to be enabled when a composite location is triggered.

Example 10 (Transition Into a Composite Location)
Let A be a composite location that is transitioned into. A has two children locations a and b, and each of them is equally likely to be the start location of A. If A was a primitive location, we could have modeled the transition by setting a constraint that marks A at the next time step. However, here A is composite, therefore its marking depends on the status of its children locations. Consequently, we introduce a decision variable which purpose is to mark one of the locations a or b at the following time step. As soon as a or b are marked, A will be marked too. □

Initial Distribution

Finally, the last kind of decision variable is the one describing the choice of an initial state for the system. A single decision variable models the probabilistic choice of an initial state. Let us suppose that s initial states are defined, $S_1, ..., S_s$, with probabilities $P_1, ..., P_s$. We introduce a decision variable I of domain $\{i_1, ..., i_s\}$ and respective costs $log(P_1), ..., log(P_s)$. Then, the initial state distribution is modeled by the following set of constraints: $(I = i_j \ \wedge \ L \in S_j) \ \Rightarrow \ L^0 = \texttt{marked}$.

Now that we have described a mapping from a plant model to an OCSP, we propose a method to perform mode estimation by using this OCSP encoding.

5.2.4 Mode Estimation Problem Framing

In this section, we describe a process that allows us to enumerate the most likely state trajectories of a system by solving instances of OCSP's. We start by adding observations from the system into the problem. Then, we compute the most likely sets of enabled transitions. We use this result as a hypothesis for a new OCSP, the solutions of which are the most likely mode assignments. Hence, by solving the estimation problem with 2 consecutive constraint optimization instances, we decrease the number of decision variables as compared to previous approaches [40, 27].

Instead of computing which transitions are taken and which locations are marked at the same time, we separate these questions into two consecutive problem instances.

Otherwise, location decision variables would be mixed with all the other decision variables in the same OCSP instance. By separating the problems, we solve a first simpler OCSP without location decision variables, then determine the location markings mostly by consistency checks and constraint propagations, which is very fast.

Observations are simple assignments to observable variables. We assumed in Subsection 5.1.2 that when a variable is observed, its value is known with certainty. Therefore, we can encode an observation as a unary constraint of the type: $observable\_variable^i = value^i$. Moreover, as pointed out by Sachenbacher *et al.* [37], unary constraints do not have an influence on a tree-decomposition of an OCSP. Hence, we can even add observations online directly to a decomposed problem.

Our objective is to determine the most-likely state trajectories, by computing the optimal sets of markings. Yet, in the previous subsection, we did not include location variables in the set of decision variables, so they are not explicitly assigned if we solve the OCSP. In order to obtain these mode assignments, it is possible to transform location variables into decision variables by adding neutral (0) costs on both `marked` and `unmarked` values. However, as we use internally a solver based on a conflict-directed A* algorithm, the presence of many decision variables with non-discriminating costs makes best-first solution enumeration inefficient.

We propose to first estimate the initial state and the transition values by solving the OCSP instance described in Subsection 5.2.3. Then, we can use the solutions as hypotheses to an analogous OCSP where the decision variables are the mode variables (with neutral costs). For the second OCSP, we use the same constraints and variables as in the first OCSP instance, and the decision variables are the mode variables and an additional utility decision variable U used to encode the solutions of the first OCSP instance. We use the same principle as in the encoding of initial state distributions in Subsection 5.2.3. Each value of the utility variable corresponds to a solution E to the first OCSP, and the corresponding cost is the actual cost of E. Then we only need to add constraints enforcing the following set of conditions: $(U = u_i) \Longleftrightarrow (E\ is\ enforced)$.

As shown in Chapter 6, this 2-pass optimization procedure allows to scale on larger

problems than a method estimating transitions and mode assignments simultaneously.

This section presented a method to encode a mode estimation problem into a constraint problem. So far, the method we presented allows to estimate the most likely state trajectories of a system from time step 0 to time step $N-1$. The next section presents a method to move this time window by efficiently updating the state estimates at each new time step.

5.3 Belief Trajectory Update

In this section, we present a method that allows us to update the state trajectory estimation by moving the estimation time window from $[i, i+N-1]$ to $[i+1, i+N]$. We first describe a method to update the trajectories when new observations are received at a new time step. Finally, we explain why our method is robust to cases when estimation errors occur in previous time steps.

5.3.1 Belief State Update Algorithm

The main idea behind the belief state update method is similar to the one that we presented in Subsection 5.2.4, when dealing with the 2-pass optimization procedure.

When we shift the estimation time window by one time step, the simplest way to update the state trajectory estimate is to shift all previous observations one time step back, by discarding the oldest set of observations, and restart an N-step estimation procedure. We would like to keep as many previous estimates as possible in order to limit the computational complexity of the problem and avoid repeating some computation. Consequently, we integrate the previously computed trajectories (truncated by one time step) after shifting observations inside the OCSP, by using a utility decision variable in a way that is very similar to the definition of an initial state in Subsection 5.2.3. The only difference here is that instead of defining a probability distribution over "initial states" we define a probability distribution over "initial trajectories". As for other decision variables, we only keep the ones associated with the latest set of transitions.

5.3.2 Recovery from False Hypotheses

The method presented in Subsection 5.3.1 leverages old state estimates in order to extend state trajectories at each time step. If it happens that none of the previous K optimal solutions can be consistent with a new observation, we need to detect that failure and restart a search. We can adapt the method from Subsection 5.3.1 in order to automatically catch and correct previous diagnosis errors if delayed symptoms occur.

We propose to add a complementary value to the domain of the utility variable in the belief state update method. We set the cost associated with the complementary value is so that the sum of the probabilities corresponding to each cost of the utility variable is equal to one. Hence, this value stands for "everything else", all the possible solutions that were not associated with the other values of the utility variable. Hence, we always enumerate feasible solutions to the OCSP, and they always correspond to valid, most-likely, mode assignments.

This section completes our technical description of a mode estimation problem framing. The next chapter presents and analyses experimental results based on an implementation of a distributed, N-step, K-best mode estimation engine integrating the innovations from Chapters 4 and 5.

Chapter 6

Real-World Mode Estimation Implementation and Analysis

We presented in Chapter 5 how to frame a mode estimation problem into an optimal constraint satisfaction problem, and we described a distributed method to solve an OCSP in Chapter 4.3. In this chapter, we present implementation details and experimental results (Section 6.1), then we analyze the performance of our algorithms and propose some possible optimization (Section 6.2).

6.1 Experimental Results

We ran our distributed mode estimation algorithm on plant model examples, to test the capabilities and analyze the performance of our algorithms. In this section, we look at experimental results on different sets of data, with various choices for parameters N and K. We first give some details about our implementation in Subsection 6.1.1, then we illustrate our modeling method with a simple circuit with two components in Subsection 6.1.2. Next, we introduce our core example and benchmark power distribution plant model in Subsection 6.1.3, and we propose mode estimation simulations for both models in Subsection 6.1.4. Finally, we present experimental results in Subsection 6.1.5.

6.1.1 Implementation

We implemented the mode estimator presented in this thesis in Common Lisp, using several libraries.

The main input to the experiments is an XML representation of a PHCA plant model that is encoded into an N-step trellis diagram. Observations and commands can be fed to the diagnosis engine through an interface, in order to be integrated into the N-step trellis encoding of the estimation problem. The OCSP induced by the trellis diagram is decomposed with a standard implementation of a bucket-tree decomposition into a tree of connected subproblems [11]. The local optimal constraint solver used to enumerate the subproblems solutions is OPSAT, an implementation by Williams of the Conflict-Directed A* algorithm [42].

We implemented the estimation and optimal search capabilities presented in this thesis. Our trellis diagram and OCSP encodings implement all the PHCA modeling features presented in Chapter 5: guarded, probabilistic transitions, probabilistic initial state definition, and moving N-step time window. Additionally, our decomposed optimal constraint satisfaction algorithm can enumerate K optimal solutions of an OCSP in a best-first order by passing local solutions and conflicts between subproblems. Our implementation simulates a distributed search algorithm, however it is neither physically distributed, nor multithreaded. A true distributed algorithm would show improved processing time, as many stages of the search procedure can be parallelized.

Our PHCA plant models presented in Subsections 6.1.2 and 6.1.3 below were generated using the Reactive Model-Based Programming Language (RMPLj) [39]. The plant models are attached in the appendix chapters for reference. The following subsections present these models and associated mode estimation experimental results.

6.1.2 Simple Inverter Example

In this subsection, we present an implementation of a simple circuit example.

This example circuit contains two components A and B that can act as inverters,

wires, or can be broken (they have three modes: *wire*, *inverter*, and *unknown*). These components can transition between these modes, as presented in Subsection 3.1. The input of the circuit is connected to the input of Component A and the output of the circuit is connected to the output of Component B. The output of Component A is connected to the input of Component B. Additionally, each component can receive three kinds of commands: *cmd_invert*, *cmd_wire*, and *cmd_reset*. A detailed model is provided in Appendix A.

This circuit is simple enough so that all possible mode estimates can be quickly computed at every time steps, in order to test the correctness of our algorithm. In contrast, the large-scale plant model presented in the next subsection models a real-world scenario and tests the performance of our method.

6.1.3 Electrical Power Distribution System

The Electrical Power Distribution model presented earlier in Section 3.4 and Figure 3-1 is the main benchmark of our method. It contains thirty-six components with many connections. Therefore, the computational complexity of diagnosis problems on this plant does not allow us to perform online exhaustive search of mode estimates. This complexity justifies the trade-off to only enumerate a set of K most-likely estimates at each time step.

A detailed implementation of this plant model is provided in Appendix B.

Now that we have described our two test models, we introduce our mode estimation experiments.

6.1.4 Estimation Scenario

Each mode estimation benchmark consists of a sequence of diagnostic simulations. Each simulation compares:

- the mode estimates of our distributed, PHCA mode estimation method,
- the mode estimates computed by using a centralized, non-distributed optimal search algorithm with our PHCA framework, and

- the actual modes of a system.

We measure the influence of the parameters N and K on the processing time and the computational complexity of the generated optimal search problems, for both plant model. Each diagnosis simulation, or scenario consists of a sequence of observations and commands over a certain time window. The next section presents our experimental results.

6.1.5 Results

This section summarizes our experimental results. The following simulations were run on a dual-core 2.6 GHz, 1 MB Cache, 64-bit PC with 2 GB of RAM.

Simple Circuit

We run a set of simulations on the simple circuit example. The performance of the mode estimation algorithms are summarized below.

We first start by measuring computation time with two time steps.

K	N	Centralized (ms)	Decentralized (ms)
1	2	49	47
2	2	50	125
3	2	51	141
4	2	52	186
5*	2	52	187

(* When $K \geq 4$, 4 solutions are computed, as there are exactly 4 valid solutions.)

- Number of decision variables: 6.
- Number of state variables: 54.
- Number of state constraints: 152.
- Number of mode variables: 16.

First of all, both centralized and distributed OCSP solvers enumerate identical and valid solutions. For instance, in a simple scenario where all commands are `Reset` commands and the initial state is unknown. In this scenario, there are four possible trajectories containing: one contains only nominal modes, two contain single failures, and one contains a double failure. If K is greater than 3, all valid solutions are computed with both algorithms.

This set of simulations also proves the efficiency of the 2-pass optimization presented in Subsection 5.2.4. If we try to compute modes and transition values as solutions of a single OCSP, computation time soars by more than one order of magnitude, even on this simple example.

We also notice that the distributed algorithm uses more memory than the centralized version. For instance, computing all solutions requires 200 kB of memory in the centralized case, and 1.4 MB in the distributed case. Consequently, the distributed solver hits the CPU cache limit earlier than the centralized version, and this affects the computation speed. However, we could still accelerate the decentralized version of the optimal solver by actually distributing it on several machines.

In the case of this simple model, conflicts barely affect the computation time. The cardinality of the scope of conflict constraints is small in this example. Therefore, conflicts can be used here.

Here are the experimental results for $N = 4$ time steps.

K	N	Centralized (ms)	Decentralized (ms)
1	4	120	810
2	4	200	1010
3	4	250	3800
4	4	260	4100
62	4	900	6000

- Number of decision variables: 18.
- Number of state variables: 114.
- Number of state constraints: 330.

- Number of mode variables: 32.

All 62 solutions can be computed in reasonable time with both methods, and they are enumerated in the same order.

Large-Scale Power Distribution System

Now, we look into the large-scale experimental data.

K	N	Centralized (ms)	Decentralized (ms)
1	2	2116	2010
2	2	2600	68000

- Number of decision variables: 96.
- Number of state variables: 842.
- Number of state constraints: 4086.
- Number of mode variables: 264.

It is now impossible to compute all possible solutions, the problem is too complex.

Our problem framing allows us to encode a large mode estimation problem into an OCSP that can be solved in reasonable time with a centralized solver. Yet, in this example, the distributed algorithm is much slower than the centralized version. This discrepancy is partly due to the termination condition in the distributed algorithm (4.3.3): in this model, many components are identical and play the same role. Hence, the plant contains many symmetries, and there are often many equally-likely diagnoses explaining the same symptom. Hence, even if the algorithm could find 2 optimal solutions very quickly, it had to keep enumerating candidate solutions in the subproblems, in order to prove that these solutions were indeed globally optimal.

Now that we have presented our implementation and experiments, we will analyze our results and comment on the advantages and possible improvements of our approach in Section 6.2.

6.2 Analysis and Possible Optimizations

In this section, we analyze the experimental results from Section 6.1. We look into several topics: problem size and complexity issues, and the advantages and drawbacks of hierarchical modeling. Finally, we discuss how to choose the parameters N and K.

6.2.1 Problem Size and Complexity

In this subsection, we comment on the size of the mode estimation OCSP problems.

According to our experimental results in the previous section, the number of variables and constraints of an OCSP encoding grows linearly with N but the computation time required to solve the optimal problem grows exponentially, as it follows the size of the set of admissible trajectories. Moreover, the addition of a mode inside a component greatly increases the complexity of the constraint problem and the search time, partly because of the exponential nature of PHCA problem framings (see Subsection 6.2.2).

6.2.2 Discussion on Hierarchy

This thesis discusses key differences between the hierarchical PHCA modeling framework and the concurrent PCCA modeling framework. PCCA models are less expressive than PHCA models, however when they can model a system, they tend to lead to simpler and more efficient constraint problem encodings.

As mentioned in Chapter 5, some components may only be in one mode at a time. Yet, PHCA are general enough so that by default, they can represent a marking of any subset of the m modes. In order to enforce mutual exclusion, an additional state constraint may be added to the component, or an additional dummy mode variable with m possible values may be added to the model. Then, even if modes are mutually exclusive, the OCSP encoding the PHCA will have a search space of cardinality 2^m, whereas the OCSP encoding of an equivalent PCCA with a single mode variable will have a search space of cardinality m.

Consequently, a system with i components with an order of m modes each leads to an OCSP with a polynomial number of choices for a PCCA model ($\mathcal{O}(m^i)$), and an exponential number of choices for a PHCA model ($\mathcal{O}(2^{m*i})$). Therefore, it seems that a model mixing PCCA and PHCA representations is a good trade-off between expressiveness and complexity.

The set of systems that can be modeled by a PCCA representation is included inside the set of systems that can be modeled by a PHCA representation. Thus, it is worth examining which classes of subsystems cannot be factored as PCCA's. In a PCCA, a transition can mark at most one destination, and a component can be in at most one mode. Therefore, every time a subsystem must contain a transition into more than one node, the subsystem and its descendants cannot be modeled by a PCCA. Hence, every subsystem inside an `Always` construct cannot be modeled by a PCCA, because the double marking of the {`Always A`} structure "pushes a token" inside the composite automaton `A` at every time step.

6.2.3 Choice of N and K

The choices of the parameters N and K can be arbitrarily set in advance in order to test the performance of the algorithms. However, setting relevant values for N and K greatly depends on the nature of the problem and the observations. This subsection presents some insight on the choice of these parameters.

The choice of N is mainly a trade-off between the computational complexity of the estimation problem, and the ability to diagnose delayed symptoms accurately. Therefore, the size of the time window depends on the computing power, the complexity of the system and the tolerable latency in the diagnosis. The value of N may be determined empirically beforehand.

Unlike the parameter N, the parameter K can be adjusted during the mode estimation process. K determines the number of optimal trajectories that are computed at every time step. Hence, by setting K to an arbitrarily large value, it is theoretically possible to compute all the admissible trajectories that are consistent with the observations, including the extremely unlikely cases where all the components are

failing. Therefore, the choice of K directly determines the probability that the actual state trajectory of the system matches one of the computed trajectory estimates. The parameter K sets the tolerance to diagnosis error, and it determines when to stop a search.

As our optimal search method enumerates solutions in a best-first order, we may decide to abort the search either after a certain duration, or when a certain probability of success has been reached. For instance, one can decide to look for the optimal trajectories that span probability-wise 99% of the admissible trajectory set. As the cost of each solution of the search problem is directly related to the probability of the computed trajectory, the search algorithm can immediately determine when it has reached a given certainty threshold.

Another situation may occur when many equally likely trajectories are consistent with the observations, for instance when many components are identical and play the same role. In this case, one may not want to arbitrarily favor some solutions by stopping the search before all analogous state trajectories have been enumerated.

Now that we have presented and analyzed experimental results, we can conclude this thesis in Chapter 7.

Chapter 7

Conclusion and Future Work

This chapter presents several ideas for future expansions of this research and concludes this thesis. Section 7.1.1 discusses some possibilities for improvements in terms of system modeling and problem encoding, while Section 7.1.2 deals with extensions that may improve the accuracy of the representation of probabilistic behaviors. Furthermore, we recall in Section 7.1.3 that this thesis focused on discrete estimation and that the model-based monitoring paradigm can also be extended to hybrid estimation. Finally, we conclude our discussion and summarize our contributions in Section 7.2.

7.1 Future Work

We provide some ideas for expanding the capabilities of the research in this thesis. Some of them have been or are being explored by other members of the monitoring and verification communities.

7.1.1 Improvement on Modeling and Encoding

This thesis focused on PHCA-based mode estimation. PHCA models are powerful and expressive as they can represent stochastic behavior of software and hardware systems. However, this expressiveness comes with a price at computation time; PHCA

may offer a too general modeling framework when it comes to dealing with simple or very specialized embedded system monitoring. Lacking the hierarchical constructs of PHCA, PCCA representations can already deal with a wide range of discrete systems. As proposed in Chapter 6, both PHCA and PCCA representations may be mixed in order to limit problem complexity without compromising model expressiveness.

Other hierarchical automata representations such as the ones that are used in the verification community may also be used for monitoring purposes. For instance, Barros, Henrio and Madelaine used a hierarchical automata for verification of distributed hierarchical components [2]. Their model allows to replace a component or to apply some transformations on the system. Moreover, the method they present is applicable to asynchronous components, while PHCA was designed to deal with synchronous systems.

Another possible thrust of innovation concerns the design of a specific modeling language for distributed monitoring. The design of a modeling framework that can produces a distributed problem where the topology of the distributed problem matches the physical distribution of components is beyond the scope of this thesis. This topic was analyzed by Kurien, Koutsoukos and Zhao in [21].

7.1.2 Probabilistic Behavior Representation

The mode estimation capability presented in this thesis takes into account behavior uncertainty by integrating probabilities into the plant model. There are several ways to increase the accuracy of the representation of probabilistic behaviors.

In Chapter 5, we made the hypothesis that a constraint guard holds with probability 1 when it is consistent with a set of observations. With this hypothesis, mode estimation tends to overestimate the probability of component failure [24]. Martin *et al.* introduced the concept of observation functions and transition functions in order to estimate more accurately the probability of transitions [24]. This thesis did not explore the implementation issues related to the integration of observation and transition functions in a distributed estimation framework.

Another possible extension of our work concerns the possibility to modify the

probabilities in the model itself if some events occur. For instance, if a fire has been detected on an installation, the failure probability of components may be higher than in a normal situation.

7.1.3 Hybrid Continuous and Discrete Estimation

PHCA mode estimation deals with discrete, finite domain constraints, and discrete time steps. Systems such as hydraulic plants may require the addition of linear constraints. Mixed logic linear programming frameworks and their extensions may be used to frame distributed monitoring problems efficiently [20, 32].

7.2 Conclusion

In this thesis, we presented a framework for model-based, receding horizon, distributed mode estimation of discrete embedded systems. Our mode estimation employs optimal constraint solving to estimate the most-likely state trajectories of the system. Mode estimation is able to update its belief state at each new time step after receiving a new set of observations from the system. The distributed nature of our method aims at increasing the robustness of monitoring with respect to unexpected failures of computation and communication.

Our method is innovative in the fields of distributed, K-best, hard-constraint satisfaction problem solving. Moreover, it provides new ideas and insights on probabilistic automaton-based diagnosis problem framing and encoding, and proposes a way to correct erroneous estimations when very unlikely system failures occur. First, we introduced a scalable, hard-constraint optimal solver that is able to compute a set of optimal solutions in a distributed manner; it uses constraint graph clustering then alternates between local, centralized search, by exchanging results or conflict communication between clusters. Second, we proposed an algorithm to frame a PHCA plant model into an optimal, hard-constraint satisfaction problem. This algorithm can specify complex transitions and integrate assumptions about the initial state of the system we monitor. Third, we enabled efficient online estimation update by reusing

trajectory estimates performed at previous time steps.

We demonstrated the accuracy and scalability of our algorithm in simulation. We first presented our method on simple examples, then ran our algorithm on a larger-scale electric plant model. Our result on a simulated distributed algorithm showed that our mode estimation is able to compute a set of most-likely state trajectories accurately. Our solver proved to scale on large problems; however, the complexity induced by the general PHCA model may limit scalability.

We have discussed some future areas of research, and we believe that distributed, robust diagnosis is a valuable field of research, as it can increase the reliability and survivability of critical embedded systems.

Appendix A

Simple Inverter Model

RMPLj model of the simple circuit used in Chapter 6.1.

```
class Power {
        value up;
        value dn;}

class Command {
        value cmdpass;
        value cmdinvert;
        value cmdreset;}

class Mutex {
        value m0;
        value m1;
        value m2;}

class Comp {
        Power input;
        Power output;
        Command command;
        Mutex mutex;

        initial {pass}

        value pass = ((input == output) && (mutex == m1));
        value invert = ((not (input == output)) && (mutex == m2));
        value unknown = (mutex == m0);

        transition t1 pass => {
                invert with guard: (command == cmdinvert);
                pass with guard: (not (command == cmdinvert));
        } with probability: 0.99;
        transition t2 pass => {
                unknown;
        } with probability: 0.01;

        transition t3 invert => {
                pass with guard: (command == cmdpass);
                invert with guard: (not (command == cmdpass));
        } with probability: 0.99;
        transition t4 invert => {
                unknown;
        } with probability: 0.01;
```

```
        transition t5 unknown => {
                unknown with guard: (not (command == cmdreset));
                pass with guard: (command == cmdreset);
        } with probability: 1;}

class Main {
        Comp Comp1;
        Comp Comp2;

        Command cmd1;
        Command cmd2;
        Power input;
        Power output;
        Power middle;

        constraint = (
                (input == Comp1.input) &&
                (middle == Comp1.output) &&
                (middle == Comp2.input) &&
                (output == Comp2.output) &&
                (cmd1 == Comp1.command) &&
                (cmd2 == Comp2.command));

        Main() {
                Comp1 = new Comp();
                Comp2 = new Comp();}}
```

Appendix B

Power Distribution Plant Model

RMPLj model of the power distribution plant model used in Chapter 6.1.

```
class Power {
        value zero;
        value positive;}

class Command {
        value reset;
        value on;
        value off;
        value pass;
        value block;}

class PowerSupply {
        Power output1;
        Power output2;
        Command command;

        initial {power-off}

        value power-on = ((output1 == positive) && (output2 == positive));
        value power-off = ((output1 == zero) && (output2 == zero));
        value power-fail = True;

        transition t1 power-on => {
                power-off with guard: (command == off);
                power-on with guard: (not (command == off));
        } with probability: 0.99;
        transition t2 power-on => {
                power-fail;
        } with probability: 0.01;

        transition t3 power-off => {
                power-on with guard: (command == on);
                power-off with guard: (not (command == on));
        } with probability: 0.99;
        transition t4 power-off => {
                power-fail;
        } with probability: 0.01;

        transition t5 power-fail => {
                power-fail with guard: (not (command == reset));
                power-off with guard: (command == reset);
        } with probability: 1;}
```

```
class Relay {
        Power input;
        Power output;
        Command command;

        initial {relay-block}

        value relay-pass = (input == output);
        value relay-block = (output == zero);
        value relay-fail = ((input == output) || (output == zero));

        transition t1 relay-pass => {
                relay-block with guard: (command == block);
                relay-pass with guard: (not (command == block));
        } with probability: 0.95;
        transition t2 relay-pass => {
                relay-fail;
        } with probability: 0.05;

        transition t3 relay-block => {
                relay-pass with guard: (command == pass);
                relay-block with guard: (not (command == pass));
        } with probability: 0.95;
        transition t4 relay-block => {
                relay-fail;
        } with probability: 0.05;

        transition t5 relay-fail => {
                relay-fail with guard: (not (command == reset));
                relay-block with guard: (command == reset);
        } with probability: 1;}

class PowerPanel {
        Power input1;
        Power input2;
        Power output;
        Command command;

        value panel-nominal = ((((input1 == positive) || (input2 == positive)) &&
                                (output == positive)) || ((input1 == zero) &&
                                (input2 == zero) && (output == zero)));
        value panel-fail = (output == zero);

        initial {panel-nominal}

        transition t1 panel-nominal => {
                panel-nominal;
        } with probability: 0.96;
        transition t2 panel-nominal => {
                panel-fail;
        } with probability: 0.04;

        transition t3 panel-fail => {
                panel-fail with guard: (not (command == reset));
                panel-nominal with guard: (command == reset);
        } with probability: 1;}

class Dispatcher {
        Power input1;
        Power input2;
        Power output;

        value dispatcher = (((input1 == zero) && (input2 == zero) &&
                                (output == zero)) || (output == positive));

        initial {dispatcher}
```

```
        transition t dispatcher => {
                dispatcher;
        } with probability: 1;}

class Main {
        PowerSupply PS1;PowerSupply PS2;Dispatcher D1;  Dispatcher D2;
        Relay R01;      Relay R02;      Relay R03;      Relay R04;
        Relay R05;      Relay R06;      Relay R07;      Relay R08;
        Relay R09;      Relay R10;      Relay R11;      Relay R12;
        Relay R13;      Relay R14;      Relay R15;      Relay R16;
        Relay R17;      Relay R18;      Relay R19;      Relay R20;
        Relay R21;      Relay R22;      Relay R23;      Relay R24;
        PowerPanel PP1; PowerPanel PP2; PowerPanel PP3; PowerPanel PP4;
        PowerPanel PP5; PowerPanel PP6; PowerPanel PP7; PowerPanel PP8;

    Command cmd-PS1;    Command cmd-PS2;
    Command cmd-R01;    Command cmd-R02;
    Command cmd-R03;    Command cmd-R04;
    Command cmd-R05;    Command cmd-R06;
    Command cmd-R07;    Command cmd-R08;
    Command cmd-R09;    Command cmd-R10;
    Command cmd-R11;    Command cmd-R12;
    Command cmd-R13;    Command cmd-R14;
    Command cmd-R15;    Command cmd-R16;
    Command cmd-R17;    Command cmd-R18;
    Command cmd-R19;    Command cmd-R20;
    Command cmd-R21;    Command cmd-R22;
    Command cmd-R23;    Command cmd-R24;
    Command cmd-PP1;    Command cmd-PP2;
    Command cmd-PP3;    Command cmd-PP4;
    Command cmd-PP5;    Command cmd-PP6;
    Command cmd-PP7;    Command cmd-PP8;

    Power pwr-PP1;    Power pwr-PP2;
    Power pwr-PP3;    Power pwr-PP4;
    Power pwr-PP5;    Power pwr-PP6;
    Power pwr-PP7;    Power pwr-PP8;

    Power R0104;    Power R0203;

    Power PS1-out1;    Power PS2-out1;
    Power PS1-out2;    Power PS2-out2;
    Power R01-out;    Power R02-out;
    Power R03-out;    Power R04-out;
    Power R05-out;    Power R06-out;
    Power R07-out;    Power R08-out;
    Power R09-out;    Power R10-out;
    Power R11-out;    Power R12-out;
    Power R13-out;    Power R14-out;
    Power R15-out;    Power R16-out;
    Power R17-out;    Power R18-out;
    Power R19-out;    Power R20-out;
    Power R21-out;    Power R22-out;
    Power R23-out;    Power R24-out;
    Power PP1-in1;    Power PP2-in1;
    Power PP3-in1;    Power PP4-in1;
    Power PP5-in1;    Power PP6-in1;
    Power PP7-in1;    Power PP8-in1;
    Power PP1-in2;    Power PP2-in2;
    Power PP3-in2;    Power PP4-in2;
    Power PP5-in2;    Power PP6-in2;
    Power PP7-in2;    Power PP8-in2;

        constraint = (
                (PS1.output1 == R01.input) &&
                (PS1.output2 == R02.input) &&
                (PS2.output1 == R04.input) &&
                (PS2.output2 == R03.input) &&
```

```
(D1.input1 == R01.output) &&
(D2.input1 == R02.output) &&
(D1.input2 == R04.output) &&
(D2.input2 == R03.output) &&
(D1.output == R0104) &&
(D2.output == R0203) &&
(R0104 == R05.input) &&
(R0104 == R06.input) &&
(R0203 == R07.input) &&
(R0203 == R08.input) &&
(R05.output == R09.input) &&
(R05.output == R10.input) &&
(R05.output == R11.input) &&
(R05.output == R12.input) &&
(R06.output == R13.input) &&
(R06.output == R14.input) &&
(R06.output == R15.input) &&
(R06.output == R16.input) &&
(R08.output == R21.input) &&
(R08.output == R22.input) &&
(R08.output == R23.input) &&
(R08.output == R24.input) &&
(R07.output == R17.input) &&
(R07.output == R18.input) &&
(R07.output == R19.input) &&
(R07.output == R20.input) &&
(R09.output == PP2.input1) &&
(R10.output == PP1.input1) &&
(R11.output == PP4.input1) &&
(R12.output == PP3.input1) &&
(R13.output == PP6.input1) &&
(R14.output == PP5.input1) &&
(R15.output == PP8.input1) &&
(R16.output == PP7.input1) &&
(R17.output == PP2.input2) &&
(R18.output == PP1.input2) &&
(R19.output == PP4.input2) &&
(R20.output == PP3.input2) &&
(R21.output == PP6.input2) &&
(R22.output == PP5.input2) &&
(R23.output == PP8.input2) &&
(R24.output == PP7.input2) &&

(PP1.output == pwr-PP1) &&
(PP2.output == pwr-PP2) &&
(PP3.output == pwr-PP3) &&
(PP4.output == pwr-PP4) &&
(PP5.output == pwr-PP5) &&
(PP6.output == pwr-PP6) &&
(PP7.output == pwr-PP7) &&
(PP8.output == pwr-PP8) &&
(PS1.command == cmd-PS1) &&
(PS2.command == cmd-PS2) &&
(R01.command == cmd-R01) &&
(R02.command == cmd-R02) &&
(R03.command == cmd-R03) &&
(R04.command == cmd-R04) &&
(R05.command == cmd-R05) &&
(R06.command == cmd-R06) &&
(R07.command == cmd-R07) &&
(R08.command == cmd-R08) &&
(R09.command == cmd-R09) &&
(R10.command == cmd-R10) &&
(R11.command == cmd-R11) &&
(R12.command == cmd-R12) &&
(R13.command == cmd-R13) &&
(R14.command == cmd-R14) &&
(R15.command == cmd-R15) &&
```

```
        (R16.command == cmd-R16) &&
        (R17.command == cmd-R17) &&
        (R18.command == cmd-R18) &&
        (R19.command == cmd-R19) &&
        (R20.command == cmd-R20) &&
        (R21.command == cmd-R21) &&
        (R22.command == cmd-R22) &&
        (R23.command == cmd-R23) &&
        (R24.command == cmd-R24) &&
        (PP1.command == cmd-PP1) &&
        (PP2.command == cmd-PP2) &&
        (PP3.command == cmd-PP3) &&
        (PP4.command == cmd-PP4) &&
        (PP5.command == cmd-PP5) &&
        (PP6.command == cmd-PP6) &&
        (PP7.command == cmd-PP7) &&
        (PP8.command == cmd-PP8) &&

        (PS1.output1 == PS1-out1) &&
        (PS1.output2 == PS1-out2) &&
        (PS2.output1 == PS1-out1) &&
        (PS2.output2 == PS1-out2) &&
        (R01.output == R01-out) &&
        (R02.output == R02-out) &&
        (R03.output == R03-out) &&
        (R04.output == R04-out) &&
        (R05.output == R05-out) &&
        (R06.output == R06-out) &&
        (R07.output == R07-out) &&
        (R08.output == R08-out) &&
        (R09.output == R09-out) &&
        (R10.output == R10-out) &&
        (R11.output == R11-out) &&
        (R12.output == R12-out) &&
        (R13.output == R13-out) &&
        (R14.output == R14-out) &&
        (R15.output == R15-out) &&
        (R16.output == R16-out) &&
        (R17.output == R17-out) &&
        (R18.output == R18-out) &&
        (R19.output == R19-out) &&
        (R20.output == R20-out) &&
        (R21.output == R21-out) &&
        (R22.output == R22-out) &&
        (R23.output == R23-out) &&
        (R24.output == R24-out));

Main() {
        PS1 = new PowerSupply();          PS2 = new PowerSupply();
        D1 = new Dispatcher();            D2 = new Dispatcher();
        R01 = new Relay();                R02 = new Relay();
        R03 = new Relay();                R04 = new Relay();
        R05 = new Relay();                R06 = new Relay();
        R07 = new Relay();                R08 = new Relay();
        R09 = new Relay();                R10 = new Relay();
        R11 = new Relay();                R12 = new Relay();
        R13 = new Relay();                R14 = new Relay();
        R15 = new Relay();                R16 = new Relay();
        R17 = new Relay();                R18 = new Relay();
        R19 = new Relay();                R20 = new Relay();
        R21 = new Relay();                R22 = new Relay();
        R23 = new Relay();                R24 = new Relay();
        PP1 = new PowerPanel();           PP2 = new PowerPanel();
        PP3 = new PowerPanel();           PP4 = new PowerPanel();
        PP5 = new PowerPanel();           PP6 = new PowerPanel();
        PP7 = new PowerPanel();           PP8 = new PowerPanel();}}
```

Bibliography

[1] V. Armant, P. Dague, and L. Simon. Distributed Consistency-Based Diagnosis. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 113–127. Springer, 2008.

[2] T. Barros, L. Henrio, and E. Madelaine. Verification of distributed hierarchical components. *Electronic Notes in Theoretical Computer Science*, 160:41–55, 2006.

[3] J.F. Benders. Partitioning procedures for solving mixed variables programming problems. *Numeriche Mathematik*, 4:238–252, 1962.

[4] G. Berry and L. Cosserat. The ESTEREL synchronous programming language and its mathematical semantics. In *Seminar on Concurrency*, pages 389–448. Springer, 1985.

[5] D. Bertsimas and J.N. Tsitsiklis. *Introduction to Linear Optimization.* Athena Scientific, 1997.

[6] A. Chechetka and K. Sycara. No-commitment branch and bound search for distributed constraint optimization. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, page 1429. ACM, 2006.

[7] L. Console, G. Friedrich, and D. Dupré. Model-based diagnosis meets error diagnosis in logic programs. *Automated and Algorithmic Debugging*, pages 85–87, 1993.

[8] G.B. Dantzig and P. Wolfe. The decomposition principle for linear programs. *Operations Research*, (8):101–111, 1960.

[9] J. de Kleer and B.C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32, 1987.

[10] J. de Kleer and B.C. Williams. Diagnosis with behavioral modes. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 1989.

[11] R. Dechter. *Constraint processing.* Morgan Kaufmann, 2003.

[12] O. Dressler and P. Struss. The consistency-based approach to automated diagnosis of devices. In *Principles of Knowledge Representation*, pages 267–311. Center for the Study of Language and Information, 1997.

[13] E. Fabre, A. Benveniste, C. Jard, and M. Smith. Diagnosis of distributed discrete event systems, a net unfolding approach. *Preprint, Feb*, 2001.

[14] S. Genc and S. Lafortune. Distributed diagnosis of discrete-event systems using petri nets. In *ICATPN*, volume 2679 of *Lecture Notes in Computer Science*, pages 316–336. Springer, 2003.

[15] A. Gupta, C. Forgy, A. Newell, and R. Wedig. Parallel algorithms and architectures for rule-based systems. In *Proceedings of the 13th annual international symposium on Computer architecture*, pages 28–37. IEEE Computer Society Press, 1986.

[16] W. Hamscher, L. Console, and J. de Kleer. *Readings in model-based diagnosis.* Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 1992.

[17] D. Harel. Statecharts: A visual formalism for complex systems. *Science of computer programming*, 8(3):231–274, 1987.

[18] J.N. Hooker. Logic-based benders decomposition. *Mathematical Programming*, 96:2003, 1995.

[19] J.P. Ignizio. *Introduction to expert systems: the development and implementation of rule-based expert systems.* McGraw-Hill New York, 1991.

[20] P. Jégou and C. Terrioux. Hybrid backtracking bounded by tree-decomposition of constraint networks. *Artificial Intelligence*, 146(1):43–75, 2003.

[21] J. Kurien, X. Koutsoukos, and F. Zhao. Distributed diagnosis of networked, embedded systems, June 07 2002.

[22] R. Mailler and V. Lesser. Solving distributed constraint optimization problems using cooperative mediation. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems-Volume 1*, pages 438–445. IEEE Computer Society, 2004.

[23] O.B. Martin, S.H. Chung, and B.C. Williams. A tractable approach to probabilistically accurate mode estimation. In *Proceedings of the Eigth International Symposium on Artifical Intelligence, Robotics, and Automation in Space.* Citeseer, 2005.

[24] O.B. Martin, B.C. Williams, and M.D. Ingham. Diagnosis as approximate belief state enumeration for probabilistic concurrent constraint automata. In *AAAI*, pages 321–326. AAAI Press / The MIT Press, 2005.

[25] C. Mateis, M. Stumptner, and F. Wotawa. Modeling Java programs for diagnosis. In *ECAI*, pages 171–175. Citeseer, 2000.

[26] W. Mayer and M. Stumptner. Approximate modeling for debugging of program loops. *Proc. DX*, 4, 2004.

[27] T. Mikaelian. Model-based monitoring and diagnosis of systems with software-extended behavior. Master's thesis, MIT, 2005.

[28] T. Mikaelian, B.C. Williams, and M. Sachenbacher. Model-based monitoring and diagnosis of systems with software-extended behavior. In M.M. Veloso and S. Kambhampati, editors, *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA*, pages 327–333. AAAI Press / The MIT Press, 2005.

[29] T. Mikaelian, B.C. Williams, and M. Sachenbacher. Probabilistic Monitoring from Mixed Software and Hardware Specifications. *WS5*, page 95, 2005.

[30] P.J. Modi, W.M. Shen, M. Tambe, and M. Yokoo. An asynchronous complete method for distributed constraint optimization. In *Proceedings of the second international joint conference on Autonomous agents and multiagent systems, July*, pages 14–18. Citeseer, 2003.

[31] A. Petcu and B. Faltings. A scalable method for multiagent constraint optimization. In *International Joint Conference on Artificial Intelligence*, volume 19, page 266. Citeseer, 2005.

[32] C. Pinto and C. Terrioux. A new method for computing suitable tree-decompositions with respect to structured CSP solving. In *ICTAI (1)*, pages 491–495. IEEE Computer Society, 2008.

[33] R.J. Ragno. Solving optimal satisfiability problems through clause-directed A*. Master's thesis, MIT, 2002.

[34] N. Robertson and P.D. Seymour. Graph minors. III. Planar tree-width. *Journal of Combinatorial Theory, Series B*, 36(1):49–64, 1984.

[35] N. Rolander, M. Pekala, and D. Scheidt. A software simulation testbed to evaluate next-generation control algorithms, 2009.

[36] N. Roos, A. Ten Teije, A. Bos, and C. Witteveen. Multi-agent diagnosis: an analysis. In *Proceedings of the Belgium-Dutch Conference on Artificial Intelligence (BNAIC-01)*, pages 221–228. Citeseer, 2001.

[37] M. Sachenbacher and B.C. Williams. On-demand bound computation, 2004.

[38] R. Stallman and G.J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9, 1977.

[39] B.C. Williams, S.H. Chung, and V. Gupta. Mode estimation of model-based programs: Monitoring systems with complex behavior. In *International Joint Conference on Artificial Intelligence*, pages 579–585, 2001.

[40] B.C. Williams, M.D. Ingham, S.H. Chung, and P.H. Elliott. Model-based programming of intelligent embedded systems and robotic space explorers. IEEE: *Special Issue on Modeling and Design of Embedded Software*, 9(1), 2003.

[41] B.C. Williams and P. Nayak. A model-based approach to reactive self-configuring systems. In *Proceedings of the National Conference on Artificial Intelligence*, 1996.

[42] B.C. Williams and R.J. Ragno. Conflict-directed A* and its role in model-based embedded systems. *Discrete Applied Mathematics*, 155(12):1562–1595, 2007.