Achieving Real-Time Mode Estimation through Offline Compilation

John M. Van Eepoel

Achieving Real-time Mode Estimation through Offline Compilation

by

John M. Van Eepoel

Submitted to the Department of Aeronautics and Astronautics
on September 19, 2002 in Partial Fulfillment of the
Requirements for the Degree of Master of Science in
Aeronautics and Astronautics

ABSTRACT

As exploration of our solar system and outerspace move into the future, spacecraft are being developed to venture on increasingly challenging missions with bold objectives. The spacecraft tasked with completing these missions are becoming progressively more complex. This increases the potential for mission failure due to hardware malfunctions and unexpected spacecraft behavior. A solution to this problem lies in the development of an advanced fault management system. Fault management enables spacecraft to respond to failures and take repair actions so that it may continue its mission.

The two main approaches developed for spacecraft fault management have been rule-based and model-based systems. Rules map sensor information to system behaviors, thus achieving fast response times, and making the actions of the fault management system explicit. These rules are developed by having a human reason through the interactions between spacecraft components. This process is limited by the number of interactions a human can reason about correctly. In the model-based approach, the human provides component models, and the fault management system reasons automatically about system wide interactions and complex fault combinations. This approach improves correctness, and makes explicit the underlying system models, whereas these are implicit in the rule-based approach.

We propose a fault detection engine, Compiled Mode Estimation (CME) that unifies the strengths of the rule-based and model-based approaches. CME uses a compiled model to determine spacecraft behavior more accurately. Reasoning related to fault detection is compiled in an off-line process into a set of concurrent, localized diagnostic rules. These are then combined on-line along with sensor information to reconstruct the diagnosis of the system. These rules enable a human to inspect the diagnostic

consequences of CME. Additionally, CME is capable of reasoning through component interactions automatically and still provide fast and correct responses. The implementation of this engine has been tested against the NEAR spacecraft advanced rule-based system, resulting in detection of failures beyond that of the rules. This evolution in fault detection will enable future missions to explore the furthest reaches of the solar system without the burden of human intervention to repair failed components.

Thesis Supervisor: Brian C. Williams
Title: Associate Professor of Aeronautics and Astronautics

This page intentionally left blank.

Acknowledgements

This page intentionally left blank.

Table of Contents

List of Figures

Achieving Real-time Mode Estimation through Offline Compilation   13

This page intentionally left blank.

1   Introduction

1.1   Motivation

Spacecraft face many challenges in current and future missions due to the harsh environment of space and the complexity of spacecraft systems. Coupled with these challenges, additional problems are created by the growing number of spacecraft being developed, system design and manufacturing flaws and the increasing complexity of missions. These can cause unpredictable spacecraft behavior as well as component and system failures, which can have deadly repercussions. Spacecraft require a technology to increase robustness in the face of these problems. Spacecraft autonomy, more specifically fault management, provides a solution that permits space exploration and spacecraft to move beyond these obstacles. Fault management embodies the spacecraft with the intelligence that allows it to reason about faulty components and work around them to continue to achieve its mission goals. Spacecraft with this capability reduce the impact of failures and increase the likelihood of mission success.

Fault management systems can be designed at varying levels and complexities. In the most basic sense, a spacecraft can be considered autonomous if it has the ability to detect pre-specified failures and take repair actions. This type of autonomous system is based on a set of scenarios developed by human modelers and embedded in the spacecraft

processor. Anything outside of these scenarios causes the spacecraft to radio Earth for further instructions. In order to develop more complex scenarios, a human would have to reason about multiple components, their individual behaviors and failures. A more sophisticated fault management system automates this reasoning using a model of the spacecraft and the foundations of artificial intelligence. A system of this type reasons through component behaviors and interactions as prescribed by the model. These two distinct approaches demonstrate the difference between current fault management in spacecraft - rule-based systems that give repair actions for only certain specified faults, and the model-based approach that determines system behavior and repair actions for many faults.

The necessity of fault management is best demonstrated by looking at the needs of past and future missions. Take as an example the Mars Polar Lander. This spacecraft was scheduled to land in the polar regions of Mars, an environment with assumedly harsh conditions. Upon descent, the spacecraft prematurely cut its engine while it was still approximately 130 ft (40 m) off of the ground. This command likely caused the spacecraft to plummet to the surface and break apart on impact. It was determined that after the landing legs had deployed, a failed sensor mistakenly read that its landing leg had touched the surface. A more sophisticated fault management system would have enabled the spacecraft to compare the readings of all sensors, including the laser range finder. With a majority of the landing sensors reading '*no-ground-contact*', and the laser range finder reading a distance of 40 m, it could have reasoned that there was a faulty sensor and ignored it. This reasoning capability protects the spacecraft from component failures, allowing it to recover and complete its mission.

Take as another example the MESSENGER [*JHUAPL, 2002*] mission to Mercury currently being built and operated out of the Applied Physics Lab at Johns Hopkins University. System failures caused by the harsh

environment around Mercury are a primary challenge of this mission. In addition, due to the time delay of communication, the dependence of the spacecraft on transmissions from Earth hinders science collection and the completion of mission goals. Spacecraft autonomy would enable the spacecraft to independently plan and execute activities, and perform operations to maintain the health of the spacecraft. It offers the MESSENGER spacecraft a robust approach to handling failures and completing mission goals with minimal contact with Earth.

These examples give a variety of possible applications of basic and more sophisticated levels of fault management and autonomy. These are essential for missions as they explore further into our solar system and as spacecraft grow in complexity. If something unexpected occurs, the spacecraft could recover and still complete the activity without ever having to contact the ground for help. For the reasons detailed here, model-based autonomy and fault management will have a prominent role in the development of future spacecraft.

1.2   Mode Estimation Evolution

A component of the fault management system is mode estimation, which determines the behavior of the system using current sensor information. Mode estimation determines if components are faulty, but also tracks the nominal behavior of the system. This is a key aspect that enables an autonomous system to accurately control the spacecraft systems.

An accurate mode estimation engine must have several key attributes to achieve the goal of detecting failures and determining system behavior accurately. The engine must be capable of detecting single and multiple failures, using multiple sources of information to determine system

behavior, and have the ability to rank diagnoses of the system. Additionally, as available resources, including time, computational power and storage space, for fault management on board a spacecraft dwindle it becomes necessary to require faster response times and smaller memory allocation for these software processes. The mode estimation engine that has been developed, Compiled Mode Estimation (CME), was designed to address these concerns and be an improvement over previous mode estimation approaches.

Mode estimation leverages models and reasoning algorithms to determine the behavior of the system. Previous mode estimation engines required many computations in order to estimate the system behavior using these models and the current sensor information. CME has been developed to reduce the number of computations at run-time and address the real-time performance issues of these previous engines. CME is divided into two steps, an offline model compilation phase and online mode estimation engine. In the offline stage, the compiled model is generated by removing particular information that is costly to determine at run-time. This allows for the design of an any-time algorithm that can determine the system behavior in the online phase. CME addresses the concerns faced by current and future missions by providing a capability that can identify failures and nominal system behavior, and provide these for a real-time system.

Additionally, previous mode estimation engines have the potential to increase the risk of a mission. The benefits of developing models of the system and using reasoning algorithms to determine system behavior are to have the ability to identify many behaviors of the system, not just those that can be specified by a human modeler. However, the results of previous engines were unpredictable prior to the operation of the system. One of the key benefits of CME is it makes the possible diagnoses of the system explicit before the system operates due to the compiled model. This enables a human modeler to inspect the diagnoses for correctness.

Compiled Mode Estimation only provides one capability of a larger autonomy system. The following section presents the architecture of an autonomy system to highlight the utility of mode estimation, and the capability of an autonomous system.

1.3   Model-based Spacecraft Autonomy

Several different methods have been explored to engineer an autonomous system for spacecraft. To date, the two main approaches utilized have been rule-based and model-based systems. Rule-based autonomy specifies repair actions in response to observations of undesirable sensor information. These repair actions are based on a fixed set of scenarios identified by human modelers that have reasoned through the spacecraft component interactions. Model-based autonomy produces a robust approach to handling system failures by considering a larger set of spacecraft behavior using models and reasoning algorithms. It offers a way for human modelers to convey knowledge of failures in terms of common sense engineering models of spacecraft components. These models enable reasoning algorithms to determine the current behavior of the spacecraft, identify failures, diagnose and repair using sensor information. Model-based autonomy was selected as the basis of this research as it allows the spacecraft to reason through component interactions independent of a human modeler.

A model-based autonomous system is best understood through an explanation of its main components, and their interactions. Shown in Figure 1-1 is the paradigm of a model-based program and a model-based executive [*Williams 2, 2002*]. Here the fault management portion is labeled as the 'Deductive Controller'.
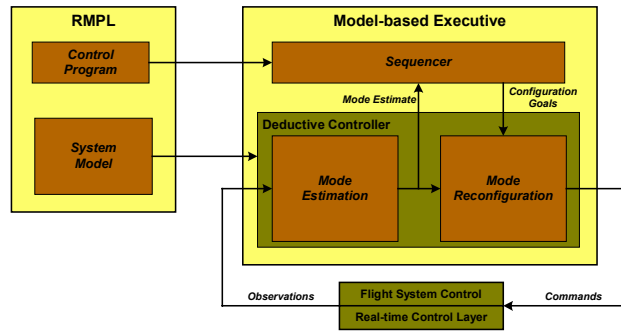
Figure 1-1 - Model-based Executive Architecture

The architecture shows the model as the starting point, described in the Reactive Model-based Programming Language (RMPL) [*Ingham, 2001*]. The model has two different levels, a control program and a system model. The control program encodes a model of the intended behavior of the spacecraft. This is a way to describe sequences of actions that achieve certain goals, such as telling the propulsion system to thrust. The system model encodes the spacecraft component behavior and their interactions.

The model-based executive acts as a high level controller using the estimated behavior of the spacecraft to determine control actions, encoded as '*commands*' in Figure 1-1, which place the spacecraft in a desired configuration. The model-based executive is comprised of three major components, the *Sequencer*, the *Mode Reconfiguration* engine, and the *Mode Estimation* engine. The Sequencer's task is to execute a specified sequence of actions, where the actions are specified within the control program. These actions are then translated by the *Sequencer* to a '*configuration goal*', which specifies the desired modes for the spacecraft components. The *Mode Reconfiguration* engine then uses these configuration goals, the current *mode estimate* of the system and the system model, to determine the control actions, or *commands*, to apply to the spacecraft components in order to achieve the configuration goal. The final piece of the architecture is the *Mode Estimation* engine that uses the

observations, commands and the system model to determine the current *mode estimate* of the system. Observations represent the current readings of sensors in the spacecraft system and are vital to determining the current behavior of the spacecraft. The *Mode Estimation* and the *Mode Reconfiguration* engines work together to provide the spacecraft with a fault management capability. The *mode estimates* represent the current behavior of the system, and are used to exact repairs on the system determined by the *Mode Reconfiguration* engine.

A mode estimate represents the *Mode Estimation* engine's best determination of the behavior of the components in the spacecraft. The behavior of a component is encoded in the system model, and the task of Mode Estimation is to determine the best mode for each component in the system that is consistent with the observations, commands and the model. The Mode Estimation engine can be thought of as the doctor on the spacecraft. It identifies the behavior of the spacecraft including normal or faulty operation. It diagnoses the components' behavior by determining the most likely component modes. Estimating system behavior is an essential task for an autonomy architecture to correctly and accurately control the system. Mode estimation provides an accurate representation of the current behavior of the system, which is needed to control the system. It is essential to increase the accuracy of mode estimation to enable the correct control on the spacecraft by the model-based executive.

CME seeks to increase the accuracy of mode estimation and provide an engine with the capabilities described previously. However, to understand the process of determining system behavior, requires developing a very primitive mode estimation engine and demonstrating this using an example. The following sections present an approach to mode estimation in 1.4, followed by the enhancements to this process using model compilation in Section 1.5.

1.4   Mode Estimation

The Mode Estimation engine maps the system model, the observations and commands to a set of component modes that reflect the behavior of the system. The task of mode estimation is to choose the proper component modes that are consistent with the model constraints, and also agree with the observations and commands. Mode estimation is an example of the task of inferring hidden state [*Wiliams 2, 2002*]. Since the modes of these components cannot be directly obtained, hence hidden, then they can only be estimated using the system model, observations and commands. In the case of spacecraft systems, there are only observations that give insight into the behavior of the components in the spacecraft. Mode estimation is framed using the theory of hidden state problems, the foundations of logical inference and the theory of Hidden Markov Models.

The process to estimate these component modes is best understood by first discussing the inputs and outputs of the mode estimation algorithm, and then demonstrating the process on an example spacecraft system. The example gives a context and a grounded way to discuss the basic steps of mode estimation.

1.4.1   Inputs and Outputs

The mode estimation engine uses the system model, the current observations and commands to determine an estimate of the component behavior, represented by a mode estimate. These have been discussed briefly, but a more thorough definition of each of these inputs and outputs is now given. Figure 1-2 depicts these inputs and outputs.
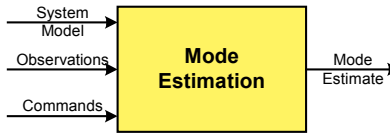
Figure 1-2 - Inputs and Outputs of Mode Estimation

The '*system model*' represents the behavior of each component in the system being monitored. The components are modeled by a set of discrete modes. Each discrete mode is expressed by a set of constraints that describe the component behavior within the mode and probabilistic transitions to other modes of the component. These constraints relate the observations, commands and intermediate variables. The '*observations*' represent the sensor information of the system. The '*commands*' represent the control actions that the Model-based Executive may perform on the system. The intermediate variables are an internal variable in the system model that enables communication between different components.

The output '*mode estimate*' is an assignment of modes, one for each component in the system that is consistent with the system model, the observations and the commands. There are many mode estimates of the system at any given time, which are ordered using probabilities. This assignment of component modes is only an estimate since the system model includes probabilistic transitions. Probabilistic transitions are necessary to capture the behavior of failures and intermittency within a real system.

1.4.2 Mode Estimation Example

There have been many systems that solve the mode estimation problem [*deKleer, 1987, deKleer, 1989, Williams 1996, Kurien, 2000, Hamscher, 1992*]. This section presents the basic steps of mode estimation, followed

by a description of a spacecraft system, and ends with a description of mode estimation applied to the example spacecraft system.

1.4.2.1 The Mode Estimation Process at a Glance

The '*system model*', as described before, is comprised of models of each component in the spacecraft system. Each of these models includes modes that characterize different behaviors of the component within the overall spacecraft system. The modes are described by specified model constraints that capture the behavior of that mode and by probabilistic transitions to modes within the same component model.

Mode estimation determines the set of component mode assignments that are consistent with the constraints associated with the component modes and the transitions. To accomplish this, mode estimation must perform two key steps:

1. Determine a set of likely next mode assignments given likely mode assignments in the previous state and the transitions.
2. Choose the most likely, current component modes that are consistent with the mode constraints, the observations and control values.

Mode estimation computes the likely next mode assignments by choosing transitions that mention mode assignments in the previous state and storing the targets of the transitions in the set of likely next mode assignments. The second step of mode estimation computes the current mode estimate by searching for combinations of component modes and determining if they are consistent with the constraints. Effectively, the mode estimation process must choose the optimal component modes, optimal due to the probabilistic transitions. Mode estimation is then framed as an optimal

constraint satisfaction problem where the solution is the set of component modes that gives the highest probability, and that also satisfy the model constraints.

The process of mode estimation gives the system the ability to determine component behavior accurately and at a higher level than the continuous dynamics of the system. Mode estimation has the ability to determine faulty components in terms of discrete modes. For instance, mode estimation is able to determine that a valve is *stuck-open* instead of specifying this in terms of continuous sensor readings, such as *flow = 0.54 ft³/min*. This high level specification of the system behavior enables the Model-based Executive to determine recovery actions.

1.4.2.2 NEAR Spacecraft Power System

The steps of the basic mode estimation are best demonstrated by example. Our example is taken from the Near Earth Asteroid Rendezvous (NEAR) mission, operated by the Johns Hopkins University Applied Physics Lab in Columbia, MD. The mission launched in February of 1996, rendezvoused with the Eros asteroid on February 14, 2000. The spacecraft lasted much longer than anticipated and performed a groundbreaking maneuver. It landed on the surface of the Eros asteroid in February of 2001, and the spacecraft continued to transmit data back to Earth until it ran out of power in February 28, 2001.

The NEAR spacecraft has eight systems interacting together to maintain the health of the spacecraft, to control the attitude, to collect science information, to enable communication, and to provide power to the spacecraft. The power system of the NEAR spacecraft is chosen as the example system for its complexity and familiarity from everyday life. For instance, the interactions of a battery and a charger are easy to understand

since they are in cars, trains, cell phones, etc. However, the power system of the NEAR spacecraft does offer interesting complexities due to the collection of power in space. For instance, the power generated by the solar arrays must be regulated to a specific level so that the sensitive instruments are not harmed.

The NEAR Power sub-system is shown below in Figure 1-3. The example focuses in particular on the NEAR Power storage sub-system, highlighted with a circle in the figure.
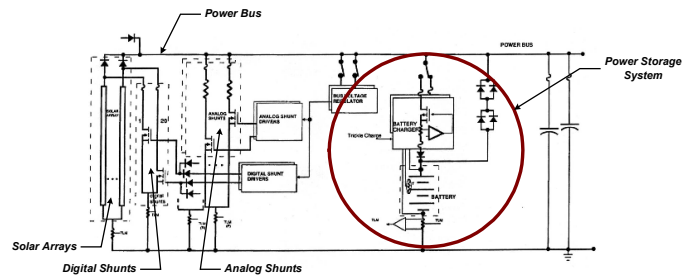
Figure 1-3 - NEAR Power System

The power system is built up using solar arrays that generate power, digital and analog shunts that regulate the power, and components to store the power, built using a switch, redundant battery chargers and a battery. The NEAR Power system is an example of a direct energy transfer (DET) power system [*Wertz, 1999*]. All of the incoming power gathered from the solar arrays is initially put on the power bus. However, this incoming power might be too much for the power bus and spacecraft components to handle. The digital and analog shunts are placed in the system to prevent this excess power from affecting the spacecraft components. These shunts act to dissipate the excess power when they are enabled. These shunts are supported by the analog and digital shunt drivers, and bus voltage regulator that determine when shunts should be enabled or disabled.

Achieving Real-time Mode Estimation through Offline Compilation   27

The next stage of this power system is the power storage system. The components of the power storage system are a switch, two redundant chargers, and a NiCd battery. The available sensors for the power storage system measure the incoming bus voltage, the outgoing battery voltage, and the temperature of the battery. The switch is linked to the redundant chargers to change the charger that receives the bus voltage. This switch guarantees that only one charger can charge the battery at any given time. The chargers use the voltage from the switch to output a current that charges the battery. The chargers have two different charging modes, a trickle charge and a full-on charge. The trickle charge is used if the battery is nearly fully charged so as to keep it at a full charge. This mode delivers a small current to the battery. The full-on charge is used if the battery charge is low. This mode delivers the maximum current possible to charge the battery as quickly as possible. The battery behavior is based on the level of charge remaining in the battery and the current rate of discharge of the battery. The indicator of the level of charge in the battery is the temperature, since there is no direct sensor for the level of charge. The indicator for the rate of discharge of the battery is the voltage sensor, depicted between the bottom of the battery and the power bus in Figure 1-3. These observations indicate if the battery is currently discharging, charging or full.

The power generation system, made up of the solar arrays, shunts and shunt drivers, and the power storage system interact to give the voltage required by the NEAR spacecraft. The power storage system reacts to the needs of the spacecraft and the available power generated from the power generation components. If the solar arrays provide too much power, as is the case when the spacecraft is near Earth, then the power storage system stores this extra power, up to the capacity of the battery. If the solar arrays cannot provide enough power for the spacecraft, then the power storage system reacts automatically and supplies the necessary voltage. The reason that the

Achieving Real-time Mode Estimation through Offline Compilation

solar arrays provide too much power near Earth is that the solar arrays are designed to provide the required power for the spacecraft when it is at the asteroid, Eros. Since the asteroid is further away from the sun than Earth, the solar power available is much less. It is for these reasons that the power system has a means to dissipate, as well as store, excess power.

The power storage system is made the focus of further discussion and example because of its component interactions and interesting component modes.   The modes of the components and interactions between the components are detailed in Figure 1-4. The different types of variables and their domains are listed below.

Observable: *bus-voltage, battery-voltage, battery-temperature*

Intermediate: *switch-voltage, charger-current*

Component: *switch, charger-one, charger-two, battery*

Command: NONE

The domains for each variable type are:

voltage: zero, low, nominal, high
temperature: low, nominal, high
current: zero, trickle, nominal, high

Bus Voltage

Switch

Charger-One
Charger-Two
Stuck-Charger-One
Stuck-Charger-Two
Unknown

Switch Voltage   Switch Voltage

Charger-One

Full-On
Trickle
Off
Broken
Unknown

Charger-Two

Full-On
Trickle
Off
Broken
Unknown

Charger Current

Battery

Full
Charging
Discharging
Dead
Unknown

Battery Temperature   Battery Temperature

Battery Voltage
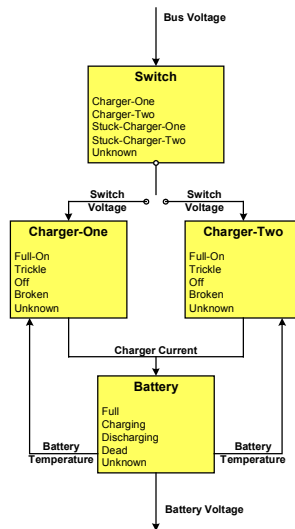
Figure 1-4 - Component Mode Breakdown of the NEAR Power Storage System

The power storage system has several design characteristics worth noting. For instance, Figure 1-4 and Figure 1-3 shows the chargers using the temperature of the battery as an input. This sensor reading indicates the level of charge in the battery, which is used by the charger to determine how to charge the battery. When the temperature is high, this means that the battery is full, indicating to the charger that it only needs to trickle-charge the battery. When the temperature is nominal, this means that the battery is not full, indicating to the charger that it should apply the maximum current possible, putting the *charger* in the *full-on* mode.

The component modes shown here each have associated constraints describing their behavior. The switch modes, for either '*charger-1*' or '*stuck-charger-1*', are used to pass the incoming *bus-voltage* to *charger-1*. The difference between the two is that the mode '*stuck-charger-1*' is a failure mode indicating that the switch cannot move from the position for *charger-1*. The modes of the charger model the type of charge being

applied to the battery. In the *full-on* mode, the charger is sending a *nominal* current to the battery to give it the highest charge possible. In the *trickle* mode the *charger* sends only a trickle-charge to the battery to keep the charge level full. The broken mode for the chargers may be deduced by detecting that the output '*charger-current*' is *high*. The model for the *charger* is built using the switch voltage and the output charger current to model the component modes, and using the battery temperature to model the transitions between modes. For the *battery*, the '*full*', '*charging*' and '*discharging*' modes model the behavior described earlier using the input current from the charger and the output battery voltage. The full representation of these component models is given in Appendix A.

1.4.2.3 Mode Estimation Example

This section demonstrates the two basic steps of mode estimation using the NEAR Power Storage system. Recall that the first step of mode estimation assumes that there already exists a previous mode estimate. Using the transitions and the previous mode estimate, the algorithm determines the set of component modes that are reachable in one time step. To determine this, the algorithm first finds the transitions whose source are the component modes in the previous mode estimate. The constraints are then extracted from the transitions and added to the model constraints. This is represented graphically in Figure 1-5.
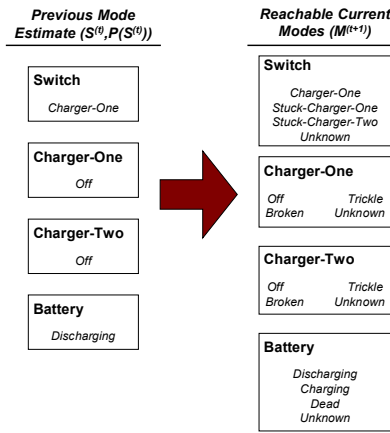
Figure 1-5 - Step 1 of the Mode Estimation Process

Depicted on the figure above is the previous mode estimate, which is the pair $(S^{(t)}, P(S^{(t)}))$. This pair denotes the state, $S^{(t)}$, as a choice of a single mode for each component in the system, and the probability of this mode estimate, $P(S^{(t)})$. For this example, the probability of the previous mode estimate is 1. The figure denotes the set of component modes that are reachable in the current time step, '$t+1$', and these are determined by the transitions. For instance, in the case of the *switch*, the '*charger-2*' mode is not allowed in the current modes because the *switch* only transitions to '*charger-2*' if *charger-1* fails. Since *charger-1* was '*off*' in the previous mode estimate, then the transition of the *switch* from '*charger-1*' to '*charger-2*' is not allowed.

To summarize, the first step of mode estimation has determined the transitions that are allowed from the previous mode estimate, and calculated the set of reachable current component modes. The mode estimation algorithm has added the constraints from all the transitions into the model constraints and extracted the model constraints from the reachable current

component modes. These constraints and this set of reachable current component modes are then used in the second step of mode estimation.

The second step of mode estimation determines which sets of reachable component modes are consistent with the model constraints and the observations. In order to determine all different combinations of the component modes, the calculation must be performed methodically. The sets of current component modes are generated through systematic search. As a straw man, mode estimation uses chronological search to determine the sets of component modes, depicted in Figure 1-6.
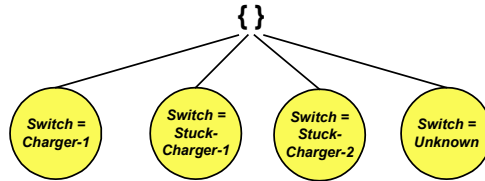
Figure 1-6 - Search Tree Expansion Using Component Modes

This first expansion shows the search using the current component modes of the *switch*. The search then continues to expand the tree until it determines a mode to each component in the power storage system. The search follows the first leaf of the tree, '*switch = charger-1*' and expands the next component under it, *charger-1*. Figure 1-7 depicts this expansion.
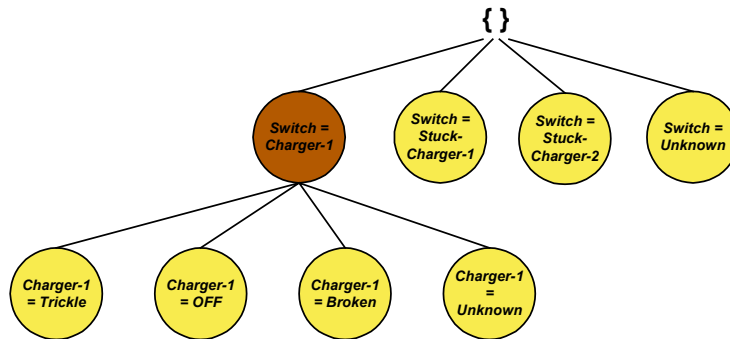
Figure 1-7 - Search Tree Expansion with Two Components Shown
Achieving Real-time Mode Estimation through Offline Compilation   33

The search would continue until it determined a complete set of component modes. From the listing of current component modes in Figure 1-4, the first full choice of component modes is:

(*switch = charger-1*), (*charger-1 = trickle*), (*charger-2 = trickle*), (*battery = charging*)

This set of reachable component modes must be checked to insure that it is consistent with the mode constraints. To demonstrate this process, consider the following current observations of the system.

(*bus-voltage = nominal*), (*battery-temperature = nominal*), (*battery-voltage = nominal*)

To determine if the mode estimate is consistent, mode estimation begins by propagating variable values through the model constraints of the component modes.[1] This process enables mode estimation to predict values for many of the observation and intermediate variables in the system. For a mode estimate to be consistent, any value it predicts must agree with the current observations. Using the mode estimate from above, the remaining values within the system that must be determined are the *switch-voltage* and the *charger-current*, one of each per charger. Beginning at the *switch*, and using the observation '*bus-voltage = nominal*', this is propagated through the component model for '*switch = charger-1*', which gives '*charger-1.switch-voltage = nominal*' and '*charger-2.switch-voltage = zero*'. These values are then propagated through the models of the chargers for '*charger-1 = trickle*' and '*charger-2 = trickle*'. The resultant value for the output of charger-1 is '*charger-1.charger-current = trickle*'. When propagating through the component model for the mode '*charger-2 = trickle*', the input

---

[1] Using a complete satisfiability algorithm, if no variable value is predicted for a variable, an assignment must be found that is consistent with the observations. [*Williams, 2002*]

switch-voltage must be '*nominal*' or '*low*' according to the mode constraints. This results in conflicting results for the variable '*charger-2.switch-voltage*'. The mode estimation algorithm then throww out this set of component modes as an inconsistent mode estimate.

Once mode estimation determines if a set of component modes is consistent or inconsistent, it uses the search tree to generate another set of component modes to test for consistency. This process repeats until the generation of mode estimates has explored a certain amount of the probability space, or the entire search tree is explored and all consistent mode estimates have been generated. The steps of the mode estimation process described here have:

1. Generated a set of current component modes using the transitions and a previous mode estimate.
2. Used this set of current component modes to generate mode estimates
3. Tested each for consistency, and kept those that are consistent.

The algorithm described above is an overly simplified approach to calculating these key steps. However, even this simple algorithm contains many of the key attributes of a mode estimation engine, described in section 1.2. It is able to use multiple sources of information to determine the modes of components, and it is able to determine single and multiple faults. Finally, the algorithm ranks mode estimates using probabilistic transitions. This information however can be used more efficiently in the search. There have been many algorithms designed to perform a variant of mode estimation [*deKleer, 1987, deKleer, 1989, Williams, 1996, Kurien, 2000, Ingham, 2001*]. Earlier mode estimation engines [*deKleer, 1987, deKleer, 1989*] did not have transitions in the models of components.

1.4.3   Issues in Mode Estimation

The mode estimation algorithm described in the previous section is a brute force approach to generating mode estimates. The algorithm generates many combinations of component modes that are inconsistent with the model constraints and observations. The problem with generating these inconsistent mode estimates is the time spent in determining that it is inconsistent. The propagation of model information and the search over possible component modes is an NP-hard problem resulting in an exponential computation in the number of components.

The test for consistency of mode estimates is costly due to the search for possible assignments in the system. The example above demonstrated this search and the ensuing propagation of variable values. Notice in the example above the amount of time taken to determine the values of the '*charger-1.switch-voltage*' and the '*charger-2.switch-voltage*'. In particular, in order to determine these values, mode estimation performed a search over variables whose values were not determined by propagation. This results in an overall exponential computation. As the number of components in the system increases, so do the number of variables that must be determined for each mode estimate. Determining these values is the computational bottleneck of mode estimation.

1.4.4   Tracking System Trajectories

Recent mode estimation engines have incorporated transitions into the models of system components to enhance the accuracy of mode estimates [*Williams, 1996, Kurien 2000*]. These systems tracked the behavior of the system over time by maintaining the likely mode estimates at each time step. The trajectory tracking is depicted in Figure 1-8, where one path (noted in red) is kept at each time increment.
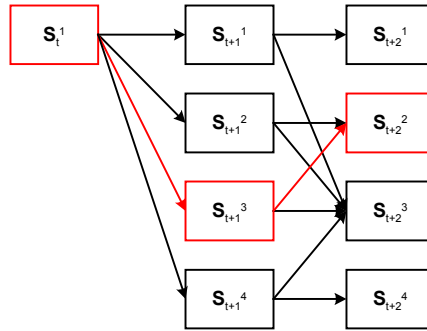
Figure 1-8 - Tracking Mode Estimates Over Time

Tracking mode estimates over time gives the benefit of diagnosing complex failures that evolve over time. Trajectory tracking requires determining if certain transitions between states are allowed. Determining this requires a consistency test, similar to the one described for mode constraints. Tracking likely trajectories limits the computations required to determine if taking a transition is consistent with the system model. However, only tracking likely mode estimates limits the diagnoses of the system to these likely trajectories, but a less likely trajectory could become a likely one in the future as more observations are collected to refine the mode estimates. Systems that track likely trajectories may miss these types of diagnoses.

An alternative approach is to track consistent mode estimates from one time step to the next. This approach enables more accurate estimation of the system behavior since states, not trajectories, are tracked over time. A mode estimation engine with this capability tracks the evolution of many mode estimates, requiring many more computations than the tracking of likely trajectories. However, the benefits of tracking mode estimates over time is the increased accuracy of the mode estimates and the ability to diagnose complex failures. CME develops an approach for tracking mode estimates that is enabled by the compiled model.

1.5 Compilation

The performance of mode estimation may be improved by compiling the system model before the system needs the mode estimates. This process removes the need to determine consistency of mode estimates by identifying all sets of infeasible component modes in the system and compiling transitions to remove the need for consistency determination at run-time. The compilation process is the key enabling technology for the next evolution of mode estimation for spacecraft, CME.

Compilation enables the mode estimation process to perform fewer computations to determine consistent mode estimates, as well as making the reasoning process of mode estimation more explicit. Compilation is a two step process of compiling the mode constraints and the transitions. The compilation of the model constraints results in generating conflicts, which are a more intuitive representation of the model constraints than an uncompiled model. The conflicts represent infeasible assignments that correspond to particular observations. These are easier to grasp and inspect by a human, making the diagnoses more explicit. By determining all conflicts in an offline process, the exponential computation of consistency is no longer performed at the time of execution.

The compilation process that has been designed is discussed first, followed by a simple example to demonstrate the compilation process. This discussion focuses on the compilation of mode constraints. The compilation of transitions is presented in Chapter 4.

1.5.1 The Basics

Recall, from the mode estimation example (Section 1.4.2.3), that the mode estimate was inconsistent with the system model and observations. To determine the inconsistency, the algorithm identified a discrepancy between the observed value for the '*charger-2.switch-voltage*' and the value predicted from the system model. The identification of this discrepancy leads to a conflict. A conflict is defined as a set of component modes that cannot be true given the current observation. In the case of the example, the resulting conflict would be:

$$\neg\, [\, (charger\text{-}2 = trickle)\,]$$

This states that it is inconsistent to assign *charger-2* the mode *trickle* because of the discrepancy between the observation and the prediction. Identifying these discrepancies and determining the infeasible sets of component mode assignments, conflicts, is the key to the compilation process. By generating all conflicts, mode estimates can be generated without performing search and propagation for assignments to intermediate variables. The savings of this compiled model

The outputs of the compilation process are all conflicts of the component mode constraints, and the observations used to generate them. For instance, the observation (*bus-voltage = nominal*) is associated with the conflict $\neg\,[$ (*charger-2 = trickle*)]. In order to determine all of the conflicts within the system model, the compilation algorithm uses the component mode constraints, and tries all combinations of observations. The different combinations of observation and component mode variables are propagated through the system model, and compilation identifies all conflicts using search techniques. So, the compilation process in effect pretends that the observations are real, and stores all conflicts associated with a set of observations. This process has the potential to be an exponential search due to the permutations of observations and component modes, which increases

dramatically as the system grows. Compilation is made tractable by developing an algorithm that looks for the minimal set of conflicts and that does not explore any supersets of a previously generated conflict. The algorithm designed to perform the compilation of the system model is presented in Chapter 3. The process of transition compilation is developed in Chapter 5.

1.5.2 Compilation Example

Recall that there are two parts to the system model, the constraints associated with the modes of components and the constraints on transitions between these modes. The compilation process compiles both of these portions separately. For simplicity, the example shown here only describes the compilation of the constraints on the component modes.

Using the NEAR Power Storage system, the example shows the compilation of the intermediate variable '*charger-1.switch-voltage*' and '*charger-2.switch-voltage*'. In compiling this intermediate variable, the compilation process searches over the observation and component mode variables to identify inconsistent combinations. The compilation tests for inconsistency by performing backtrack search and propagation. For instance, if the observation variable *bus-voltage* is found in the search, it is then propagated through the *switch*, *charger-1* and *charger-2*, for different combinations of these modes. Considering the observation '*bus-voltage = nominal*', the compilation process then chooses a component mode for the *switch* by searching for it. In choosing the component mode '*switch = charger-1*', the compilation process then propagates the variables and determines '*charger-1.switch-voltage*' and '*charger-2.switch-voltage*'. As before, these values are '*charger-1.switch-voltage = nominal*' and '*charger-2.switch-voltage = zero*'. The compilation process then tries

different combinations of the component modes for *charger-1* and *charger-2* to determine all conflicts.

The compilation process would determine that *charger-1* could not be in the *off* mode if the incoming voltage was nominal. This is because if the incoming voltage is greater than zero, then the charger should be charging the battery in some manner, either trickle charging or giving it a maximum charge. In the case of *charger-2*, compilation would identify that the charger could not be in any mode, except for *off*, because the voltage coming into the charger is not greater than zero. The compilation process then identifies the following conflicts for the observation '*bus-voltage = nominal*'.

$$\neg \, [\, (switch = charger\text{-}1) \wedge (charger\text{-}1 = off) \,]$$
$$\neg \, [\, (switch = charger\text{-}1) \wedge (charger\text{-}2 = trickle) \,]$$
$$\neg \, [\, (switch = charger\text{-}1) \wedge (charger\text{-}2 = full\text{-}on) \,]$$

The conflicts shown here represent the result of reasoning using observations and the constraints of a system model. The conflict states that if the '*bus-voltage = nominal*', then it is not possible for the *switch* to be at the *charger-1* position, and the *charger-2* to be in the *trickle* or *full-on* mode. The conflicts give an intuitive interpretation between observation values and modes of components that are infeasible. By determining these conflicts before the spacecraft operates makes mode estimates more explicit and inspectable by a human modeler. For instance it is easier to understand the conflicts above than verifying the correctness of the mode estimate using the uncompiled model.

$(switch = charger\text{-}1)$, $(charger\text{-}1 = trickle)$, $(charger\text{-}2 = off)$, $(battery = charging)$

This explicit representation increases the confidence in the underlying system model, allowing a human modeler to inspect the correctness of the diagnoses before the operation of the system. The intuitive representation of conflicts is easier to grasp. Being given a mode estimate and a set of observations still requires a human to think about the component models and interactions to insure correctness. However, a conflict is simpler as it ordinarily does not contain a large number of component modes, thereby localizing the reasoning for a human to insure correctness.

1.6 Compilation and Mode Estimation

Compilation is only one piece that enables the next evolution of mode estimation. Compilation transforms the system model into a representation that makes the computations of mode estimates simpler. However, the two basic steps of mode estimation must still be performed during the time the spacecraft is operating. The first step of the overall mode estimation process is unchanged. The mode estimation algorithm still creates a list of reachable, current component modes using the transitions. However, this is enabled by the compiled transitions so that the engine does not require any satisfiability to determine if a transition is enabled.

The difference comes in the second step of the mode estimation process. The conflicts enable the search algorithms to be designed such that the sets of current component modes generated automatically satisfy the model constraints and are consistent with the observations. The algorithms that perform these computations follow in the remaining chapters. Compiled Mode Estimation is designed to contain the key attributes of a mode estimation engine described in Section 1.2. This engine is capable of using multiple sources of information, determining single and multiple faults, rank the mode estimates, and track multiple mode estimates over time.

Chapters 2, 3 and 4 present previous mode estimation systems and compilation, followed by the formal presentation of Compiled Mode Estimation in Chapters 5 and 6. An advanced reader may wish to skip these chapters and jump to the chapters relating to the Compiled Mode Estimation engine. Chapter 7 presents the system used to validate the correctness of this new mode estimation algorithm, accompanied by the results of the mode estimation algorithm diagnosing this system. Chapter 8 presents conclusions drawn from this work, and Chapter 9 presents future work that could further enhance model-based autonomy and the Compiled Mode Estimation engine.

2   Conflict-Based Mode Estimation

2.1   Model-based Mode Estimation Framework

Model-based mode estimation identifies the behavior of a system's components using a system model and current observations and commands. It is the aim of this research to develop a method to compile the system model to enable a mode estimation engine that is capable of determining mode estimates more efficiently than previous mode estimation systems. Model compilation is built upon the heritage of conflict-based algorithms designed to perform mode estimation efficiently. The goals of this research are to develop the algorithms for a mode estimation engine that exploits the properties of the compiled model. Our approach, called Compiled Mode Estimation, builds upon the results of a series of diagnostic engines, in particular the General Diagnostic Engine [*deKleer 1987*], Sherlock [*deKleer 1989*], Livingstone [*Williams, 1996*] and Mini-ME [*Chung, 2001*] diagnostic tools.

It is important to review these engines to give the development of mode estimation and the relation of each to Compiled Mode Estimation. The GDE engine developed the use of conflicts to determine diagnoses efficiently. The Sherlock engine expanded upon GDE by using behavioral modes and introduced incremental generation of the diagnosis through a 'generate-and-test' approach. Mini-ME is the first diagnostic engine to use a compiled model to generate diagnoses for the system. Finally,

Livingstone was the first engine to incorporate transitions into the system model and developed a modification to Sherlock's 'generate-and-test' approach to determine a diagnosis of the system. CME extracts these key benefits of each system described here to build a mode estimation engine that is efficient in its computations, and explicit in its diagnoses.

The General Diagnostic Engine (GDE) [*deKleer, 1987*] relies solely on the model of operational modes to isolate faults. GDE detects failures using a model of correct behavior to determine discrepancies between expected and observed behavior. GDE relates the discrepancy to the component modes that predicted the behavior. These component modes are identified by GDE as an infeasible combination of component modes, or a conflict. If the observations are inconsistent with the model of correct behavior then a subset of the components are determined to be faulty. However, GDE does not have the capacity to specify how components would fail.

The Sherlock [*deKleer, 1989*] diagnostic engine generalizes many of the ideas of GDE, such as using the differences between expected and observed behavior, and generating conflicts to determine the likely mode assignments. Sherlock uses nominal and faulty behavioral modes to describe the model of components. The use of behavioral modes improves the diagnostic discrimination over GDE and enables the ability of the engine to identify failure mechanisms. This improved discrimination allows the overall autonomy system to determine the system behavior more precisely. Note, however that Sherlock and GDE only give an instantaneous diagnosis of the system as opposed to tracking variations in mode assignments over time.

The Mini-ME [*Chung, 2001*] diagnostic engine uses the GDE approach of divide and conquer, but the divide step of diagnosis is performed in offline compilation. Mini-ME uses the Sherlock model of behavioral modes to

describe the models of components. However, it does not incur the penalty of determining consistency of the mode estimate with the observations as this has already been performed at compile-time. Mini-ME's is able to give diagnostic discrimination similar to Sherlock, but can still only determine instantaneous mode estimates of the system.

The ability to track mode estimates over time further improves diagnostic discrimination and offers the ability to track intermittent faults. The Livingstone reactive system leverages the foundations of GDE and Sherlock [*Williams, 1996*] to track the most likely mode estimate at each time step. The mode estimation engine used within the Livingstone system built upon the concept of behavioral modes in Sherlock, and introduced transitions between these behavioral modes to track the behavior of the system over time. The introduction of transitions enabled Livingstone to increase diagnostic discrimination of and extend it to intermittent faults. Like GDE and Sherlock, Livingstone incorporates the use of conflicts into its mode estimation algorithm, and introduces a method to test mode estimates more efficiently [*Williams, 1998*]. This was done so that Livingstone could be used in real-time to provide mode estimates and enable a reactive autonomy system that controlled a spacecraft. Livingstone was tested on the Deep Space One spacecraft that rendezvoused with Comet Borrelly in November 2001. The test successfully demonstrated the benefits and uses of fault management and planning on-board a spacecraft under an array of fault scenarios.

These three systems are first presented to lay the groundwork for Compiled Mode Estimation and the approach to compiling the system model. This chapter discusses the GDE and Sherlock diagnostic engines. This framework is then used to present the Mini-ME diagnostic engine in Chapter 3 along with the approach to compiling the mode constraints. Chapter 4 discusses the underlying system model used within the

Livingstone system, and the Livingstone process of generating mode estimates. These are then used to present the Compiled Mode Estimation engine and the compilation of transitions in Chapter 5.

2.2   General Diagnostic Engine (GDE)

One of the early systems to perform multiple diagnostic tasks was the General Diagnostic Engine (GDE), developed by deKleer and Williams [*deKleer, 1987*]. GDE diagnoses systems through a divide and conquer approach. As mentioned previously, GDE uses the notion of a conflict to direct its search for the correct diagnosis. GDE uses the conflicts to 'divide' the problem of diagnosis into sub-problems, and then combines the solutions to these sub-problems, or 'conquers' them, into a full, consistent diagnosis of the system. Our approach, Compiled Mode Estimation, uses a similar divide and conquer approach, but shifts the first step, conflict recognition, to an offline process called Dissent Generation.

This section reviews diagnosis in GDE by first defining the inputs and outputs of the architecture, then detailing the algorithm by example, and concluding with an analysis of GDE.

2.2.1   GDE Inputs and Outputs

GDE uses observations and a system model as inputs to determine a set of diagnoses that represents the possible behavior of the system at a particular point in time. The architecture of GDE denoting this is shown in Figure 2-1.
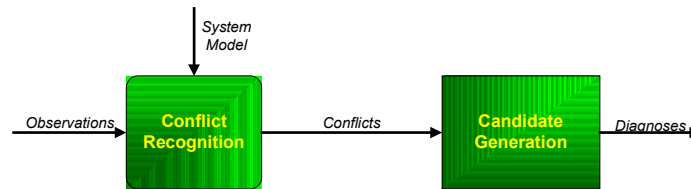
Figure 2-1 - General Diagnostic Engine Architecture

The observations are an assignment to each observation variable in the system model and represent the sensor information. The conflicts represent infeasible sets of component modes. GDE generates all conflicts for a given set of observations in the *Conflict Recognition* phase. Each output diagnosis assigns to each component in the system a mode that expresses its current behavior. The diagnosis is constrained to be consistent with the observations and the system model. A diagnosis is similar to a mode estimate, except that a diagnosis generated by GDE has only two modes per component, *ok* and *not ok*.

The constraints on the 'ok' mode express the normal operation of the component. The 'not ok' mode does not have any constraints associated with it, thereby being consistent with any behavior outside of normal operation. GDE was developed to model components such as simple logic systems (and, or, not, etc. gates) and mathematical operators (addition, subtraction, multiplication, division, etc.), which consist of a single operating mode.

Figure 2-1 identifies the two steps of the GDE algorithm. The first, 'conflict recognition', uses the system model and observations to generate conflicts. Conflicts are a representation of infeasible mode assignments, as described in Chapter 1. The second step, 'candidate generation', uses these conflicts to generate the current diagnoses for the system. Within the 'candidate generation' phase several computations occur that transform the conflicts

first into constituent diagnoses that represent feasible mode assignments, and then into kernel diagnoses that represent the minimal sets of component modes that satisfy the constituent diagnoses.[2] The definitions of the inputs, outputs and internal types to GDE are given below.

- Candidate $\equiv \langle (x_{1m}{=}ok\ or\ not{-}ok),...,(x_{nm}{=}ok\ or\ not{-}ok) \rangle$ where $x_{im} \in \Pi_{im}$
  n = number of mode variables in the system.

- Diagnosis (D) $\equiv \langle (x_{1m}{=}ok\ or\ not{-}ok),...,\neg(x_{nm}{=}ok\ or\ not{-}ok) \rangle$ where $x_{im} \in \Pi_{im}$,
  with $D \wedge C_{Mi}^{(t)} \wedge O^{(t+1)}$ consistent, where $C_{Mi}^{(t)}$ represents the mode constraints, and $O^{(t+1)}$ represents the current observations.

- Conflict $\equiv \neg\left[(x_{1m}{=}ok),...,(x_{pm}{=}ok) \right]$ where $x_{im} \in \Pi_{im}$, and $p \le n$, where
  n = number of mode variables in the system. Denotes that the combination of component modes is not true, so it cannot be true that x_{1m} is ok and x_{pm} is ok.

- Constituent Diagnosis (cd) $\equiv \left[(x_{1m}{=}ok\ or\ not{-}ok),...,(x_{pm}{=}ok\ or\ not{-}ok) \right]$ where
  $x_{im} \in \Pi_{im}$ and $p \le n$. The assignment $x_{im} = ok$ is considered a constituent diagnosis, and cd represents the set of constituent diagnoses for a conflict.

- Kernel Diagnosis (kd) $\equiv \left[(x_{1m}{=}ok\ or\ not{-}ok),...,(x_{pm}{=}ok\ or\ not{-}ok) \right]$ where
  $x_{im} \in \Pi_{im}$ and $p \le n$. The kernel diagnosis represents a minimal set covering of the set of conflicts.

2.2.2   Diagnosis with GDE

Recall that GDE relies on a divide and conquer paradigm to generate diagnoses for the system. The divide step is embodied in the 'conflict recognition' phase of the algorithm, while the conquer step is given by the 'candidate generation' phase. This section details each of these steps through an example. For a formal discussion on the theory of GDE, see [*deKleer, 1987*].

---

[2] This is a rational reconstruction of GDE according to [REF OPSAT Paper]

The diagnostic process of GDE has the key property of generating conflicts from discrepancies in observations. This is leveraged extensively in the compilation of the system model that enables the Compiled Mode Estimation engine. Additionally, the 'candidate generation' phase and the approach to generating kernel diagnoses lays the groundwork for the online mode estimation engine of CME. The following example gives the intuition for generating conflicts and the process to use these to determine diagnoses.

Consider the example of the NEAR Power system described in Chapter 1, with the simplification shown in Figure 2-2. The models cannot capture the complexity of the different modes of the NEAR Power Storage system, but is adequate to demonstrate the GDE diagnostic process.
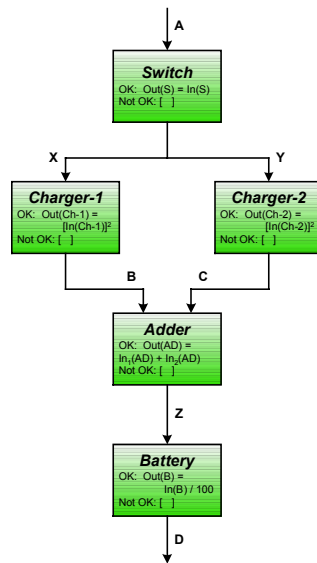
Figure 2-2 - Simplified NEAR Power Storage System for GDE Example

In this figure, the observable variables, Π_o, are represented by 'A', 'B', 'C', and 'D', and the hidden, or intermediate variables Π_d, are represented by

50   Achieving Real-time Mode Estimation through Offline Compilation

'X', 'Y' and 'Z'. The figure shows the constraints on each component mode variable as well. The operations that each component performs are explained as follows. The *switch* delivers its input to the chargers if it is in the '*ok*' mode. The *chargers* take the output from the *switch* and square it. For an input of A = 2, this results in the chargers outputting B = 4 and C = 4. These values are then summed by an *adder*, to result in Z = 8 in this case, and then passed to the *battery*. The *battery* outputs its input divided by 100, which results in the value D = 2/25.

2.2.2.1 Conflict Recognition

The process of conflict recognition relies on several operations to determine all conflicts. First, the process must identify discrepant values, or symptoms. Second, these symptoms must be traced back to the mode assignments used to predict the discrepant values in the symptom. These mode assignments comprise the conflict that represents the infeasible mode assignments for the current observations. GDE generates the minimal set of conflicts for all symptoms using a combination of constraint propagation and an Assumption-based Truth Maintenance system [*deKleer 2, 1987*]. The details of the ATMS is beyond the scope of this document.

Consider the following observations: A = 5, B = 9, C = 9, and D = 0.18. Using the model in Figure 2-2, GDE generates all conflicts for this set of observations by propagating values through the models of the components of a candidate and comparing the observed behavior and the predicted behavior. If a discrepancy is found, then a conflict is extracted from the candidate.

To demonstrate this, assume that the *switch*, *charger-1*, *charger-2*, *adder* and *battery* are all in the '*ok*' mode. GDE first searches over single

component mode assignments to test in the conflict recognition phase. Consider the mode (*switch = ok*). Propagating the input A = 5 through this results in the values X = 5 and Y = 5. This does not result in any discrepant values, so GDE continues to search for combinations of component mode assignments to test. Consider the combination {(*switch = ok*), (*charger-1 = ok*)}, and propagating the observation B = 9 back through the *charger-1* constraints results in the value X = 3. GDE recognizes that the two values do not agree and has identified a symptom. GDE then traces this symptom back to the components used to determine the values for X to identify the conflict. GDE determines that the component modes *switch = ok* and *charger-1 = ok* are the conflict for this symptom. GDE continues to propagate and search for symptoms to generate the minimal set of conflicts. For this set of observations GDE generates the conflicts:

$$\neg\,[\,(switch = ok) \wedge (charger\text{-}1 = ok)\,]$$
$$\neg\,[\,(switch = ok) \wedge (charger\text{-}2 = ok)\,]$$

These conflicts are used in the next phase of GDE, 'candidate generation'.

2.2.2.2 Candidate Generation

The candidate generation phase uses the conflicts to determine the minimal set of component mode assignments that resolve the conflicts, represented as kernel diagnoses. The conflicts can be transformed through logic operations to obtain:

$$[\,\neg\,(switch = ok) \vee \neg\,(charger\text{-}1 = ok)\,]$$
$$[\,\neg\,(switch = ok) \vee \neg\,(charger\text{-}2 = ok)\,]$$

This is interpreted, in the case of the first conflict, that the *switch* is *not-ok* or the *charger-1* is *not-ok*. Either of these component mode assignments will resolve the first conflict, associated with the discrepant values B = 9 and B = 25. Similarly, the assignments *switch = not-ok* and *charger-1 =*

Achieving Real-time Mode Estimation through Offline Compilation

not-ok resolve the second conflict associated with the discrepancy in C. The minimal set of component mode assignments that resolves all conflicts, the kernel diagnosis, is generated by performing a minimal set covering over the conflicts. For this example, the resulting kernel diagnosis is (*switch = not-ok*) as it is the only mode assignment that satisfies the two constituent diagnoses. A full diagnosis is given by extending the kernel diagnosis to include a mode for each component in the system. Any superset of a kernel diagnosis is also a diagnosis, so GDE finds many diagnoses for the system. Each of these full diagnoses must contain the mode (*switch = not-ok*) to be correct. Some of the diagnoses are:

(*switch = not-ok*), (*charger-1 = ok*), (*charger-2 = ok*), (*adder = ok*),
    (*battery = ok*)

(*switch = not-ok*), (*charger-1 = ok*), (*charger-2 = ok*), (*adder = not- ok*),
    (*battery = ok*)

(*switch = not-ok*), (*charger-1 = ok*), (*charger-2 = not-ok*), (*adder = ok*),
    (*battery = ok*)

(*switch = not-ok*), (*charger-1 = ok*), (*charger-2 = ok*), (*adder = ok*),
    (*battery = not-ok*)

This example demonstrates the basic steps of the GDE algorithm. This section demonstrated the steps of the 'candidate generation' phase, and used the results of the 'conflict recognition' phase. The 'candidate generation' phase not only generates a single diagnosis, but also generates all diagnoses of the system for a given set of observations and ranks them. For instance, the example above is ordered by likelihood since the diagnosis with a single fault, *switch = not-ok* is listed first, and the remaining diagnoses all contain two failed components.

The combination of conflict recognition and candidate generation solves an NP hard problem, and hence is worst case exponential in the number of

mode variables. GDE uses several techniques to focus the search in the conflict recognition phase, given in [*deKleer, 1987*].

2.2.3 Analysis of GDE

GDE has many benefits in its approach to determining system behavior. The diagnostic process of GDE is predicated on identifying all conflicts for a given set of observations, and reconstructing all possible diagnoses from these conflicts. GDE has shown that the complete set of conflicts is sufficient to generate all diagnoses. This is the key point of developing GDE because CME is predicated on the same approach. The difference is that CME shifts the identification of conflicts to an offline compilation phase, and reconstructs the diagnoses from these conflicts online.

GDE focused on the diagnosis of static systems and assumed no knowledge of failure models. Sherlock, discussed in the next section, introduces fault models and focuses diagnosis on generating the most likely diagnoses. Mini-ME is a compiled version of Sherlock because it identifies conflicts in an offline phase, while still generating only the most likely diagnoses online. Livingstone generalized Sherlock to systems with dynamic, time-varying behavior.

2.3 Sherlock

GDE addressed the diagnosis problem for static systems where the behavior of components are expressed as either *ok* or *not-ok*. Sherlock [*deKleer, 1989*] extends the space of possible behaviors for components by incorporating knowledge of nominal and failure modes. Sherlock improves upon the conflict-based approach to diagnosis of GDE by focusing on

generating only most likely diagnoses. The approach to generating most likely diagnoses is the key contribution of Sherlock to CME.

The introduction of behavioral modes creates a significant increase in the computations needed to determine a diagnosis. Sherlock addresses this by generating diagnoses in a generate and test approach. Instead of generating all conflicts associated with the current observations as GDE has done, Sherlock generates the conflicts incrementally by identifying likely combinations of component mode assignments, candidates, using the probabilities.

This section gives an overview of the Sherlock diagnostic process by first discussing its inputs and outputs, and then demonstrating the Sherlock algorithm by example.

2.3.1   Sherlock Inputs and Outputs

Sherlock uses a best-first 'generate and test' approach to determine the likely diagnoses for a set of current observations. Sherlock first generates a set of component mode assignments, a *candidate*, and then tests this candidate to determine if it is consistent with the current observations and system model. If the candidate is not consistent, then it generates one or more conflicts for the candidate, which are returned to the generator. The generator then determines the next most likely set of component mode assignments that satisfy the known conflicts, similar to GDE's candidate generation. This loop continues until all possible diagnoses have been generated or some stopping criterion has been met, such as a particular number of diagnoses. The architecture of Sherlock is shown in Figure 2-3.
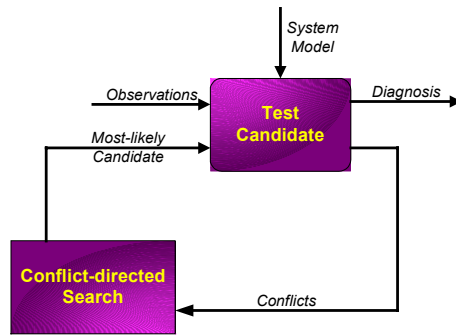
Figure 2-3 - Sherlock Diagnostic Engine Architecture

The input system model is expanded from the model of GDE by using behavioral modes to describe component behavior. These modes are capable of describing constraints for different nominal operational modes and for different fault modes. Fault modes always include the *unknown* mode, which contains no constraint. Sherlock expresses mode constraints similar to GDE by generalizing the domain of the variables from {*ok*, *not-ok*} to {*nominal*, … *fault* …, *unknown* }. The Sherlock system model is defined as follows:

$$\text{System Model} \equiv \cup \left[(x_{im} = v_{ij}), C_{Mi}, p(x_{im} = v_{ij}) \right] \text{ where the } \sum_{x_{im}} p(x_{im} = v_{ij}) = 1.$$ Denotes that each component mode, $x_{im} = v_{ij}$, has an associated constraint, and an associated probability.

The set of observations, a candidate, a diagnosis and conflicts are similar to GDE. The set of observations are an assignment to each observation variable. A candidate is an assignment to mode variables, and a diagnosis is a candidate that is consistent with the mode constraints and observations. The conflicts represent inconsistent sets of component mode assignments. The candidate and diagnosis have an associated probability, give as:

$$P(C) = \prod_{x_{im} = v_{ij} \in C} p(x_{im} = v_{ij})$$

Equation 2-1 - Probability of a Candidate in Sherlock

The remaining section give the intuition of Sherlock's best-first generate and test algorithm through an example.

2.3.2 Diagnosis with Sherlock

Sherlock frames diagnosis as a best-first generate and test search where candidates are generated, tested for consistency, and conflicts are extracted from the candidate if it is inconsistent. These conflicts are used to generate a new candidate. This process is necessary since the behavioral modes explodes the space of possible diagnoses, making them exponential in the number of components. It is infeasible to enumerate and test these diagnoses for consistency since the test for consistency is an exponential computation.

Instead, Sherlock uses the probabilities on component modes to focus the diagnosis to test likely candidates for consistency before testing less likely candidates. The probability of a candidate, defined in Equation 2-1, is given by the product of the probabilities of the component mode assignments in that candidate. The probability of a candidate is updated using the probability, P(O). The update equation is given as follows:

$$P(C_i \mid O) = \frac{P_i(O) \bullet \prod_{x_{im} \in C_i} p\left(x_{im} = v_{ij}\right)}{\sum_j P_j(O) \bullet \prod_{x_{im} \in C_j} p\left(x_{im} = v_{ij}\right)}$$

The numerator represents the probability that a candidate predicts all current observations, and the denominator is a normalization factor. P(O) represents the probability that a candidate correctly predicts an observation. If a candidate predicts all observations correctly, then P(O) = 1. If a candidate does not predict, or refutes, the observations, then P(O) = 0. Finally, if a candidate neither predicts or refutes an observation, then any

value in the domain of the observation is equally likely, so $P(O) = 1/n$, where n is the number of possible values for the observation. As an example, a candidate that contains the unknown mode of a component makes no predictions on the observation variable q, so in that case $P(O) = 1/n_q$, assuming the candidate correctly predicts the remaining observations.

To demonstrate the best-first candidate generation of Sherlock, consider the NEAR Power storage system used in GDE, modified now to have behavior modes.
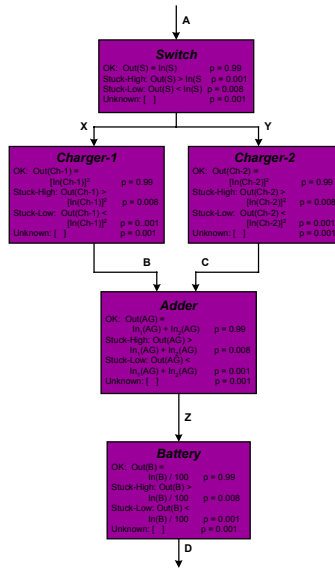
Figure 2-4 - NEAR Power Storage System modified to have Behavioral Modes

The modes of the components give additional fault modes, and still maintain the operational mode described in GDE, and the unknown mode that does not have any model constraints. The probability for each component mode is shown to the right of its constraint. The fault modes for the switch are stuck-high, and stuck-low capturing that the output sent to the chargers is either higher or lower than expected. The chargers are

modeled with a stuck-high and a stuck-low fault mode that captures when the output is higher or lower than the expected squaring, respectively. The adder and battery have similar modes stuck-high and stuck-low that constrain the output to be greater or lower than expected.

Sherlock generates conflicts for a given set of observations and a candidate in the same way GDE performed conflict recognition, except that Sherlock does not determine all conflicts for a given set of observations, but only those relevant to the particular candidate. As more observation information is incorporated, more conflicts are generated enabling Sherlock to focus the diagnosis more.

Sherlock is able to determine instantaneous diagnoses given the current observations for A, B, C and D, and the system model. Sherlock first chooses a candidate, and in the absence of conflicts, chooses the most likely mode assignment for each component. Suppose that A = 5, B = 9, C = 9 and D = 0.18. Sherlock determines that the most likely candidate is:

   {*switch = ok, charger-1 = ok, charger-2 = ok, adder = ok, battery = ok*}
with p = 0.95.

Sherlock then tests if this candidate is consistent with the system mode constraints and the observations. The consistency check identifies a discrepancy in the values of X and Y. The mode *switch = ok* predicts that X = 5 and Y = 5. However, the modes *charger-1 = ok* and *charger-2 = ok* results in X = 3 and Y = 3, respectively. The resultant conflicts are:

   ¬[(*switch = ok*), (*charger-1 = ok*)]
   ¬[(*switch = ok*), (*charger-2 = ok*)]

Sherlock uses these conflicts and the probabilities of component modes to focus on likely diagnoses. The conflict identifies infeasible sets of

assignments. To resolve the conflicts, Sherlock chooses other component modes not mentioned in the conflict. For instance the modes that would resolve the first conflict include: *switch = stuck-high, switch = stuck-low, switch = unknown, charger-1 = stuck-high, charger-1 = stuck-low, charger-1 = unknown*. Sherlock chooses the minimal set of most likely component modes that resolves all conflicts, or kernel diagnoses. Sherlock only generates the most likely kernel diagnosis, and then extends this to a candidate to be tested. In the case of these conflicts, the most likely kernel diagnosis is:

(*switch = stuck-low*, p = 0.008)

This results in the candidate:

{ *switch = stuck-low, charger-1 = ok, charger-2 = ok, adder = ok, battery = ok* }

with p = 0.00768.

Although this probability is low, it has not been normalized by the sum of all the probabilities of the diagnoses. Sherlock then tests this candidate for consistency. In performing this, Sherlock identifies that the component mode *switch = stuck-low* predicts X < 5 and Y < 5. Using the component modes *charger-1 = ok* and *charger-2 = ok* results in X = 3 and Y = 3. The consistency test does not identify any more conflicts, so this is then labeled as a diagnosis of the system.

The key feature to note is the speed with which Sherlock found the most likely diagnosis of the system. The benefits of using a conflict-directed search and guiding the choice of candidates by probability focus the search for the most likely diagnosis. A detailed, updated presentation of a Sherlock-like algorithm is presented in [*Williams, 2002*], with the original algorithm given in [*deKleer, 1989*].

2.3.3 Analysis of Sherlock

The Sherlock diagnostic system has built upon the foundations of the GDE algorithm and its use of conflicts to generate diagnoses. Sherlock has the ability to use multiple sources of information, the observations, to determine the current behavior of the system. The key benefit of Sherlock is its approach to generating most likely diagnoses in a best-first order using a conflict-directed search and the probabilities of component modes. This search enables Sherlock to solve the problem of exponential cost in the candidate generation phase. The CME engine leverages this search approach to generate mode estimates online in a best first order. The combination of the compilation and the conflict-directed best first search enable CME to track multiple mode estimates over time. The drawback of the Sherlock approach is the exponential cost of satisfiability to generate conflicts at run-time. The Mini-ME engine addresses this issue by compiling the mode constraints on component modes in an offline process.

This page intentionally left blank.

3 Compilation of Conflict-Based Mode Estimation

3.1 *Motivation for Mode Compilation*

The GDE and Sherlock methods of diagnosis both incur significant computational costs at run-time while generating conflicts. This is exponential in the worst case. In addition, for GDE, candidate generation determines all possible diagnoses for the system, while only a few most likely diagnoses are required. The set of all diagnoses is exponential in the worst case. Sherlock addresses the problem with candidate generation through best first enumeration. However, it incurs an exponential cost testing consistency of the candidate and extracting the conflicts of the candidate. The goal of Mini-ME is to increase performance by removing the need for satisfiability and conflict generation in the online determination of system behavior. The key insight from GDE and Sherlock is that all conflicts are sufficient to reconstruct the diagnoses of the system. Mini-ME then moves the process of conflict generation to an off-line process.

This relates directly to CME and the goal of removing satisfiability completely from the online process. GDE, Sherlock and Mini-ME do not incorporate transitions into the system model. This is done in the Livingstone system, discussed in Chapter 4. CME must then compile both portions of the system model, the mode constraints and the transitions, to have the capability to track mode estimates over time without the need for

satisfiability. Mini-ME develops the approach for one portion, mode compilation that is leveraged by CME. The approach for transition compilation is developed in Chapter 5. This chapter presents the Mini-ME engine and its method of using the compiled model online to generate diagnoses in Section 3.2. Section 3.3 presents the method employed to compile mode constraints for Mini-ME and CME.

3.2 Mini-ME

The first step towards model compilation for CME is a compiled version of Sherlock, called Miniature Mode Estimation (Mini-ME) [*Chung, 2001*]. This engine compiled component mode constraints into conflicts, and used these conflicts in an online mode estimation algorithm to determine mode estimates for the system. The online mode estimation algorithm is similar to the candidate generation step of GDE, and uses probabilities to generate likely mode estimates similar to Sherlock. The conflicts are used to generate a kernel diagnosis that satisfies all conflicts, and this kernel diagnosis is extended to a mode estimate by ensuring that all components in the system have an assigned mode. The architecture of the Mini-ME engine is shown below in Figure 3-1.
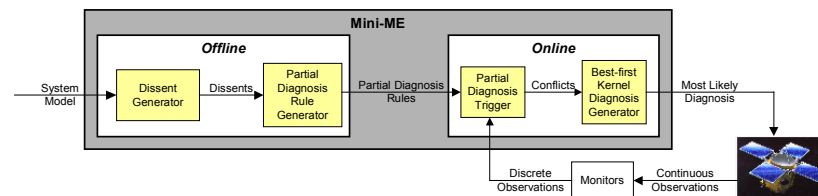
Figure 3-1 - Architecture of the Mini-ME Engine

The architecture denotes the generation of dissents in an offline process. Dissents are a mapping of observations to conflicts. The dissents are

transformed by Mini-ME offline into *partial diagnoses*. These partial diagnoses have a similar representation to the constituent diagnoses in GDE, so the term constituent diagnosis is used to refer to these partial diagnoses. This offline transformation enables Mini-ME to avoid performing this step online. In the online portion, Mini-ME only needs to determine the appropriate sets of constituent diagnoses to use given the current observations. The final step to generating a consistent diagnosis is to determine the smallest set of component mode assignments, kernel diagnoses, that are a minimal set covering of the constituent diagnoses. By choosing assignments in the constituent diagnoses, Mini-ME reconstructs the diagnosis from the conflicts, enabling the assignments chosen to satisfy all conflicts and be consistent with the observations. Mini-ME uses component mode probabilities to generate the most likely kernel diagnoses, and then extends the kernel to a full diagnosis.

3.2.1   Mini-ME Example

The diagnostic process of Mini-ME is best demonstrated by example using the NEAR Power storage system described in Chapter 1. Focusing on the interaction of the *switch* and redundant *chargers* with the observation variables of the *bus-voltage*, Figure 3-2 depicts the system.
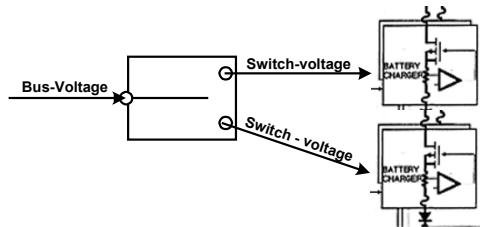
Figure 3-2 - NEAR Power Storage System Example

The modes of the components are given below (note that the unknown mode is not shown):

 switch

(*charger-1*, p=0.49), (*charger-2*, p=0.49), (*stuck-charger-1,* p=0.01),

(*stuck-charger-2,* p=0.01)

charger-1, charger-2

(*full-on*, p = 0.39), (*trickle*, p = 0.39), (*off*, p = 0.2), (*broken*, p = 0.02)

bus-voltage : { *zero, low, nominal* }

The following are some of the relevant dissents:

```
[ ] ⇒ ¬[ SWITCH = STUCK-CHARGER-1 ∧ CHARGER-2 = FULL-ON]
[ ] ⇒ ¬[ SWITCH = STUCK-CHARGER-1 ∧ CHARGER-2 = TRICKLE]
[ ] ⇒ ¬[ SWITCH = CHARGER-2 ∧ CHARGER-1 = FULL-ON]
[ ] ⇒ ¬[ SWITCH = CHARGER-2 ∧ CHARGER-1 = TRICKLE]
[ ] ⇒ ¬[ SWITCH = CHARGER-1 ∧ CHARGER-2 = FULL-ON]
[ ] ⇒ ¬[ SWITCH = CHARGER-1 ∧ CHARGER-2 = TRICKLE]
[ BUS-VOLTAGE = LOW ] ⇒ ¬[ SWITCH = CHARGER-1 ∧ CHARGER-1 =
FULL-ON ]
[ BUS-VOLTAGE = LOW ] ⇒ ¬[ SWITCH = CHARGER-1 ∧ CHARGER-1 =
OFF ]
[ BUS-VOLTAGE = NOMINAL ] ⇒ ¬[ SWITCH = CHARGER-1 ∧ CHARGER-1
= OFF ]
```

These dissents express the links between *switch* and *charger* modes so that only one charger is on at any time, and that the charger that is on corresponds to the position of the switch. The dissents make this explicit. For instance, in the third and fourth dissents, note that the component modes that are inconsistent are the *switch = charger-2* and the mode *charger-1 = full-on* or *trickle*. This limits the modes of the *charger-1* to be either *off*, *broken* or *unknown*.

The first step in Mini-ME is to use the current observations to determine the relevant dissents, and their consequents, the conflicts. Consider the observation that the *bus-voltage = nominal*. Mini-ME triggers those dissents that mention the observable *bus-voltage = nominal*, and any that do not mention an observable. The following constituent diagnoses represent the first two dissents:

```
[ SWITCH = CHARGER-1, SWITCH = CHARGER-2, SWITCH = STUCK-
CHARGER-2, CHARGER-2 =
  TRICKLE, CHARGER-2 = OFF, CHARGER-2 = BROKEN]
[ SWITCH = CHARGER-1, SWITCH = CHARGER-2, SWITCH = STUCK-
CHARGER-2, CHARGER-2 = FULL-
  ON, CHARGER-2 = OFF, CHARGER-2 = BROKEN]
```

The remaining sets of constituent diagnoses are not shown here for brevity. Mini-ME uses these sets of constituent diagnoses to generate kernel diagnoses, which represent a minimal set covering of the constituent diagnoses. This process is similar to the GDE process of 'candidate generation'. The generation of kernel diagnoses is guided by the probability of component mode assignments. The set covering begins by determining the most likely component mode assignment in the first set of constituent diagnoses. In this case, this results in:

$$switch = charger\text{-}1, \text{p} = 0.49$$

To perform the minimal set covering, Mini-ME determines the sets of constituent diagnoses that mention this assignment as a constituent diagnosis. Additionally, Mini-ME chooses a set of constituent diagnoses that this one does not appear. For instance, the assignment *switch = charger-1* would not appear in the set of constituent diagnoses derived from dissents 5 and 6. The sets of constituent diagnoses for dissent 5 are:

```
[SWITCH = CHARGER-2, SWITCH = STUCK-CHARGER-1, SWITCH =
STUCK-CHARGER-2, CHARGER-2 =
  TRICKLE, CHARGER-2 = OFF, CHARGER-2 = BROKEN]
```

Mini-ME uses this set of constituent diagnoses to choose a mode assignment for *charger-2* that is the most probable. This corresponds to the mode assignment (*charger-2 = trickle*) with p = 0.39. This results in the set of assignments { (*switch = charger-1)*, (*charger-2 = trickle*) } with p = 0.1911. Mini-ME would however recognize that this set of assignments is infeasible because of the 6th dissent that says that the two are infeasible.

Mini-ME would then choose another constituent diagnosis from the constituent diagnoses for dissent 5. The next most likely component mode assignment is *charger-2* = *off* with p = 0.2. The combination of assignments results in a p = 0.098. This set of assignments does satisfy the current dissents for this observation. This results in Mini-ME extending this kernel diagnoses to a full diagnosis by choosing the most likely component mode for *charger-1*, which results in *full-on* with p = 0.49.

The mode estimate determined by Mini-ME is the most likely of all possible mode estimates since the search for it was guided by probabilities. Mini-ME determines the most likely diagnosis using the dissents that pertain to the current observations. This diagnosis is guaranteed to be consistent with the observations because the set of conflicts are sufficient to generate all diagnoses, as shown by GDE and Sherlock. What remains is to develop the process of mode compilation to generate dissents offline.

3.3   Mode Compilation

This section focuses on the offline compilation of the system model, more specifically the compilation of the mode constraints. The compilation process is developed by first discussing the inputs and outputs, and then discussing the mode compilation algorithm. Finally, the section concludes with an example demonstrating mode compilation.

3.3.1   Inputs and Outputs

The mode compilation algorithm uses a system model to compile the constraints on the modes of components in the system. The algorithm outputs a set of dissents that map the observations to the conflicts. The dissents are generated by identifying conflicts for sets of observations and

component modes. Compilation is related to the GDE step of conflict recognition, but now uses all combinations of observations and component modes. Figure 3-3 shows these inputs and outputs below.
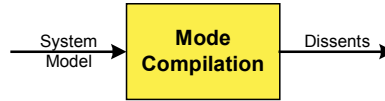
Figure 3-3 - Mode Compilation Inputs and Outputs

The system model is the same representation used in the Sherlock diagnostic engine, with constraints restricted to propositional logic. The system model is comprised of behavioral modes for each component, each with associated constraints. Mode compilation compiles these constraints into conflicts, encoded as dissents.

To achieve efficiency, all conflicts should be generated offline. This is accomplished by generating all conflicts for all possible combinations of observations. The dependency between the observations and the conflicts is encoded compactly in the dissents. A dissent has the following general form:

- dissent (d) $\equiv$ observations $\Rightarrow$ conflict

This definition states that a combination of observation assignments implies a conflict, or an infeasible set of component mode assignments. The definitions of the inputs and outputs are then:

- System Model $\equiv \ \bigcup \left[(x_{im}=v_{ij}), C_{Mi}, p(x_{im}=v_{ij}) \right]$ where $x_{im} \in \ \Pi_m$ and C_{Mi} is

  expressed using discrete observation assignments, x_{io}, and component

  mode assignments, x_{im}.

- Dissents (D) $\equiv \bigcup_j \left\{ \left[\left(x_{1o}=v_{1l_1} \right) \wedge \cdots \wedge \left(x_{po}=v_{pl_p} \right) \right] \Rightarrow \neg \left[\left(x_{1m}=v_{1l_1} \right) \wedge \cdots \wedge \left(x_{qm}=v_{ql_q} \right) \right] \right\}_j$

  where $x_{io} \in \Pi_O$ and $x_{im} \in \Pi_m$, also $p \le n_o$ and $q \le n_m$

3.3.2 Mode Compilation Algorithm

Dissents are generated from the system model by enumerating all possible combinations of observations and component mode assignments. In order to determine if a particular combination of observations and component mode assignments is a dissent, the algorithm must determine if it is inconsistent with the mode constraints. This follows from the logical statement:

$$\Phi \models (observations \Rightarrow conflict)$$

Equation 3-1 - Logical Statement for Dissent Generation

This states that the system model, Φ, entails the dissent, or the statement that observations imply a conflict. This is transformed to a statement of inconsistency:

$$\Phi \wedge observations \wedge \text{modes is inconsistent}$$

Here *modes* represents the component modes in *conflict*. The mode compilation algorithm then tests various combinations of observation and component mode assignments and determines if they are inconsistent with the system model.

The mode compilation algorithm only generates the smallest number of dissents that captures the constraints of the system model. This requires generating the minimal set of dissents so that no dissent is a superset of another. So, to be a dissent, a combination of observations and component modes must be inconsistent with the system model and not be a superset of any previously generated dissents. The mode compilation algorithm uses a conflict-directed Enumeration algorithm to guarantee that the minimal set of dissents is generated.

The Enumeration algorithm is framed as an optimal constraint satisfaction problem. The key is to use the satisfiability engine as an unsatisfiability

engine that is capable of determining if sets of assignments are inconsistent with the constraints of the model. The Enumeration algorithm seeks to generate the minimal set of dissents by enumerating from longest to shortest by length and performing a subsumption check so that no supersets of a dissent are generated. To increase performance, the Enumeration algorithm uses dissents that have been generated to limit the search tree of the OCSP. The algorithm adds a dissent as a conflict of the search, thus pruning those branches of the search tree that would explore the assignments in the dissent. For example, if the Enumeration algorithm identifies that {*bus-voltage = nominal*, *switch = charger-1*, *charger-1 = off*} as a dissent, then this combination of assignments is used so that no supersets are ever explored.

This frames the Enumeration algorithm as an OCSP thus leveraging the previous work of OCSP solvers [*Williams, 2002*]. In order to develop the enumeration algorithm, the problem of optimal constraint satisfaction is reviewed, followed by the algorithm that generates dissents using the optimal constraint satisfaction solver, OPSAT

3.3.3 Optimal Constraint Satisfaction

An optimal constraint satisfaction problem finds a solution, x, that satisfies a set of constraints and maximizes a cost function, $f(x)$. Formally, an OCSP is defined as:

Given a set of variables 'x' and their domains, choose the best assignment to all variables that will maximize the function f(x), subject to constraints G_x.

Obtaining the solution, x, to these problems has been the focus of much research and many algorithms. One such algorithm that solves OCSP's is the OPSAT algorithm [*Williams, 2002*]. OPSAT solves the constraint satisfaction problem by determining the best assignments to a set of optimization variables, y, that are a subset of x. The choice of these assignments is guided by the optimization function, f. An OPSAT problem is stated as follows.

$$OPSAT(s) \equiv \langle \vec{y}, f, CSP \rangle$$

$$CSP(s) \equiv \langle \vec{x}, D_x, G_x \rangle$$

where

$\vec{x} \equiv$ all variables in the system model
$D_x \equiv$ the domains of the vector of variables, $\vec{x}$
$G_x \equiv$ the model constraints to be satisfied, or unsatisfied
$\vec{y} \equiv$ a subset of the variables $\vec{x}$ to be optimized
$f \equiv$ a function to optimize
$OPSAT(s) \rightarrow$ an assignment to each variable $\vec{x}$

Figure 3-4 - Definition of an OPSAT Problem

The solution generated by OPSAT is an assignment to all variables in x a value from their domain, D_x that satisfies the constraints G_x, where the assignments to the subset of variables y maximizes the function f. OPSAT determines the assignments for the variables, y, in a generate and test approach, similar to Sherlock. OPSAT generates candidates using a conflict directed search, and then tests these candidates for consistency using the modeling constraints, G_x. The test for consistency is captured in satisfiability and unsatisfiability engines. Recall that Sherlock used the probabilities on component modes and maximized the product of the probabilities to generate candidates. OPSAT generalizes this to a function, f, to use to find the optimal set of assignments to y. OPSAT uses the function f to guide the generation of candidates so that likely candidates are

explored before less likely candidates. OPSAT uses a full satisfiability approach to determine consistency that is similar to GDE and Sherlock.

The satisfiability engine generates conflicts by identifying discrepancies in variable values. OPSAT generates only the minimal set of conflicts, meaning that no supersets of a conflict are generated. To achieve this, OPSAT maintains all conflicts generated and installs them in the satisfiability engine. Recall in Sherlock that the conflict recognition phase tested various component mode assignments with observations to propagate variable values. By installing previously generated conflicts, this removed component mode assignments from ever being explored again. This means that when the consistency check is performed, the installed conflicts prune the search space to decrease the number of combinations in the search.

OPSAT is capable of not only determining a set of assignments that are consistent with the constraints, G_x, but can also determine a set of assignments that are inconsistent with the constraints. This is performed by using a complete sat engine as an unsatisfiability engine. This dual use enables the OPSAT algorithm to solve many different types of optimal constraint satisfaction problems.

3.3.4 Dissent Generation as Optimal Constraint Satisfaction

The entailment statement above frames the generation of dissents as a search for sets of observation and component mode assignments that are inconsistent with the constraints. This is framed as an OPSAT problem to generate the dissents. The entailment statement in section 3.3.2 denotes the combination of observation and component mode assignments as an inconsistent set of assignments. The unsatisfiability engine within OPSAT

is used to determine this inconsistency. The Enumeration algorithm is then framed as an OPSAT problem as follows:

$$OPSAT(s) \equiv \langle \bar{y}, f, CSP \rangle$$

$$CSP(s) \equiv \langle \bar{x}, D_x, G_x \rangle$$

where

$\bar{x} \equiv$ observations, component mode and intermediary variables

$D_x \equiv$ the domains of the vector of variables, $\bar{x}$

$G_x \equiv$ the mode constraints to be unsatisfied

$\bar{y} \equiv$ observation and component mode variables

$f \equiv$ sum of the number of assignments in a dissent

$OPSAT(s) \rightarrow$ an assignment to each variable $\bar{y}$

Figure 3-5 - Enumeration Algorithm as OPSAT

The Enumeration algorithm uses the OPSAT unsatisfiability engine to test candidates for inconsistency. Candidates in OPSAT now represent the combinations of observation and component mode assignments. The process of generating the dissents first generates candidates guided by length. This means that singleton candidates are explored and tested first, followed by length two, three and so on. When a candidate is generated, the Enumeration algorithm tests it for inconsistency. If the candidate is inconsistent, then it is identified as a dissent and installed as a conflict in the unsatisfiability engine. This enables the unsatisfiability engine to improve performance as it generates more dissents as described in section 3.3.3. The generated Dissents are also used to prune branches of the conflict directed search by performing a subsumption check whenever a candidate is generated. Installing dissents as conflicts in the unsatisfiability search and subsumption checking guarantee that supersets of dissents are not generated. Since dissents are generated by increasing length, then this guarantees that the minimal set of dissents are generated by the Enumeration algorithm.

The resulting Enumeration algorithm is summarized below.

Enumeration Algorithm(C_M, y, $\mathbf{D}_y$)

1   Create a queue, Q, that maintains the list of *nodes*, where a *node* is made up of a list of   assignments, and the cost, which is the length of the set of assignments
2   Create a list of *dissents*, D that will hold the newly generated dissents
3   Loop **while** Q is *not empty*
4   Extract shortest list of assignments from Q, the *best-node*
5   Test *best-node* for **subsumption** using D
6   **if** *best-node* not subsumed, **then unsat**(*best-node*, C_M) to test for inconsistency
7   **if** *best-node* and C_M are inconsistent, **then** place assignments of *best-node* in D and place
           best-node as a conflict in **unsat**
8   **otherwise**, extend *best-node* as follows
9   **for** a variable, y_i in y, not mentioned in *best-node*
10   **for each** element v_{ij} in $\mathbf{D}_{yi}$ of variable y_i
11   create a *new-node* adding $y_i = v_{ij}$ to *best-node*
12   add *new-node* to Q by length
13   **end for**
14   **end if**
15   **end while**
16   **return** D

The Enumeration algorithm described here attempts all combinations of observation and component mode assignments. This generates all dissents in the system model. Since any diagnosis can be reconstructed from the conflicts in the system, then the Enumeration algorithm compiles the model without loss of information.

3.3.5 Mode Compilation Example

The Enumeration Algorithm is next demonstrated using the NEAR Power Storage system described in Chapter 1. This example focuses on the interactions between the *switch*, *charger-1* and *charger-2*, depicted in Figure 3-6. Notice that the *switch* and *chargers* communicate through the shared variable, *switch-voltage*. It is this variable that compilation removes from the mode constraints.
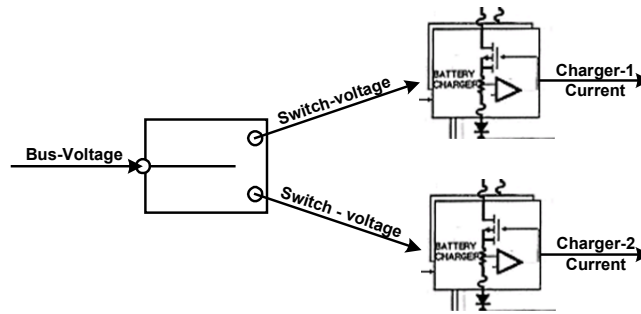
Figure 3-6 - Switch and Redundant Chargers in the NEAR Power Storage System

For the example, the component mode variables to assign to are the *switch.mode, charger-1.mode, charger-2.mode*, and the observables are the *bus-voltage, charger-1.charger-current,* and *charger-2.charger-current.* The domains of each variable are as follows:

     switch.mode   { *charger-1, charger-2, stuck-charger-1, stuck-charger-2* }

       charger-1.mode   { *full-on, trickle, off, broken* }
       charger-2.mode   { *full-on, trickle, off, broken* }
       bus-voltage   { *zero, low, nominal* }
       charger-1.current{ *zero, trickle, nominal* }
       charger-2.current{ *zero, trickle, nominal* }

The Enumeration algorithm can be visualized as a search tree where the first step expands on all assignments, and each expansion that follows depends on which variables have not been assigned. Using the subset of the NEAR Power Storage system described in Figure 3-6, the following depicts the example search tree.
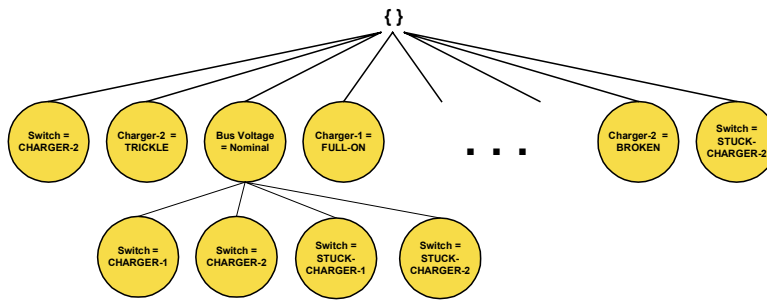
Figure 3-7 - Example Search Tree for Mode Compilation

From the search tree, assume the algorithm follows the path *bus-voltage = nominal* and *switch = charger-1*. This by itself is not a dissent because it is consistent with the model as it predicts that *charger-1.switch-voltage = nominal* and *charger-2.switch-voltage = zero*. The next expansion using the component *charger-1*, several dissents are produced in the unsatisfiability engine.
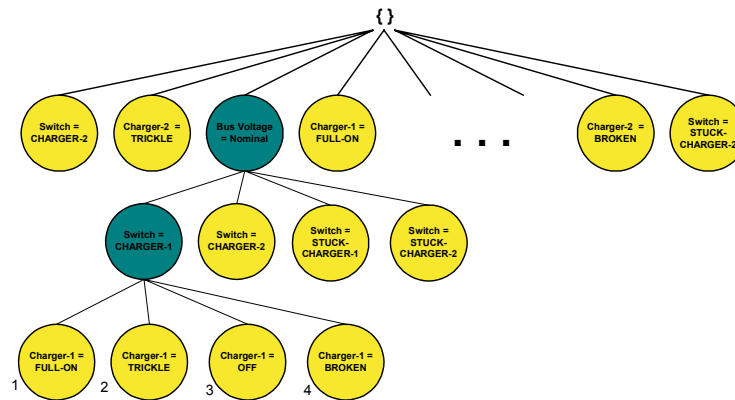
Figure 3-8 - Next Expansion of the Search Tree for Mode Compilation

The different combinations of component mode assignments from this expansion are:

1. (*bus-voltage = nominal*), (*switch = charger-1*), (*charger-1 = full-on*)

2. (*bus-voltage = nominal*), (*switch = charger-1*), (*charger-1 =trickle*)

3. (*bus-voltage = nominal*), (*switch = charger-1*), (*charger-1 = off*)

4. (*bus-voltage = nominal*), (*switch = charger-1*), (*charger-1 = broken*)

The unsatisfiability engine in the Enumeration algorithm determines that for the first candidate *charger-1.switch-voltage = nominal* by propagating the *bus-voltage = nominal* through the constraints for *switch = charger-1*. When propagating using the mode assignment *charger-1 = full-on*, the *switch-voltage* attains the same value. Since there is no discrepancy, candidate 1 is determined to be consistent with the model constraints, and therefore not a dissent. The unsatisfiability continues and tests the third candidate. This results in *charger-1.switch-voltage = zero* using the *charger-1 = off* component mode constraints. The resulting discrepancy is identified by the unsatisfiability engine, and this candidate is then marked as a dissent. The Enumeration algorithm then places this in the list of dissents and continues exploring the search tree for other dissents.

The search and propagation performed here by the unsatisfiability engine is the exponential computation that is removed from the online process. Attempting this many combinations of component modes and observation variables online would render the mode estimation algorithm inoperable in a large system. This determination has been deferred to an offline process so that the exponential computation is avoided at run-time.

3.3.6 Analysis of Mode Compilation and Mini-ME

The mode compilation process described here enables the Mini-ME diagnostic engine to perform diagnosis with fewer computations online.

The Mini-ME engine provides instantaneous mode estimates of the system using current observations. Mini-ME addresses the problems of Sherlock and GDE's exponential computation to determine consistency of mode estimates. Like GDE and Sherlock, Mini-ME is capable of using multiple sources of information to determine a diagnosis of the system. Mini-ME is also capable of ranking these diagnoses using the associated probabilities on component modes, similar to Sherlock. This enables Mini-ME to overcome the problem of diagnostic discrimination in GDE. However, like GDE and Sherlock, Mini-ME is still only capable of providing instantaneous mode estimates of the system. Even though it can diagnose time varying systems, it does not gain diagnostic discrimination of these systems because it does not track the behavior over time. This capability was first introduced in the Livingstone engine with the addition of transitions to the system model. The CME engine also gains this capability because it tracks mode estimates over time.

4 Conflict Based Mode Estimation with Transitions

4.1 Mode Estimation and the Need for Transitions

Tracking mode estimates over time is the next step in developing a fault management system that can handle single and multiple faults, and diagnose complex behaviors of time varying systems. Tracking mode estimates requires a more expressive model and different algorithms to use this new information. A system developed after GDE and Sherlock, the Livingstone diagnostic engine addressed the problem of tracking mode estimates.

The previous diagnostic systems, GDE, Sherlock and Mini-ME limited the expressiveness of the model to contain component modes, constraints on these component modes, and probabilities on these modes. These diagnostic systems are able detect a number of types of instantaneous failures in a system. While GDE, Sherlock and Mini-ME handle novel failures, they require that the symptoms propagate from the failure mode to the observation variables in the same time step, otherwise they are unable to diagnose the failure.

For example, consider the *switch* in the NEAR Power storage system. It has a *charger-1* operational mode and a *stuck-charger-1* failure mode. Each of these modes exhibits the same behavior by passing the input to *charger-1* only. Sherlock and Mini-ME would not be able to differentiate between these two modes. The use of transitions allows components to move between modes, enabling an engine to determine the difference. To discriminate between these two modes, a transition between the modes of the *switch*, *charger-1* and *charger-2* is specified with the constraint that an input command must be given to make the transition. If the command is given to transition from *charger-1* to *charger-2*, then *charger-1* is not a valid component mode in the current time step. So, if the observations support the behavior for the mode *switch = charger-1*, then it must be that the true mode is actually *switch = stuck-charger-1*. Without transitions, this type of reasoning could never occur.

The first system that used behavioral modes and transitions between modes was the Livingstone reactive system [*Williams, 1996*]. Livingstone generates mode estimates similar to Sherlock in a best-first generate and test fashion. The difference is that Livingstone uses the transitions to adjust the component mode probabilities at run-time, whereas these values were static in Sherlock. The Livingstone engine is presented in section 4.3. In order to discuss the mode estimation performed in Livingstone, it is necessary to review the system model and define its elements, which is given in section 4.2.

4.2   System Model Framework

The system model used within Livingstone includes behavioral modes for components, and adds in transitions with an associated probability. The

system model is described as a Concurrent Constraint Automaton (CCA) [*Williams 2, 2002*] that has the following constituents:

1. Discrete modes
2. Model constraints
3. Constraints describing communication between components
4. Probabilistic transitions

The constituents of a CCA create a compact encoding of a Hidden Markov Model (HMM). An HMM is a framework for expressing the hidden state problems for dynamic systems. Mode estimation is an example of this problem since the component modes are not directly observable. The HMM framework offers equations to calculate probabilities of mode estimates, known as belief update.

A CCA's compact encoding builds up the system model using constraint automata, one automata for each component in the system. Concurrency here relates to the operation of constraint automata acting synchronously, as do components in a system. Constraints are used to represent component modes, transitions, and interactions between components. Probabilistic transitions are used to model the stochasticity of component behavior such as failures and intermittent behavior (resettable failures). The following sections give the background for Hidden Markov Models and the standard belief update equations, followed by Concurrent Constraint Automata, and the roles they play in performing mode estimation.

4.2.1 Hidden Markov Models

The theory of Hidden Markov Models [*Elliott, 1995, Williams 2, 2002*] offers an approach to framing the hidden state problem. This section reviews HMMs and gives the standard belief update equations.

An HMM is given by a tuple $\langle \Sigma, \mathbf{O}, P_\theta, P_\mathbf{T}, P_\mathbf{O} \rangle$, where each element is defined as:

- $\Sigma \equiv$ finite set of feasible states, s_i
- $\mathbf{O} \equiv$ finite set of observations, o_i
- $P_\theta \left[s^{(0)} = s_i \right]$ denotes the probability that s_i is the initial state
- $P_\mathbf{T} \left[s_i^{(t)} \mapsto s_j^{(t+1)} \right]$ denotes the conditional probability that $s_j^{(t+1)}$ is the next state given that $s_i^{(t)}$ is the current state.
- $P_\mathbf{O} \left[s_i^{(t)} \mapsto o_k^{(t)} \right]$ denotes the conditional probability that $o_k^{(t)}$ is observed given state $s_i^{(t)}$.

Figure 4-1 - Definitions of a Hidden Markov Model

The elements of a Hidden Markov Model are defined in Figure 4-1, with P_θ known as the initial state function, $P_\mathbf{T}$ the transition function and $P_\mathbf{O}$ the observation function. The set of states, Σ, represents all combinations of component modes in the system. The set of observations, o_i, represents the sensor information in the system. The transition function captures the constraints between modes of a component and the probabilities associated with these transitions. The observation function captures the constraints associated with component modes and the probability that a particular state, $s_i^{(t)}$, predicts the observations, $o^{(t)}$.

An HMM is used to perform belief update. Belief update computes the likelihood of each state, $s_i^{(t)}$, at each time step. Belief update is an incremental process, performed each time observations are made and control actions are given to the system. Belief update computes the likelihood of the current mode estimate using transition probabilities, previous mode estimate probabilities and the current observations and control actions. The equations for this operation are as follows.

$$\sigma^{(\cdot t+1)}\left[s_i\right] \equiv P\left[s_i^{(t+1)} \mid o^{(0)}, \ldots, o^{(t)}, \mu^{(0)}, \ldots, \mu^{(t)}\right]$$

$$\sigma^{(t+1\cdot)}\left[s_i\right] \equiv P\left[s_i^{(t+1)} \mid o^{(0)}, \ldots, o^{(t+1)}, \mu^{(0)}, \ldots, \mu^{(t)}\right]$$

Equation 4-1 - Belief Update Equations for HMMs

Here, $\sigma^{(\bullet t+1)}$ is used to determine an a-priori probability for state s_i that includes observations and control actions up to time 't'. The posterior probability, $\sigma^{(t+1\bullet)}$, adjusts the a-priori calculation to include observations up to time '$t+1$'. This brings the mode estimate up to the time of the latest observations. These calculations are performed for each state, s_i, giving a corresponding $\sigma^{(t+1\bullet)}$. The set of all pairs $\langle s_i, \sigma^{(t+1\bullet)} \rangle$ is known as the belief state.

The Markov property is exploited to compute the belief state at time '$t+1$', using only the control actions at time 't' and the observations at time '$t+1$'. The control actions are assumed implicit in the transition function, $P_\mathbf{T}$. The standard belief update equations are:

$$\sigma^{(\cdot t+1)}\left[s_i\right] \equiv \sum_{j=1}^{n} \sigma^{(t\cdot)}\left[s_j\right] P_\mathbf{T}\left[s_j \mapsto s_i\right]$$

$$\sigma^{(t+1\cdot)}\left[s_i\right] \equiv \sigma^{(\cdot t+1)}\left[s_i\right] \frac{P_\mathbf{O}\left[s_i \mapsto o_k\right]}{\sum_{j=1}^{n} \sigma^{(\cdot t+1)}\left[s_i\right] P_\mathbf{O}\left[s_j \mapsto o_k\right]}$$

Equation 4-2 - Standard Belief Update Equations

These equations express the link between the probabilities in the system model and the probabilities on a state at a specified time. The first equation calculates the a-priori probability of a state by taking the probability of a previous state, s_j, and multiplying it by the probability of transitioning from state s_j to the current state s_i. The total a-priori probability is then given by the sum over all previous states. The posteriori probability is calculated by

updating the a-priori using the observations. The numerator denotes the product of the a-priori probability for state s_i and the probability that it predicts the observations o_k. The denominator is a normalization factor ensuring that the posteriori probability does not exceed 1. The Sherlock equation for calculating probabilities on component modes was derived from the posteriori probability for HMMs.

The belief states and system trajectories can be visualized using a trellis diagram shown in Figure blah. Belief update associates a probability with each state in the figure. Paths through the diagram represent trajectories of the states of the system. The process of mode estimation tracks these trajectories over time to estimate the state of the system.
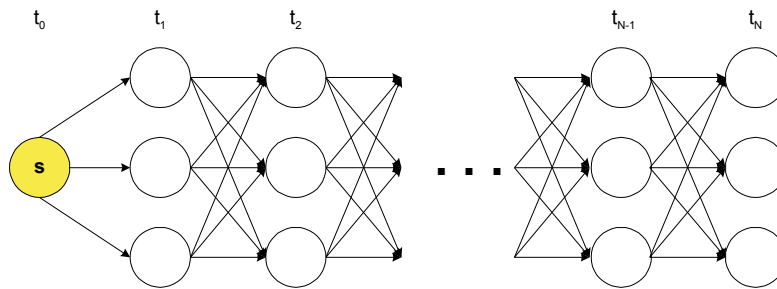
Figure 4-2 - Trellis Diagram

Model-based mode estimation extends the belief update to systems encoded using constraints through the compact encoding of Concurrent Constraint Automata (CCA).

4.2.2 Concurrent Constraint Automata

CCAs used within Livingstone offer a compact encoding of constraints and transitions. The concurrent constraint automata for a system are built up from constraint automata. These constraint automata capture the model of

individual components, including the modes, constraints on these modes, and transitions between modes. The concurrent constraint automata capture the individual constraint automata, and the constraints between these individual automata. This section first develops the definition of a constraint automata followed by the concurrent constraint automata.

4.2.2.1 Constraint Automata

A constraint automaton is characterized by a *mode variable,* with an associated *domain*. Given a *mode variable*, a *mode assignment* is a value from the *domain*, with an associated *constraint*. The constraints are expressed over the *attribute variables* of the automaton. For instance, consider the NEAR Power storage system described in Chapter 1. The battery chargers, *charger-1* and *charger-2*, have attribute variables *switch-voltage, battery-temperature* and *charger-current*. A constraint automaton can change modes as specified by a *transition function*. In constraint automaton, there is a set of specified transitions for each *mode assignment*, each having an associated probability. These *constraints* and *transition function* allow the representation of the behavioral modes of a component including nominal, failure and intermittent operation.

A constraint automaton for a component '*i*' is a tuple
$$\left\langle \Pi_i, M_i, T_i, P_{\theta_i}, P_{T_i}, \mathbb{R}_i \right\rangle \text{ where:}$$

- Π_i is a set of variables for the component where each x in Π_i ranges over a finite domain **D***(x)*. Π_i is partitioned into a singleton set, Π_{im}, containing the component *mode variable*, x_{im}, and a set Π_{ia} of *attribute variables* x_{ia}. The constraints of the component each range over Π_{ia}. The representation of constraints follows the definition of a constraint automaton.

- $M_i : D\left(\Pi_{im}\right) \rightarrow \mathbb{C}\left(\Pi_i\right)$, associates with each mode variable assignment $x_{im} = v_{ij}$ a finite domain constraint $M_i\left(x_{im} = v_{ij}\right) \in \mathbb{C}\left(\Pi_i\right)$. This constraint captures the components behavior in a given mode.

- $\mathbf{T}_i : \mathbf{D}\left(\Pi_{im}\right) \times \mathbb{C}\left(\Pi_i\right) \rightarrow \mathbf{D}\left(\Pi_{im}\right)$ associates with each mode variable assignment $x_{im} = v_{ij}$ a set $\mathbf{T}_i\left(x_{im} = v_{ij}\right)$ of transition functions. Each transition function $\mathbf{T}_i^k\left(x_{im} = v_{ij}\right) \in \mathbf{T}_i\left(x_{im} = v_{ij}\right)$ specifies an assignment to x_{im} at time $t + 1$, given assignments to the variables Π_i, at time t (including $x_{im} = v_{ij}$). The transitions representing nominal behavior are denoted by $\mathbf{T}_i^n\left(x_{im} = v_{ij}\right)$. These transitions allow for transitions to other mode assignments in the component, as well as the same mode assignment, known as the idle transition.

- $P_{\theta_i} : \mathbf{D}\left(\Pi_{im}\right) \rightarrow \Re\left[0,1\right]$ denotes the probability that $x_{im} = v_{ij}$ is the initial mode for component i.

- $P_{\mathbf{T}_i} : \mathbf{T}_i\left(x_{im} = v_{ij}\right) \rightarrow \Re\left[0,1\right]$ denotes for each mode assignment $x_{im} = v_{ij}$, a probability distribution over the possible transition functions $\mathbf{T}_i^k\left(x_{im} = v_{ij}\right)$.

Equation 4-3 - Definition of a Constraint Automata [*Williams 2, 2002*][3]

The definition of a constraint automaton denotes the single mode variable, Π_{m}, and its set of attribute variables, Π_{ia}. These attribute variables can include observation, intermediary and other component mode variables. The constraint automaton also maintains constraints on mode variables and constraints on transitions. In order for a mode estimate to be consistent now requires using the component mode constraints and the constraints on transitions. The definition of the constraint automaton also incorporates the probabilities on transitions in the probability distribution, $P_{\mathbf{T}i}$.

The transition functions are specified on each component mode variable, as denoted by $\mathbf{T}_i(x_{im} = v_{ij})$. Each transition function $\mathbf{T}_i^k(x_{im} = v_{ij})$ is represented as a set of transition pairs $(l_{im},\ v_{in})$. Here, l_{im} is a set of labels on the transition, denoted by c if entailed and $\overline{c}$ if not entailed, where $c \in C(\Pi_i)$. The destination mode of the transition is denoted by v_{in}, where $v_{in} \in \mathbf{D}(x_{im})$. This corresponds to the traditional representation of a transition with labeled arcs in a graph, and is visualized in the following figure.

---
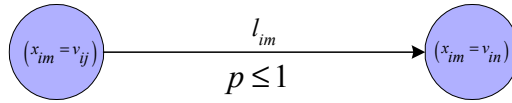
[3] Note that reward is not included here as it is irrelevant to mode estimation.

Figure 4-3 - Representation of a Constraint Automaton Transition

The constraints in the component modes and transitions are expressed using standard propositional logic. Expressions are created using propositions and composed using standard logical connectives. The following specifies the form of these expressions using the Backus-Naur Form (BNF):

$$constraint \rightarrow proposition \mid wff$$

$$proposition \rightarrow TRUE \mid FALSE \mid assignment \mid (NOT\ assignment)$$

$$assignment \rightarrow (variable = value)\ or\ (x_i = v_{ij})$$

$$wff \rightarrow ask\ constraint\ connective\ ask\ constraint$$

$$connective \rightarrow AND \mid OR \mid IMPLIES \mid IFF$$

Figure 4-4 - Propositional Logic Form of a Constraint

This concludes the specification of constraint automata and all of the constituents. The definitions and their uses are best demonstrated by example.

4.2.2.2 Constraint Automaton Example

Consider the *battery-charger* in the NEAR Power Storage system described in Chapter 1. Its inputs are the *switch-voltage* and the *battery-temperature*, and outputs the *charger-current,* all of which are *attribute variables*. The domain of this component is **D**(*battery-charger*) = {*full-on, trickle, off, broken, unknown*}. The *switch-voltage* has the domain {*zero, low, nominal*}, and the *battery-temperature* has the domain {*low, nominal, high*}. The output variable, *charger-current* has the domain {*zero, trickle, nominal, high*}. A figure showing the charger and the charger automata are given below.
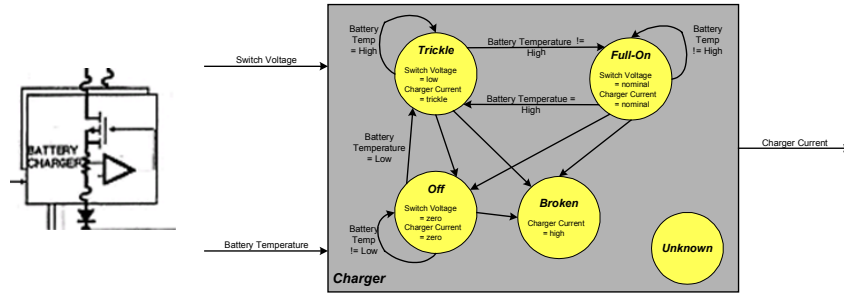
Figure 4-5 - Automaton of the NEAR Power System Charger

The figure denotes the model constraints, M_i as:

$$M_{battery\text{-}charger}(battery\text{-}charger=trickle) = (switch\text{-}voltage=low) \wedge (charger\text{-}current=trickle)$$

$$M_{battery\text{-}charger}(battery\text{-}charger=full\text{-}on) = (switch\text{-}voltage = nominal) \wedge (charger\text{-}current=nominal)$$

$$M_{battery\text{-}charger}(battery\text{-}charger=off) = (switch\text{-}voltage=zero) \wedge (charger\text{-}current=zero)$$

$$M_{battery\text{-}charger}(battery\text{-}charger=broken) = (charger\text{-}current=high)$$

$$M_{battery\text{-}charger}(battery\text{-}charger=unknown) = TRUE$$

The transition function, T_i, is denoted on the figure as the following:

$$T_{battery\text{-}charger}^{n}(battery\text{-}charger=trickle) = \left\{ \begin{array}{l} (battery\text{-}temperature \mathrel{!=} high,\, full\text{-}on\,), \\ (battery\text{-}temperature = high,\, trickle) \\ (TRUE,\, off\,) \end{array} \right\}$$

$$T_{battery\text{-}charger}^{f}(battery\text{-}charger=trickle) = \left\{ \begin{array}{l} (TRUE,\, broken\,) \\ (TRUE,\, unknown\,) \end{array} \right\}$$

$$T_{battery\text{-}charger}^{n}(battery\text{-}charger=full\text{-}on) = \left\{ \begin{array}{l} (battery\text{-}temperature = high,\, trickle\,), \\ (battery\text{-}temperature \mathrel{!=} high,\, full\text{-}on) \\ (TRUE,\, off\,) \end{array} \right\}$$

$$T_{battery\text{-}charger}^{f}(battery\text{-}charger=full\text{-}on) = \left\{ \begin{array}{l} (TRUE,\, broken\,) \\ (TRUE,\, unknown\,) \end{array} \right\}$$

$$T_{battery\text{-}charger}^{n}(battery\text{-}charger=off) = \left\{ \begin{array}{l} (battery\text{-}temperature = low,\, trickle\,), \\ (battery\text{-}temperature \mathrel{!=} low,\, off\,) \end{array} \right\}$$

$$T_{battery\text{-}charger}^{f}\left(battery\text{-}charger{=}off\right)=\left\{\begin{array}{l}(TRUE,broken\ )\\(TRUE,unknown\ )\end{array}\right\}$$

$$T_{battery\text{-}charger}^{f}\left(battery\text{-}charger{=}broken\right)=\left\{\begin{array}{l}(TRUE,unknown\ ),\\(TRUE,broken)\end{array}\right\}$$

$$T_{battery\text{-}charger}^{f}\left(battery\text{-}charger{=}unknown\right)=\left\{(TRUE,unknown\ )\right\}$$

In these transition functions, the probabilities must be specified in order to complete the definition of this constraint automaton. The total probability of enabled transitions out of a component mode must sum to one. This makes the probability on nominal transitions, $\mathbf{T}^{n}_{battery\text{-}charger}$ equal to 0.95, and for fault transitions, $\mathbf{T}^{f}_{battery\text{-}charger}$ equal to 0.04 for transitioning to 'broken', and 0.01 for transitioning to 'unknown'. Not specified here is the probability distribution on initial modes.

4.2.2.3   Concurrent Constraint Automata

Using the foundation of the constraint automaton, the concurrent constraint automata (CCA) definition can now be elaborated. A CCA models the spacecraft system as a group of constraint automata all acting concurrently, executing transitions in a synchronous manner. This group of constraint automata represents the components in the plant, one automata for each component. The framework of the CCA captures the interconnections between the constraint automata and the interactions the plant has with the environment.

A CCA is described by a tuple $\langle A, \Pi, \mathbf{I} \rangle$, where:

- $A = \{A_1, A_2, A_3, ..., A_n\}$ denotes the finite set of constraint automata that are associated with the n components in the plant system.

- Π is a set of *plant variables* where each $x \in \Pi$ ranges over a finite domain $\mathbf{D}(x)$. $\mathbb{C}_{tell}(\Pi)$ denotes the set of finite *tell* constraints over Π. Π is partitioned into sets of *mode variables*, Π_m, *observable variables*, Π_o, *control variables*, Π_c, and *dependent variables*, Π_d.
  - Mode variables, Π_m, represent the different modes of a component in the plant. The set $\Pi_m = \bigcup\{\Pi_{im} \mid i = 1..n\}$ contains all of the mode variables.
  - Observable variables capture the information of the plant sensors. They represent a subset of the *attribute variables* of the set of component constraint automata, A. Formally, $\Pi_o \subset \bigcup\{\Pi_{ia} \mid i = 1..n\}$
  - Control variables provide a way to assert external actions on the plant. Commands to components such as actuators are relayed through these variables. They too represent a subset of the *attribute variables* of the set of component constraint automata, A. Formally, $\Pi_c \subset \bigcup\{\Pi_{ia} \mid i = 1..n\}$ and $\Pi_o \cap \Pi_c = \varnothing$.
  - Dependent variables represent the shared variables between the components in the plant or the interconnections. These are used to propagate effects of the *control variables* and the *observable variables* throughout the plant. They represent another subset of the *attribute variables* of the set of component constraint automata, A. Formally $\Pi_d \subset \bigcup\{\Pi_{ia} \mid i = 1..n\}$ with the condition that $\Pi_d \cap \Pi_c = \varnothing, \Pi_d \cap \Pi_o = \varnothing,$ and $\Pi_d \cup \Pi_c \cup \Pi_o = \bigcup\{\Pi_{ia} \mid i = 1..n\}$.

  - The state space of Π, denoted $\mathbf{D}_\Pi$, is the cross product of the $\mathbf{D}(x)$ for all variables $x \in \Pi$. The state space of the plant component modes, Π_{im}, is then $\mathbf{D}_{\Pi_{im}} = \mathbf{D}(x_{im}) \times \mathbf{D}(x_{im})$ for all variables $x_{im} \in \Pi_{im}$. A state snapshot, $s^{(t)}$, of the plant components at time t is then an assignment to all mode variables $x_{im} \in \Pi_{im}$ a value from their domain, $\mathbf{D}(x_{im})$.

- $\mathbf{I} \in \mathbb{C}(\Pi_c \cup \Pi_o \cup \Pi_d)$ is a conjunction of constraints modeling the interconnections between the *attribute variables* of the set of constraint automata, A.

Equation 4-4 - Definition of a Concurrent Constraint Automaton [*Williams 2, 2002*]

Using this definition of a CCA, it is now possible to describe multiple components and characterize their interactions via the intermediate variables. The sensor and control information can be brought into the component model to incorporate these constraints. The following example shows the use of these definitions of a CCA.

Consider the NEAR Power Storage system from Chapter 1, focusing on the *switch* and the *charger* depicted in Figure 4-6. The component models are simplified to decrease the number of modes since only the interactions between the *switch* and one *charger* are considered for this example. The

switch has the domain {*charger-1, off, broken*}, and the *charger* has the domain of {*full-on, trickle, off, broken*}. The attribute variables of the *switch* are the inputs *switch-cmd* and *bus-voltage*, and the output *switch-voltage*. The attribute variables of the *charger* are the input *switch-voltage* and the output *charger-current*.
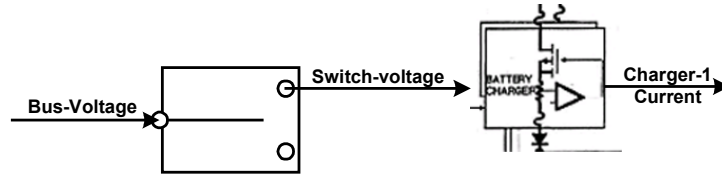
Figure 4-6 - Switch and Battery Charger from the NEAR Power Subsystem

The automata for the *switch* component is shown below and would be expressed in the set, *A*, of constraint automata for the CCA. Recall the automata in Figure 4-5 of a general charger and consider it without the *unknown* mode. The constraints on each mode are also shown along with the constraints on the transitions.
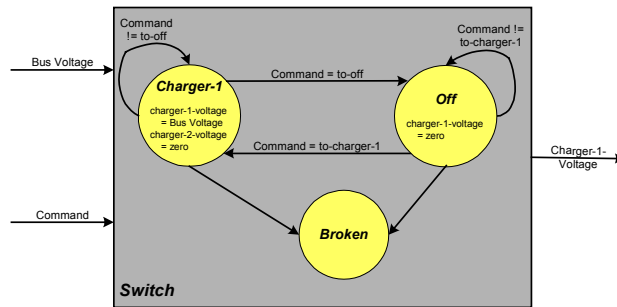
Figure 4-7 - Constraint Automaton for a Switch

The components communicate through the shared variable *switch-voltage*, therefore this is the only member of the set Π_d, with Π_d = {*switch.switch-voltage, charger-1.switch-voltage*}. The control variables in this example are represented by the command to the switch, with Π_c = {*switch-cmd*}.

The observable variables are noted as Π_o = {*bus-voltage, charger-1-current*}.

The interconnection between the *switch-voltage* of the *switch* and the *switch-voltage* of the *charger* is then described by the set: **I** = {

(switch.switch-voltage = low) IFF (charger-1.switch-voltage = low)
(switch.switch-voltage = nominal) IFF (charger-1.switch-voltage = nominal)
(switch.switch-voltage = zero) IFF (charger-1.switch-voltage = zero) }

This example demonstrates the use of the different elements of a CCA. Once the constraint automata have been specified, then the links between these automata can be made using the framework of a CCA and the interconnection constraints, **I**.

4.2.2.4   CCA's and Mode Estimation

The remaining portion of the CCA specification is to detail the execution of concurrent constraint automata properly to determine mode estimates. Recalling the trellis diagram of Figure 4-2, identifying mode estimates is then the process of selecting a trajectory through the trellis diagram to arrive at a particular mode estimate. This trajectory is constrained to be consistent with the transitions, the model constraints of the CCA and the current observations. The task of mode estimation is to determine the likely trajectories through the trellis diagram using the probabilities on the transitions to guide the choice of the trajectory. The choice is guided by the belief update equations of HMMs applied to CCAs.

A CCA, while a compact encoding of an HMM, makes explicit certain structural properties left out of the definition of an HMM. The observation and transition functions are not explicitly defined in an HMM but are

defined in a CCA. The transition function of a CCA is given by the individual transition functions of the constraint automata. The observation function is implicit in the mode constraints of the individual constraint automata and in the constraints between automata in the CCA. Additionally, a CCA is concurrent, denoting that all components make a transition at each time step, which is also not expressed in an HMM.

What remains is to define the probabilities associated with the transition and observation functions to be used in the belief update equations. The constraints expressed in a CCA and the transitions divide the space of mode estimates into feasible and infeasible sets. Mode estimation uses the constraints and transitions to determine the feasible mode estimates, and constrain the probability of any infeasible mode estimate to be zero. The definitions of P_T and P_O for CCA must capture this.

To define the transition function probability, recall that a plant transition **T** for a state 's_k' of a CCA is comprised of a set of component transitions, one for each component mode assignment in the state. Using the individual component transition probabilities $P_T(x_{im} = v_{ij})$, calculating P_T then only requires determining the product of these individual transitions with the key assumption that component mode transitions are independent of one another, given the current state, 's_k'. The equation to calculate P_T is given as follows:

$$P_{\mathbf{T}}\left(s_k\right) = \prod_{\left(x_i = v_{ij}\right) \in s_k} P_{\mathbf{T}_i}\left(x_i = v_{ij}\right)$$

Equation 4-5 - Calculation of the Transition Function Probability

The next step is to define the observation function, P_O. The calculation of the observation probabilities is performed using the constraints on the state, 's_k'. These constraints are built up from the individual component

constraints $M_i(x_{im} = v_{ij})$ of each mode assignment in 's_k'. If an observation is entailed by the constraints and the mode estimate, then $P_O = 1$. If an observation is refuted, or not entailed, then $P_O = 0$. In the case that entailment of an observation cannot be determined, the observation is neither entailed nor refuted. One approach to assume a uniform prior probability and set $P_O = 1/n$, where 'n' is the number of different values in the domain of the observation. GDE was the first to develop and use this approach to calculating the observation function, and this same approach is used in Sherlock and Mini-ME

The definitions for P_T and P_O enable a mode estimation algorithm for CCA that uses the standard belief update equations. The algorithm takes as an input the model of the system expressed as a CCA, a set of previous mode estimates, $B^{(t)}$, which are the pair $\langle s_i^{(t)}\ \sigma^{(t\bullet)}\rangle$, the commands, $\mu^{(t)}$, and current observations, $o^{(t+1)}$. ME-CCA returns the current set of mode estimates, $B^{(t+1)}$, which are the pair $\langle s_j^{(t+1)}, \sigma^{(t+1\bullet)}\rangle$. The steps of the mode estimation algorithm for CCA (ME-CCA) are given below in words, followed by a detailed mathematical expression.

1. Identify the constraints $C_{Mi}^{(t)}$ associated with each state $s_i^{(t)} \in S^{(t)}$
2. For each state $s_i^{(t)} \in S^{(t)}$, build the states $s_j^{(t+1)}$ using the transition function $P_T[s_i \rightarrow s_j]$, and take their union
    a. For each mode assignment m_{ik} in $s_i^{(t)}$
        i. identify the transitions enabled by the constraints $C_{Mi}^{(t)}$
        ii. add the targets of each enabled transition to the set of reachable next assignments, $N(m_{jk}^{(t)})$.
    b. Using the sets $N(m_{jk}^{(t)})$, create all possible next states, $s_j^{(t+1)}$, by taking the cross product of the $N(m_{jk}^{(t)})$, for all $m_{ij} \in s_i^{(t)}$, and calculate P_T as specified by Equation 4-5

3. For each state $s_j^{(t+1)}$, calculate the a-priori probability by summing over the previous mode estimates, $s_i^{(t)}$, the posteriori probability $\sigma^{(t\bullet)}[s_i] \bullet P_\mathbf{T}[s_i \rightarrow s_j]$.

4. Extract the constraints $C_{\mathrm{Mj}}^{(t+1)}$ for each state $s_j^{(t+1)} \in S^{(t+1)}$

5. Determine the consistent states, $s_r^{(t+1)}$, using the current observations $o^{(t+1)}$ and the constraints $C_{\mathrm{Mj}}^{(t+1)}$, determining $P_\mathbf{O}[s_r^{(t+1)} \rightarrow o_i^{(t+1)}]$ in the process

6. Calculate the posterior probability of each consistent state, $s_r^{(t+1)}$, using the standard belief update equation and $P_\mathbf{O}$ from step 5

7. Return the set of pairs $\langle s_r^{(t+1)}, \sigma^{(t+1\bullet)} \rangle$

The pedagogical ME-CCA algorithm given above calculates mode estimates in a brute force approach by first generating all reachable states using the transition function and previous mode estimates. The algorithm then determines if a state is consistent with the observations and model constraints. If a state is consistent, then the observation function probability is calculated. If it is not, the state is marked as inconsistent and is associated the value $P_\mathbf{O} = 0$. The final step of the ME-CCA algorithm is to calculate the posteriori probability on the states using the belief update equation.

4.2.2.4.1 ME-CCA Example

The steps of this algorithm are demonstrated using the NEAR Power storage system, in particular the *switch* and *charger* combination detailed in section 4.2.2.3. Considering the following inputs for the ME-CCA algorithm:

$S^{(t)} = \{ switch = charger\text{-}1, charger\text{-}1 = trickle \}$

$\sigma^{(t\bullet)} = 1.0$

$\mu^{(t)} = switch.cmd = to\text{-}off$

$$o^{(t+1)} = \{\textit{bus-voltage} = \textit{zero}, \textit{charger-1.current} = \textit{zero}, \textit{battery-}$$
temperature = nominal}

Applying the first step of the ME-CCA algorithm extracts the constraints on the modes *switch = charger-1* resulting in $C_{Mi}^{(t)} = \{$ *charger-1.voltage = bus-voltage* $\}$, and for *charger-1 = trickle*, $C_{Mi}^{(t)} = \{$ *switch-voltage = low* and *charger-1.charger-current = trickle* $\}$.

These constraints and the commands are used to determine the enabled transitions. The command *switch.cmd = to-off* results in the transition *switch = charger-1* to *switch = off* with a probability of 0.99. ME-CCA identifies the transitions for the *charger* from *trickle* to *off, broken* and *trickle* because of the idle transition. The calculations of step 2 of the informal ME-CCA algorithm result in the following set of component modes each with an associated probability.

$N(m_{jk}^{(t+1)}) = \{$ ⟨*switch = off*, p = 0.99⟩, ⟨*switch = broken*, p = 0.01⟩, ⟨*charger-1 = trickle*, p = 0.95⟩, ⟨*charger-1 = off*, p = 0.04⟩, ⟨*charger-1 = broken*, p = 0.01⟩.

The second phase of step 2 would generate all combinations of the component mode assignments and calculate their transition probabilities. For brevity, not all are detailed here, but a few are:

$\qquad$ { *switch = off, charger-1 = trickle*, p = 0.9405 }
$\qquad$ { *switch = off, charger-1 = off*, p = 0.0396 }
$\qquad$ { *switch = broken, charger-1 = off*, p = 0.0004 }

The third step of the ME-CCA algorithm determines the apriori probability for each state generated in the previous step. Since there is only one previous mode estimate with a probability of 1.0, then the probability calculated by step 2 is unchanged.

The fourth step of the ME-CCA algorithm extracts the constraints on the different states generated in step 2. This requires extracting the constraints on all of the different component modes within the states. Not all are listed here for brevity. The first constraint is from the system model constraints constraining the output of the *switch* to be equal to the input of *charger-1*.

$C_{Mj}^{(t+1)}$ (*switch-charger CCA*) = { *switch.charger-1-voltage = charger-1.switch-voltage* }

$C_{Mj}^{(t+1)}$ (*switch = off*) = { *charger-1.voltage = zero* }

$C_{Mj}^{(t+1)}$ (*switch = broken*) = { }

$C_{Mj}^{(t+1)}$ (*charger-1 = off*) = { *charger-1.switch-voltage = zero, charger-1.current = zero* }

The fifth step of ME-CCA now determines the states that are consistent with the observations and the system constraints. This computation results in the consistent states and their associated observation function probabilities. A few of the states generated by this step are given below.

{ *switch = off, charger-1 = off* , p = 1 }

{ *switch = off, charger-1 = broken*, p = 1/3 }

{ *switch = off, charger-1 = trickle*, p = 0.0 }

These probabilities are used in determining the posteriori probability calculation as defined by the standard belief update equations. This calculation results in the following probabilities for the states listed above. The first two mode estimates are returned from the ME-CCA algorithm, along with the remaining mode estimates not listed here that are also consistent. The final mode estimate listed here is not returned since it has a zero probability, and is thus labeled as an inconsistent mode estimate.

{ *switch = off, charger-1 = off* , p = 0.912 }

{ *switch = off, charger-1 = broken*, p = 0.076 }

{ *switch = off, charger-1 = trickle*, p = 0.0 }

The formal statement of the ME-CCA algorithm is given in this section. The inputs to the ME-CCA algorithm are denoted by P as the system model, $S^{(t)}$ as the previous states, with an associated posteriori probability given by $\sigma^{(t\bullet)}$, $\mu^{(t)}$ as the control actions, and the current observations given by $o^{(t+1)}$. The output belief state of the algorithm is denoted by $S^{(t+1)}$ as the state, and $\sigma^{(t+1\bullet)}$ as the associated posteriori probability.

$$ME - CCA\left(P, S^{(t)}, \sigma^{(t\bullet)}, \mu^{(t)}, o^{(t+1)}\right) \rightarrow \left(S^{(t+1)}, \sigma^{(t+1\bullet)}\right) ::$$

$$1.\, M1 := \left\{\left\langle s_i^{(t)}, C_{Mi}^{(t)}\right\rangle \mid s_i^{(t)} \in S^{(t)}, C_{Mi}^{(t)} = \bigwedge_{(x_{km}=v_{kl})\in s_i^{(t)}} M_k\left(x_{km} = v_{kl}\right)\right\}$$

$$2.\, M2 := \left\{\left\langle s_i^{(t)}, s_j^{(t+1)}, p_{ij}\right\rangle \mid \left\langle s_i^{(t)}, C_{Mi}^{(t)}\right\rangle \in M1,\right.$$
$$\left.\left\langle s_j^{(t+1)}, p_{ij}\right\rangle \in \overline{T}\left(s_i^{(t)}, C_{Mi}^{(t)}, \mu^{(t)}\right)\right\}$$

$$3.\, M3 := \left\{\left\langle s_j^{(t+1)}, p_j\right\rangle \mid \left\langle s_i^{(t)}, s_j^{(t+1)}, p_{ij}\right\rangle \in M2,\, p_j = \sum_{s_i^{(t)}} \sigma^{(t\bullet)}\left[s_i^{(t)}\right] p_{ij}\right\}$$

$$4.\, M4 := \left\{\left\langle s_j^{(t+1)}, p_j, C_{Mj}^{(t+1)}\right\rangle \mid \left\langle s_j^{(t+1)}, p_j\right\rangle \in M3,\right.$$
$$\left. C_{Mj}^{(t+1)} = \bigwedge_{(x_{km}=v_{kl})\in s_j^{(t+1)}} M_k\left(x_{km} = v_{kl}\right)\right\}$$

$$5.\, M5 := \left\{\left\langle s_r^{(t+1)}, p_j, P_{\mathbf{O}}\right\rangle \mid \left\langle s_j^{(t+1)}, p_j, C_{Mj}^{(t+1)}\right\rangle \in M4,\right.$$
$$\left. s_j^{(t+1)} \wedge C_{Mj}^{(t+1)} \wedge o^{(t+1)} \text{ is consistent, then } s_r^{(t+1)} = s_j^{(t+1)}\right\}$$

$$6.\, M6 := \left\{\left\langle s_r^{(t+1)}, \sigma^{(t+1\bullet)}\right\rangle \mid \left\langle s_j^{(t+1)}, p_j, P_{\mathbf{O}}\right\rangle \in M5\right\}$$

$$7.\, return\ M6$$

The function **T** used in step 2 of ME-CCA performs the operations outlined in parts 2a and 2b of the informal algorithm.   More precisely, $\overline{T}\left(s_i^{(t)}, C_{Mi}^{(t)}, \mu^{(t)}\right)$ computes the following:

- For each mode variable assignment $m \in s_i^{(t)}$, and for each transition function $\mathbf{T}_i^k (m) \in \mathbf{T}_i (m)$:
  - identify the transition pair (l_{im}, v_{in}) that is *enabled* by $C_{Mi}^{(t)}$ and $\mu^{(t)}$
  - add the pair $\left(x_{im} = v_{in}, P_{\mathbf{T}_i^k}(m) \right)$ to *list*(m)
- Let $TP = \Pi_{m \in s_i^{(t)}} list(m)$. This cross product gives the full set of possible destination states for a given $s_i^{(t)}$, and assigns each a probability.
- For each $tp_j = \left\langle \left(x_{1m} = v_{1l_1}, p_1 \right), ..., \left(x_{nm} = v_{nl_n}, p_n \right) \right\rangle \in TP$, define:
  - $s_j^{(t+1)} = \left\langle \left(x_{1m} = v_{1l_1} \right), ..., \left(x_{nm} = v_{nl_n} \right) \right\rangle$
  - $p_{ij} = \Pi_{k=1..n} p_k$
- Return $\cup_j \left\langle s_j^{(t+1)}, p_{ij} \right\rangle$

Figure 4-8 - Mode Estimation Algorithm for CCA (ME-CCA) [*Williams 2, 2002*]

4.3 Livingstone

The next step in model-based mode estimation after GDE and Sherlock is the Livingstone engine [*Williams, 1996*]. Livingstone uses the framework of CCA and builds upon the conflict based algorithms of Sherlock to produce a mode estimation engine capable of tracking mode estimates over time. To characterize Livingstone as solely a diagnostic engine is inaccurate. Livingstone was developed to provide mode estimates and use these mode estimates to determine control and repair actions to achieve goals. The architecture of Livingstone is similar to the architecture of a model-based executive presented in Chapter 1. The Livingstone system was validated on the Deep Space 1 spacecraft in 1999.

The pedagogical ME-CCA algorithm presented in the previous section in not practical for systems with large numbers of components due to the large belief state, which grows exponentially with the number of components in the system. Livingstone approximates the belief state by tracking the most likely trajectories in the trellis diagram in Figure 2-2. Livingstone builds

upon the algorithm developed by Sherlock, generate and test, where conflicts are incrementally generated, and then a search determines the smallest set of component mode assignments that satisfies these conflicts. The addition of transitions enables the generation of conflicts to be more focused than in Sherlock. The price is that now Livingstone must determine if a transition is enabled. This computation requires a satisfiability computation using the constraints on transitions. This is exponential in the number of trajectories tracked. Since Livingstone maintains a similar method to testing a candidate as Sherlock, it incurs the same penalty in the satisfiability phase. To avoid further computational problems, Livingstone limits the trajectories tracked at each time step to only a single mode estimate. In order to avoid this limitation, Compiled Mode Estimation seeks to compile the transitions from the CCA to remove the need for full satisfiability. This compilation process is presented in section 5.4. The study of the Livingstone engine gives an approach to generating mode estimates using the transitions. This approach is also used within the CME engine to generate mode estimates online.

This section focuses on the mode estimation process of the Livingstone engine by first presenting the architecture of the mode estimation engine and discussing its inputs and outputs. Section 4.3.2 discusses the process of mode estimation in Livingstone and concludes with a mapping of the steps of Livingstone to the ME-CCA algorithm presented in section 4.2.2.4. The final section discusses the limitations of Livingstone.

4.3.1 Livingstone Inputs and Outputs

Livingstone determines mode estimates by identifying conflicts with a candidate, the system model and the observations. It then resolves the conflict by assigning different component modes to the candidate. The

search for component modes is guided by the probabilities of the transitions. Livingstone builds upon the Sherlock architecture, with the addition of a processs that determines if transitions are enabled. The resultant architecture is shown below:
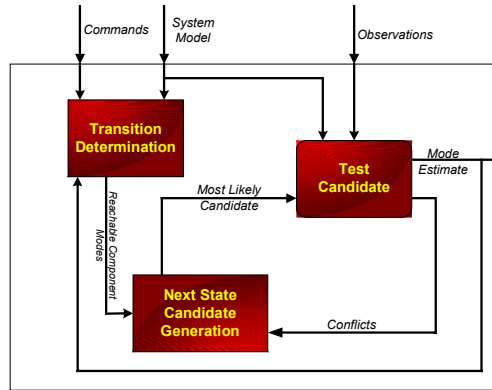
Figure 4-9 - Architecture of the Livingstone Mode Estimation Engine

The Livingstone mode estimation architecture draws on the architecture of Sherlock and its 'test candidate' and 'conflict directed search' loop. It adds a function called 'Transition Determination' that determines the reachable component modes. This step is similar to that performed in step 2 of the ME-CCA algorithm in Section 4.2.2.4.2. The 'transition determination' function maps the current commands and the system model to a set of reachable component modes.

The system model representation used by Livingstone is a CCA. The commands represent an assignment, v_{ij}, to each control variable, $x_{ic} \in \Pi_c$ within the system model. Similarly the observations represent an assignment, v_{ij}, to each observation variable $x_{io} \in \Pi_o$ in the system model.

The output of the Livingstone engine is a set of most likely mode estimates. A mode estimate is the pair of a state and the probability of that state. The

assignments in a mode estimate must be consistent with the current observations, commands and model constraints. Livingstone chooses the best mode estimate to track in the next time increment.

The internal variables in Livingstone are the reachable component modes, the conflicts and the most likely candidate. The set of reachable component modes is defined as the set of pairs of a component mode that is the target of an enabled transition, and the associated transition probability, $\langle m_{ik}^{(t+1)},$ $p_{ik}\rangle$. The conflicts maintain the same definition as that used in GDE and Sherlock, that is, a representation of infeasible component mode assignments. Livingstone limits the conflict directed search to produce only a single most likely candidate. This most likely candidate is represented as a partial set of component mode assignments. This enables Livingstone to incrementally generate the conflicts. The definitions of the inputs, outputs and internals of the Livingstone-ME engine are:

System Model $\equiv CCA$

$$\text{Mode Estimate } (ME_i) \equiv \left\langle s_i^{(t)} = \left\{ \left(x_{1m} = v_{1l_1} \right), \ldots, \left(x_{nm} = v_{nl_n} \right) \right\}, \sigma^{(t+1\bullet)} \right\rangle \forall x_{im} \in \Pi_m$$

$$\text{where } ME_i \wedge C_{Mi}^{(t+1)} \wedge o^{(t+1)} \text{ is consistent}$$

$$\text{Reachable Component Modes} \equiv \left\{ \left\langle \left(x_{1m} = v_{1l_1} \right), p_{1l_1} \right\rangle, \left\langle \left(x_{1m} = v_{2l_2} \right), p_{1l_2} \right\rangle, \ldots, \left\langle \left(x_{nm} = v_{nl_n} \right), p_{nl_n} \right\rangle \right\}$$

$$\text{where } x_{im} \in \Pi_m \text{ and } p_{ij} \text{ denotes the probability of the mode.}$$

$$\text{Most Likely Candidate} \equiv \left\langle \left(x_{1m} = v_{1l_1} \right), \ldots, \left(x_{nm} = v_{nml_n} \right) \right\rangle \text{ where } x_{im} \in \Pi_m \text{ denotes a full set of}$$

$$\text{component mode assignments satisfying all known } conflicts$$

4.3.2 Mode Estimation in Livingstone

The overall process of mode estimation in Livingstone is best described as choosing the best transition from the previous mode estimate to a current,

consistent mode estimate. A depiction of the Livingstone calculation is shown in Figure 4-10.
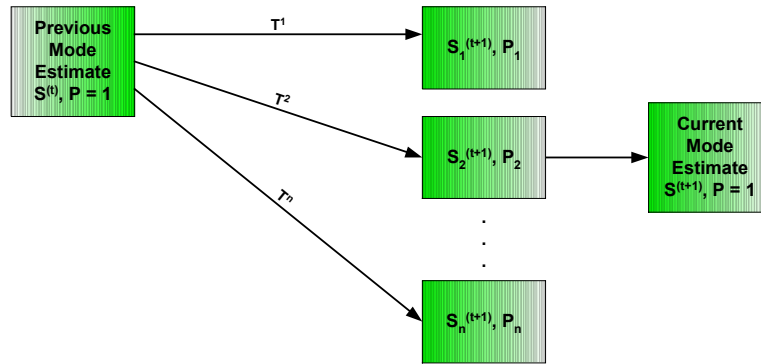
Figure 4-10 - Mode Estimate Calculation in Livingstone

Although Livingstone does not explicitly enumerate the reachable mode estimates, it does enumerate the reachable modes of the individual components. To achieve this, Livingstone first determines if the constraints on the transitions are satisfied, which requires full satisfiability. However, the "causal nature of the constraints of the system model enable full satisfiability to require little search" [*Williams, 1996*]. This statement relates to a simplification in the constraints on transitions, where only the commands and the previous mode estimate are enough to determine the reachable next modes. The result is a simpler search to determine transition consistency as now the transition system of the components are deterministic.

The assumption of a single previous mode estimate enables Livingstone to simplify the calculations of the probabilities of the reachable component modes. Recall in the ME-CCA algorithm that probabilities are specified on mode estimates, given by the standard belief update equations. In the apriori probability of Equation 4-2, the transition probability between mode estimates is multiplied with the product of the previous mode estimate.

However, since there is only a single mode estimate, the apriori probability of a mode estimate is then just the transition probability. The steps of 'transition determination' in Livingstone are summarized below:

1. **for** a mode assignment, $m_{ik}^{(t)}$ in the previous mode estimate $S_i^{(t)}$

   a. **for each** transition, $\mathbf{T}_i^k(m_{ik}^{(t)})$, determine if its constraint is consistent with $S_i^{(t)}$ and the commands, $\mu^{(t)}$

   b. **if** the transition, $\mathbf{T}_i^k(m_{ik}^{(t)})$ is enabled, **then** add its target to the list $N(m_{jk}^{(t+1)})$ of reachable component modes with the associated transition probability, p_T.

2. **return** $N(m_{jk}^{(t+1)})$

Next, recall that the transition probability between a previous and a current mode estimate is comprised of the individual component transitions (Equation 4-1). In order to determine the likely transitions from the previous mode estimate, Livingstone uses the probabilities on the individual component modes to focus in on likely candidates. Instead of constructing all possible mode estimates using the reachable component modes, Livingstone incrementally generates the likely trajectories from the previous mode estimate, guided by the conflicts in the 'test candidate' and 'conflict-directed search' loop, similar to Sherlock. However, since only a single most likely candidate is generated each time, the loop used within Livingstone is known as a Conflict-Directed A* (CDA*).

The CDA* algorithm incrementally generates solutions using as inputs the reachable component modes and their associated transition probabilities, denoted as X, the component mode constraints, denoted as C, and an optimization function, f, defined to be the product of the transition

probabilities. CDA* seeks to maximize *f*, thereby maximizing the probability of the mode estimate. The algorithm is stated below:

CDA* (X, C, f)
    Agenda = { {best-solution(*X*)} }; *Result* = ∅;
    while *Agenda* is *not empty* do
        Soln = pop(*Agenda*)
        if *Soln* satisfies C then
            Add *Soln* to *Result* ;
            if enough solutions have been found then
                return *Result* ;
            else *Succs* = immediate successors *Soln* ;
        else
            Conf = a conflict that subsumes *Soln* ;
            Succs = immediate successors of *Soln* not subsumed by Conf
        endif
        Insert each solution in *Succs* into *Agenda* in decreasing order of f ;
    endwhile
    return *Result*
end CBFS

The algorithm above generates mode estimates by maintaining an *Agenda* of unprocessed candidates. The first step is to remove the most likely candidate from the *Agenda* and test if it is a *Soln*. The test for consistency of the *Soln* using the constraints, *C*, returns true if it is consistent, or returns conflicts if it is not. If *Soln* is consistent with the constraints, then *Soln* is added to the *Result*. If the *Soln* is not consistent, then the conflict returned from the satisfiability engine is stored and used to generate successors, *Succs*, that satisfy the conflict. The conflict returned is a subset of the assignments in *Soln*. This focuses the CBFS by pruning infeasible combinations of component mode assignments. The *Succs* are candidates that are not supersets of any of the conflicts in *Conf*. The CBFS algorithm then places *Succs* in order of decreasing *f* in the *Agenda* and continues to test another possible *Soln*. The CBFS algorithm stops only when the *Agenda* is empty, denoting that all possible mode estimates have been explored, or when some stopping condition has been met. Livingstone specified this halting condition similar to Sherlock where it terminated when a certain percentage of the probability space had been explored.

The CDA* algorithm is capable of generating many solutions, representing the mode estimates, using the constraints of the system model and observations. However on DS1, Livingstone only maintained the most likely mode estimate due to the expensive computations of tracking multiple mode estimates at each time step and extreme limitations of the flight processor.

4.3.2.1   Mode Estimation Example

The Livingstone process of mode estimation is best demonstrated by example. Consider as a simple example the NEAR Power Storage system described in Chapter 1. Focusing on the *switch* and *chargers*, assume the following for the previous mode estimate and observations:

$S^{(t)}$: {*switch = charger-1*, *charger-1 = trickle*, *charger-2 = zero* }

$o^{(t+1)}$: {*bus-voltage = nominal*, *chager-1.current = nominal*, *charger-2.current = zero*}

$\mu^{(t)}$: { *switch.cmd = no-command* }

The 'transition determination' function results in the following reachable component modes:

⟨*switch = charger-1*, p = 0.99⟩, ⟨*switch = broken*, p = 0.01⟩
⟨*charger-1 = trickle*, p = 0.49⟩, ⟨*charger-1 = full-on*, p = 0.49⟩, ⟨*charger-1 = broken*, p = 0.02⟩
⟨*charger-2 = trickle*, p = 0.49⟩, ⟨*charger-2 = off*, p = 0.49⟩, ⟨*charger-2 = broken*, p = 0.02⟩

These component modes are used within the CDA* algorithm of Livingstone to determine the most likely mode estimates. Beginning with an empty agenda, CDA* would choose the most likely assignment for each component in the system as this maximizes the probability function:

switch = charger-1, *charger-1 = trickle*, *charger-2 = trickle*

CDA* then calls the satisfiability engine to test this candidate, and returns a single conflict, which results in the following:

$$\neg[\ switch = charger\text{-}2 \wedge charger\text{-}1 = trickle\ ]$$

In addition, the algorithm determines many more conflicts of the system and the assignments. These are a few that are generated from the compilation process.

$$\neg[\ switch = charger\text{-}1 \wedge charger\text{-}2 = trickle\ ]$$

$$\neg[\ charger\text{-}1 = trickle\ ]$$

These conflicts focus the CDA* search for candidates and successors. The conflicts relay the fact that it is infeasible for both the *switch* to be in the mode *charger-2* and that the *charger-1* be in the *trickle* mode. Similar reasoning applies for the second conflict. In order to determine the most likely candidate that resolves these conflicts, the CBFS performs a search by expanding the conflicts above. The resultant expansion of the first conflict is shown below:
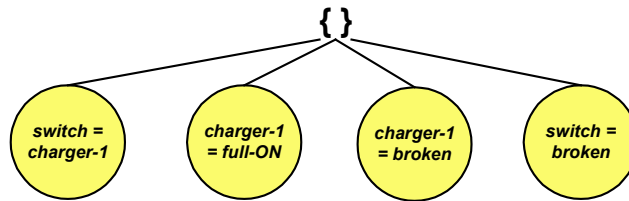

Figure 4-11 - Expansion of Conflicts in Livingstone

Choosing any assignment in this expansion resolves the first conflict. Subsequent expansions on the remaining conflicts results in the following candidate, or successor:

{ *switch = charger-1, charger-1 = full-on, charger-2 = off* } with a probability of p = 0.238

This candidate is then tested again by the satisfiability engine for consistency with the system model and the observations. If no more conflicts are generated as a result of this candidate, then it is stored in the *Result* and the CBFS continues to generate mode estimates.

4.3.2.2   *Livingstone Diagnosis and ME-CCA*

The incremental generation of a diagnosis can be related back to the steps of ME-CCA mode estimation outlined in Figure 4-8. The process of ME-CCA is a brute force approach to generating mode estimates while tracking multiple trajectories, but can give no performance guarantees since both transition enablement and consistency are exponential computations, in the worst case. Livingstone leverages the conflict direction algorithms of GDE and Sherlock, and simplifies the tracking of mode estimates to a single mode estimate to reduce the computations necessary to compute mode estimates. The correspondence of Livingstone to the ME-CCA algorithm is given below:

Step 1:
  a. ME-CCA extracts constraints, $C_{Mi}^{(t)}$, from state $s^{(t)}$
  b. Livingstone extracts constraints, $C_{Mi}^{(t)}$, from the previous mode estimate
Step 2:
  a. ME-CCA calculates all next states, $s_j^{(t+1)}$ using the transition function $\mathbf{T}_i(s_i \rightarrow s_j)$
  b. Livingstone calculate the reachable component modes, N(mik(t+1))
Step 3:
  a. ME-CCA calculates $s_j^{(t+1)}$ probabilities using posterior probabilities of $s_i^{(t)}$
  b. Livingstone does not calculate this since only one previous mode estimate is tracked
Step 4:
  a. ME-CCA extracts the constraints $C_{Mi}^{(t+1)}$ from the states $s_j^{(t+1)}$
  b. CBFS uses the constraints, C relating to the reachable component modes.
Step 5:

a.　ME-CCA prunes the states $s_j^{(t+1)}$ that are inconsistent with the observations, $o^{(t+1)}$ and the constraints, $C_{Mi}^{(t+1)}$

　　　　b.　Livingstone performs this step incrementally through the use of conflicts as described in the CBFS algorithm.

　Step 6:

　　　　a.　ME-CCA combines all states $s_j^{(t+1)}$ that are the same state

　　　　b.　Livingstone does not calculate this since no mode estimate generated is identical to another.

4.3.3　Analysis of Livingstone

The Livingstone engine was the first to incorporate transitions into the system model and use them to perform mode estimation. Transitions give the ability to track behaviors over time and diagnose intermittent failures. The price is that in order to determine if transitions are enabled, full satisfiability must be performed. Livingstone avoided full satisfiability by restricting the guards of the transitions to only the command and component mode assignments in the system. However, transitions of component modes do not have to be restricted to these. A CCA allows for the transitions to be expressed over any combination of attribute variables, which contain control, component mode and intermediate variables. The CME engine allows for transitions to be specified in this manner, but removes the need for full satisfiability by compiling the transitions in an offline process. This enables CME to enhance the mode estimation approach of Livingstone and track multiple mode estimates at each time step instead of just the most likely mode estimate. This will enable CME to track complex behaviors of the system that evolve over time. Chapter 5 presents the method employed by CME to compile transitions.

This page intentionally left blank.

5   Compilation for Mode Estimation

5.1   *Motivation for Compilation*

Mission failures and the harsh environment of space are only two reasons that motivate the need for autonomy and mode estimation. Processing power, system memory and time are tight resources on-board a spacecraft. Additionally, the harsh environment of space requires a minimization of risk and error in software processes. These challenges require that a fault management system be able to determine system behavior in real time and minimize the footprint in the system memory. To address the minimization of risk, the results of the fault management engine must be made explicit to system engineers before operation of the system. A human modeler must be able to inspect the diagnoses of the engine and insure that it is correct with the system model. The engine developed in this research, Compiled Mode Estimation (CME) addresses these concerns. CME extends the concepts of GDE, Sherlock and Livingstone in order to improve mode estimation for spacecraft. CME gives the engineer the ability to inspect the diagnoses and the accuracy of the system model through the process of compilation. The compiled model enables CME to determine mode estimates in real time, in addition to requiring a smaller onboard memory footprint. Finally, CME can determine mode estimates more accurately than the Livingstone system by tracking multiple mode estimates over time.

Compiled Mode Estimation, uses a 'divide and conquer' approach, similar to GDE, with the key difference that the divide step is performed at compile time, rather than at run-time. Recall that GDE determines a diagnosis by dividing the diagnosis problem into sub-problems. The divide step involves identifying discrepancies between predicted observations and the actual observations, and then identifying the component modes involved in the prediction. The conquer step requires choosing other component modes to remove all discrepancies between predicted and actual observations. The compilation process performs the divide step of diagnosis by identifying all potential conflicts within the system model. This results in the compiled observation function, encoded as dissents, and the compiled transition function encoded as compiled transitions. The steps yet to be developed are to use the dissents and compiled transitions to obtain a diagnosis of the system, and develop the process to compile the transitions.

This chapter introduces the architecture and process of Compiled Mode Estimation through an example and details the compilation stage of this process. Chapter 4 presented the method for compiling component mode constraints. This chapter completes the development of compilation by presenting the method to compile transitions in Section 5.6.2. To better understand the utility of the compiled model, the architecture of CME is presented in Section 5.2 followed by an example in Section 5.5 that demonstrates the online determination of mode estimates. The algorithms and detail of CME are presented in Chapter 6, with the detailed implementation of these algorithms given in Chapter 7.

5.2 Architecture

The process of Compiled Mode Estimation (CME) generates diagnoses that are consistent with the observations collected and commands given up to

time 't+1' and the model. Compiled Mode Estimation, using the architecture shown in Figure 5-1, relies on inputs of a 'System Model', 'Observations' and 'Commands', and outputs a set of 'Current Mode Estimates', representing the diagnoses of the system. The 'System Model' adheres to the definition of Concurrent Constraint Automata (CCA), given in Section 3.2. The 'Observations' are defined as an assignment to each observation variable, $x_{io} \in \Pi_o$. The 'Commands' are defined similarly as an assignment to each command variable, $x_{ic} \in \Pi_c$. The output 'Current Mode Estimates' is the same as defined for the GDE, Sherlock and Livingstone diagnostic systems, where a mode estimate assigns to each component variable, $x_{im} \in \Pi_m$, a value from its domain, and these assignments resolve all conflicts. A mode estimate has an associated probability, which indicates the likelihood of the component mode assignments.
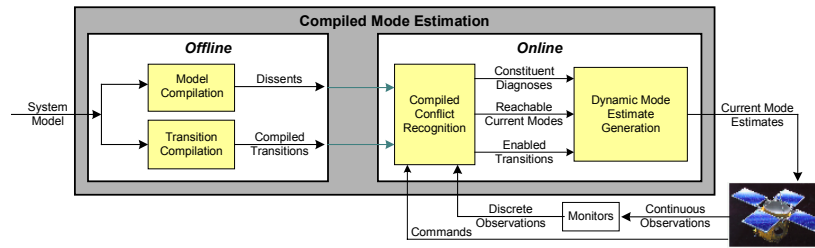
Figure 5-1 - Compiled Mode Estimation Architecture

The inputs and outputs of CME are defined formally as follows.

System Model $\equiv$ *Concurrent Constraint Automata*

Observations $\equiv \left\{ \left(x_{1o} = v_{1l_1} \right), \dots, \left(x_{no} = v_{nl_n} \right) \right\} \forall x_{io} \in \Pi_O$

Commands $\equiv \left\{ \left(x_{1c} = v_{1l_1} \right), \dots, \left(x_{nc} = v_{nl_n} \right) \right\} \forall x_{ic} \in \Pi_C$

Current Mode Estimates $\equiv \left\{ \left\langle S_i^{(t)}, P\left(S_i^{(t)} \right) \right\rangle \right\}$ where $S_i^{(t)} \equiv \left\{ \left(x_{1m} = v_{1l_1} \right), \dots, \left(x_{nm} = v_{nl_n} \right) \right\} \forall x_{im} \in \Pi_m$

$S_i^{(t)}$ satisfies all conflicts at time t, and $P\left(S_i^{(t)} \right)$ is the probability of $S_i^{(t)}$

Compiled Mode Estimation is divided into two processes. In the "offline" stage the system model is compiled into 'Dissents' and 'Compiled

Transitions'. This maps the 'System Model', expressed as a CCA, to a compiled concurrent automata (CMPCA), expressed using the 'Dissents' and 'Compiled Transitions'. In the 'online' stage, CME uses the CMPCA, the 'Observations' and 'Commands' over the time period '*t*' to '*t+1*' to generate the 'Current Mode Estimates' of the system.

5.3   Dissents

Recall that dissents are a compiled form of the observation function of Hidden Markov Models, and represent the component mode constraints of a CCA.   A dissent maps observations to a set of component mode assignments that are infeasible, a conflict.   As an example, consider the dissent below from the NEAR Power Storage system.

$$[(\text{bus voltage} = \text{nominal})] \Rightarrow \neg[(\text{switch} = \text{charger-1}) \wedge (\text{charger-1} = \text{off})]$$

Equation 5-1 – Example Dissent

This dissent expresses the observation '*bus voltage = nominal*' and the link between the infeasible component modes *switch = charger-1* and *charger-1 = off*. This inconsistency of component mode assignments arises because if the incoming bus voltage is nominal, then the charger must be either trickle charging or giving a full charge to the battery, otherwise, the switch cannot be at that charger position. It follows then that the *switch* is either at *charger-2* or broken in some manner, or that the *charger-1* is in trickle or full-on charge mode.

Dissents encode the relationship of observations and component mode assignments through the logical implication connective. The process of generating dissents using the enumeration algorithm is described in Section 2.4. The characteristics to note here are that the dissents are comprised of information known in the antecedent (observations) and information that is inconsistent, or that cause a conflict, in the consequent (component modes).

This is exploited in the online phase of CME to simplify the diagnosis process, which is demonstrated in Section 5.5.
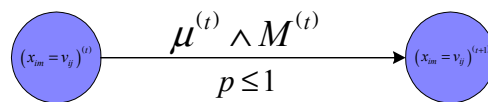
5.4 Compiled Transitions

Compiled transitions encode the transition function of a Hidden Markov Model and represent the component mode transitions of a CCA. They are compiled as specified by the transition compilation process described in Section 5.6.2. A transition function specifies reachable component modes from a previous state, and the compiled transitions encode the transition function using only the component mode variables, x_{im}, and the control variables, x_{ic}. Intermediate variables, are not included in a compiled transition. Take as an example a compiled transition from the NEAR Power Storage System of Chapter 1.

$$[(\text{battery} = \text{Full}) \wedge (\text{charger-1} = \text{Trickle})] \Rightarrow (\text{battery} = \text{Charging})$$

Equation 5-2 – Example Transition

Looking at Equation 5-2, recall that the *charger-1* mode was not an input to the *battery*, but the '*charger-current*' was an input. Since the '*charger-current*' is not a direct observable it is compiled away using the transition compilation process. The result of this compilation is to replace the '*charger-current*' with the mode of *charger-1* that would entail the same assignment, in this case replacing '*charger-current = trickle*' with '*charger-1 = trickle*'.

In general, a compiled transition is represented as:

$$\left(x_{im} = v_{ij}\right)^{(t)} \quad \xrightarrow{\begin{array}{c} \mu^{(t)} \wedge M^{(t)} \\ p \leq 1 \end{array}} \quad \left(x_{im} = v_{ij}\right)^{(t+1)}$$

Figure 5-2 – General Component, Compiled Transition

In this generalized transition, note that the source and targets are assignments to a single mode variable. The same variable is used in both assignments, but the value contained in the assignments may or may not be the same, allowing for idle transitions. In order for a transition to be taken, its source mode must be in the previous mode estimate, and its guard must be satisfied, meaning that the assignments in the guard must be true. The '$\mu^{(t)}$' represents the commands, and '$M^{(t)}$' represents the component mode assignments at time 't'. This allows for transitions of components to be conditioned on other components in the system. Finally, each transition has an associated probability, capturing the probabilistic behavior of actual components.

Notice that the compiled transitions are also expressed with assignments that are known at the time of execution as opposed to assignments that have to be deduced, as was the case in Livingstone. These include the *previous* commands, and the *previous* component modes at time 't'. This fact is exploited in the online mode estimation algorithms demonstrated in Section 5.5. The formal definition of a compiled transition is.

$$transition \rightarrow antecedent^{(t)}\ implies\ consequent^{(t+1)}$$

$$antecedent^{(t)} \rightarrow assignment_{im} \wedge guard$$

$$consequent^{(t+1)} \rightarrow assignment_{im}$$

$$guard \rightarrow \text{TRUE} \mid commands \wedge modes$$

$$commands \rightarrow assignment_{1c} \wedge ... \wedge assignment_{nc}$$

$$modes \rightarrow assignment_{1m} \wedge ... \wedge assignment_{jm},\ \text{where}\ j \neq i$$

$$assignment_{im} \rightarrow \left(x_{im} = v_{ij} \right)$$

$$assignment_{ic} \rightarrow \left(x_{ic} = v_{ij} \right)$$

Figure 5-3 - Definition of a Compiled Transition

Achieving Real-time Mode Estimation through Offline Compilation   117

This definition breaks the compiled transition into three distinct pieces, the source component mode assignment, the guard or transition constraint, and the destination component mode assignment. The source and destination component mode assignments are restricted to the same component variable, x_{im}. The guard is made up of any combination of command and component mode variables. The only exception is that the '*modes*' cannot contain the component mode variable x_{im} that is in the source and destination.

5.5   Online Mode Estimation at a Glance

The mapping of the compiled model to the current mode estimates is demonstrated using the NEAR Power Storage system described in Chapter 1. Considering the observations '*bus-voltage = nominal*', '*battery-voltage = nominal*', '*battery-temperature = nominal*' and the initial mode estimate '*switch = charger-1*', '*charger-1 = full-on*', '*charger-2 = off*', and '*battery = charging*', the following is a subset of the dissents and transitions for the NEAR Power Storage System. The full set of dissents and transitions are given in Appendix B, and the full example for this set of observations and initial state is shown in Appendix C.

```
1.  [ ]  ⇒ ¬[ SWITCH = STUCK-CHARGER-2 ∧ CHARGER-1 = FULL-ON
    ]
2.  [ ]  ⇒ ¬[ SWITCH = STUCK-CHARGER-1 ∧ CHARGER-2 = FULL-
    ON]
3.  [ ]  ⇒ ¬[ SWITCH = STUCK-CHARGER-1 ∧ CHARGER-2 =
    TRICKLE]
4.  [ BATTERY-TEMPERATURE = NOMINAL ] ⇒ ¬[ BATTERY = FULL ]
5.  [ BATTERY-VOLTAGE=NOMINAL ] ⇒ ¬[ BATTERY = DISCHARGING
    ]
6.  [ BUS-VOLTAGE = NOMINAL ] ⇒ ¬[ SWITCH = CHARGER-1 ∧
    CHARGER-1 = OFF ]
7.  [ BUS-VOLTAGE = NOMINAL ] ⇒ ¬[ SWITCH = CHARGER-2 ∧
    CHARGER-2 = TRICKLE ]
```

...

Switch

```
FROM CHARGER-1        GUARD (NOT (CHARGER-1.MODE = BROKEN)) TO
CHARGER-1
      p = 0.9899)
FROM CHARGER-1        GUARD NIL              TO STUCK-CHARGER-1

      p = 0.01)
```
...

Charger-1
```
FROM FULL-ON   GUARD (NOT (BATTERY-1.BATT-TEMP = HIGH))     TO
FULL-ON
      p = 0.89)
FROM FULL-ON   GUARD NIL            TO OFF
      p = 0.1)
```
...

Charger-2
```
FROM OFF       GUARD NIL            TO OFF
      p = 0.1)
FROM OFF              GUARD NIL              TO BROKEN
      p = 0.01)
```
...

Battery
```
FROM CHARGING  GUARD (CHARGER-1.MODE = FULL-ON)            TO
FULL
      p = 0.99)
FROM CHARGING  GUARD NIL            TO DEAD
      p = 0.001)
```
...

**Figure 5-4 - Dissents and Compiled Transitions for NEAR Power Storage
Example**

The transitions above specify the source component mode assignment after
"FROM", and the target after "TO". The constraints on the transition are
specified after the keyword "GUARD", where 'NIL' represents an empty
constraint.

Using the observation, initial mode estimate and control action values, a
subset of the dissents and compiled transitions are used to determine the
current mode estimates. This first step is performed by the Compiled
Conflict Recognition, which determines the dissents and compiled
transitions that pertain to the current observations, control actions and

previous mode estimates. These are mapped to a set of 'Constituent Diagnoses', 'Reachable Component Modes' and 'Enabled Transitions'. From the example dissents shown in Figure 5-4, a subset of the 'Constituent Diagnoses' is:

```
1.  [ SWITCH=CHARGER-1 ∨ SWITCH=CHARGER-2 ∨ CHARGER-1=TRICKLE
       ∨  CHARGER-1=OFF   ∨   SWITCH=STUCK-CHARGER-1   ∨   CHARGER-
       1=BROKEN ∨ SWITCH=UNKNOWN ∨ CHARGER-1=UNKNOWN ]
2.  [ SWITCH=CHARGER-1 ∨ SWITCH=CHARGER-2 ∨ CHARGER-2=TRICKLE
       ∨  CHARGER-2=OFF   ∨   SWITCH=STUCK-CHARGER-2   ∨   CHARGER-
       2=BROKEN ∨ SWITCH=UNKNOWN ∨ CHARGER-2=UNKNOWN ]
4.  [ BATTERY = CHARGING ∨ BATTERY = DISCHARGING ∨ BATTERY =
       DEAD ∨ BATTERY=UNKNOWN ]
7.  [ SWITCH=CHARGER-1 ∨ CHARGER-2=FULL-ON ∨ CHARGER-2=OFF ∨
       SWITCH=STUCK-CHARGER-1 ∨ SWITCH=STUCK-CHARGER-2 ∨ CHARGER-
       2=BROKEN ∨ SWITCH=UNKNOWN ∨ CHARGER-2=UNKNOWN ]
```

The set of constituent diagnoses represent the feasible space of mode assignments that can be chosen to satisfy each conflict. Each component mode assignment is referred to as a constituent diagnosis of the conflict, and the set is referred to as the constituent diagnoses of the conflict. By choosing component mode assignments mentioned in these constituent diagnoses, the mode assignments then resolve the conflicts. A full diagnosis resolves a conflict if it contains at least one of the constituent diagnoses of the conflict.

The compiled transitions further reduce the space of feasible component mode assignments by determining the set of '*reachable component mode assignments*'. The reachable component mode assignments represent those component modes that are the target modes of transitions from the previous mode assignments at time 't'. Recall the introductory example where the initial mode estimate and the transitions determined the possible mode assignments for each component. The '*reachable component modes*'

represents this same set of component mode assignments. For this example, the set of reachable component modes is shown in Figure 5-5.
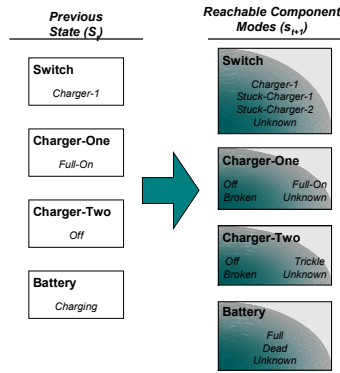

Figure 5-5 - The Set of Reachable Component Modes

Not noted on this figure are the probabilities associated with each component mode. These are shown in Appendix C with the full example. This set of component modes further reduces the space by eliminating the '*battery = discharging*' mode and the '*switch = charger-2*' mode. This set of component mode assignments is determined by Compiled Conflict Recognition by first determining the set of 'Enabled Transitions' and then using the target modes of these enabled transitions to create the list of 'Reachable Component Modes'.

Having mapped the dissents and transitions to the 'Constituent Diagnoses', 'Reachable Component Modes' and the 'Enabled Transitions', these are used in a modified version of conflict-directed A* search [*Williams, 2002*] to determine mode estimates. This process is similar to Livingstone's process of generating kernel diagnoses. The difference is that the CME process tracks an approximate belief state while Livingstone tracks the most likely trajectories. The conflict-directed A* search is guided by the constituent diagnoses to determine the minimal set of component mode

assignments, with the added constraint of generating the most likely mode estimate using the transition probabilities.

This is demonstrated using the example constituent diagnoses given above and the space of reachable component modes shown in Figure 5-5. The full tree associated with this example is detailed in Appendix C. From the first set of constituent diagnoses above, the search tree expands to:
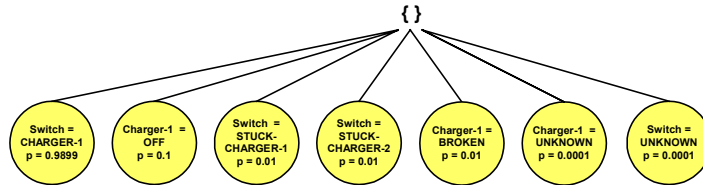
Figure 5-6 - Expansion of First Set of Constituent Diagnoses

The Dynamic Mode Estimate Generation procedure then chooses the most likely node from this search tree. From the above search tree, the proper assignment to choose is '*switch = charger-1*' with a likelihood of 0.9899. The next step of the algorithm is to determine which constituent diagnoses this assignment satisfies, and choose a constituent diagnosis to expand from this node. For instance, the component mode assignment *switch = charger-1*, satisfies constituent diagnoses 2 and 7 listed above, as well as 1. The following figure shows the subsequent expansion of the next constituent diagnosis from the node '*switch = charger-1*'.
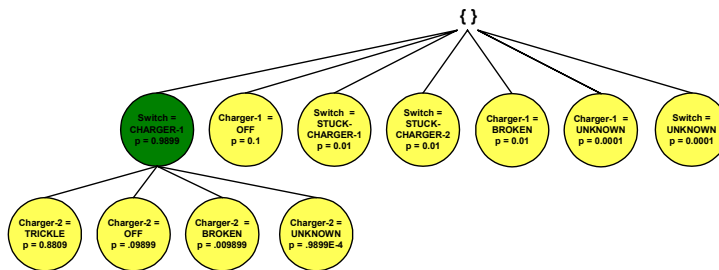
Figure 5-7 - Expansion of the Next Set of Constituent Diagnoses

Dynamic Mode Estimate Generation would again choose the most likely node and expand another constituent diagnosis. This process of expansion and choosing likely nodes describes an conflict-directed A* search that is modified to use constituent diagnoses in the expansion phase. A similar search is used in Livingstone to generate the most likely mode estimates. For this example, the resultant mode estimate, as shown in Appendix C, is:

(*switch = charger-1*), (*charger-1 = full-on*), (*charger-2 = off*), (*battery = charging*)

with a probability of $p = .04396$.

The Dynamic Mode Estimate Generation process does not require a satisfiability test since the set of conflicts is complete and the transitions are compiled. It is enough to use these constituent diagnoses to reconstruct the full diagnosis of the system. Additionally, Dynamic Mode Estimate Generation tracks multiple mode estimates at each time step. This is an improvement upon the Livingstone system that tracked a single mode estimate at each time step.

This example grounds the mapping of the compiled model as dissents and compiled transitions, to *constituent diagnoses*, *reachable component modes*, and *enabled transitions*. These outputs of the Compiled Conflict Recognition are then used in the Dynamic Mode Estimate Generation algorithm to produce the current mode estimates. The benefits of the enabled transitions are in the Dynamic Mode Estimate Generation algorithm. They were not needed here since the example assumed a single previous mode estimate.

5.6   Compilation

The number of trajectories that can be tracked by CCA mode estimation is limited by the significant cost of determining the satisfiability of transition

constraints and determining the consistency with the observations. Compiled Mode Estimation increases the number of trajectories tracked by removing the need for online satisfiability completely. The Mini-ME engine developed the process of compiling modes to dissents, hence eliminating the need for full satisfiability to test consistency with the observations. The remaining step is to develop an algorithm to compile the transition guard constraints of the component modes, hence eliminating the need for full satisfiability to determine if transition guards are entailed.

Recall that the Mini-ME engine, by compiling the component mode constraints, mapped the observation variables to a set of conflicts, encoded as dissents. These dissents represent the observation function of a Hidden Markov Model (HMM). The transition function of a Hidden Markov Model is encoded using the compiled transitions. These two elements are essential in Compiled Mode Estimation since they enable the use of standard belief update equations to determine mode estimates. Full model compilation is then broken up into two steps, depicted below:
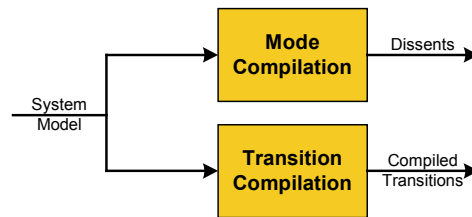
Figure 5-8 - Steps of Model Compilation

This section develops the theory and algorithm for transition compilation. First, the definition of the resultant compiled automata, Compiled Concurrent Automata (CMPCA), is given in section 5.6.1. Section 5.6.2 develops the compilation of transition constraints. The section concludes with an example demonstrating transition compilation.

5.6.1 Compiled Concurrent Automata

A Compiled Concurrent Automata (CMPCA) describes an automaton compiled from a CCA. The CCA is a compact encoding of a Hidden Markov Model, so the CMPCA is a compact encoding of the compiled observation and transition functions of an HMM. A CMPCA is encoded using the system variables, partitioned into observation, control and component mode variables. A CMPCA is built up from the dissents and compiled transitions.

A CMPCA is the tuple $\left\langle \Pi, D, T_{Ci}, P_{T_{Ci}}, P_{\theta_i} \right\rangle$:

- Π is a set of system variables where each $x \in \Pi$ ranges over a finite domain $D(x)$. Π is partitioned into sets of *mode variables*, Π_m, *observable variables*, Π_o, and *control variables*, Π_c.
  - Mode variables, Π_m, represent the different modes of components in the system. The set $\Pi_m = \cup \{\Pi_m | \ i=1..n\}$
  - Observable variables capture the values of the spacecraft sensors. The set $\Pi_o = \cup \{\Pi_o | \ i=1..n\}$
  - Control variables provide the means to assert actions on the system. The set $\Pi_c = \cup \{\Pi_c | \ i=1..n\}$

- Dissents map observations to infeasible component mode assignments. This is the set D, where elements of D are of the form $\left(x_{1o}=v_{1l_1}\right) \wedge \left(x_{2o}=v_{2l_3}\right) \cdots \left(x_{po}=v_{pl_p}\right) \Rightarrow$
$$\neg\left[\left(x_{1m}=v_{1l_1}\right)\wedge\left(x_{2m}=v_{2l_4}\right)\cdots\wedge\left(x_{qm}=v_{ql_q}\right)\right] \text{ where } p \leq n_O \text{ and } q \leq n_M$$

- $T_i : D(\Pi_m) \times \mathbb{C}(\Pi_i) \rightarrow D(\Pi_m)$ associates with each component mode a set of compiled transitions $T_i\left(x_{im}=v_{ij}\right)$. Each compiled transition function specifies an assignment, $x_{im} = v_{ij}$ in the next time step, '$t + 1$', given partial assignments to the variables in Π at time 't'. The constraints $\mathbb{C}$ are defined using the set of variables Π_i, where $\mathbb{C} \equiv \mu^{(t)} \wedge M^{(t)}$

- $P_{T_i} : T_i\left(x_{im}=v_{ij}\right) \rightarrow \Re[0,1]$ represents the probability associated with each transition $T_i^k\left(x_{im}=v_{ij}\right)$ for each mode variable in the system.

- $P_{\theta_i} : D(\Pi_m) \rightarrow \Re[0,1]$ denotes the probability that $x_{im} = v_{ij}$ is the initial mode.

Equation 5-3 - Definition of a Compiled Concurrent Automata

The definition given here for the compiled concurrent automata follows from the definitions of the constraint automata and concurrent constraint

automata. This definition captures the behaviors of the original model that are encoded in the dissents and compiled transitions. The definition maintains the probabilities on the compiled transitions, and the probability on initial modes. Each of these elements are used in the Compiled Mode Estimation algorithm developed in Chapter 4.
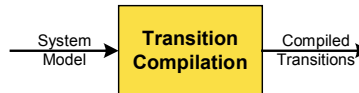
5.6.2 Transition Compilation

The final piece to enable Compiled Mode Estimation is the compilation of the transitions between component modes. Compiling transitions requires removing the need for full satisfiability of transition constraints at the time of execution. By removing this exponential computation, Compiled Mode Estimation is capable of increasing performance significantly. Mode compilation has removed the need for satisfiability with respect to the mode constraints in the system model. To complete the removal of satisfiability in determining mode estimates, the transitions must be compiled.

Transition compilation is developed by first discussing the inputs and outputs of transition compilation, followed by the development of the theory and resulting algorithm.

### 5.6.2.1 Inputs and Outputs

The compilation of transitions maps the system model to a set of compiled transitions. The figure below depicts this:

Figure 5-9 - Inputs and Outputs of Transition Compilation

The system model taken as input to the transition compilation algorithm is defined as a CCA. In particular the transition guards in the CCA are expressed over the control, component mode and intermediate variables. In order to remove the need for a satisfiability engine, the guard is replaced with an equivalent guard that contains only control and component mode variables, but no intermediate variables. Transition compilation removes these from the transition guards.

The compiled transitions are expressed similar to un-compiled transitions, with a source and a target component mode assignment, and a guard. The label is expressed using only the control variables and the component mode variables. The compiled transition is represented graphically in Figure 5-10.

$$\left(x_{im} = v_{ij}\right)_t \xrightarrow{\begin{array}{c} \mu^{(t)} \wedge M^{(t)} \\ p \leq 1 \end{array}} \left(x_{im} = v_{in}\right)_{t+1}$$
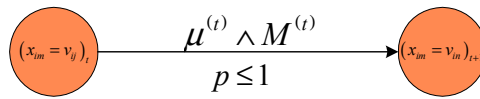
Figure 5-10 - Depiction of a Compiled Transition

Note that the probability is carried over from the original un-compiled transition in the system model.

5.6.2.2   Transition Compilation Algorithm

Generating compiled transitions requires maintaining equivalence with the original system model transitions and associated guards. The compiled transition guard must convey the same constraints as the original transition guard. To compile a transition for a particular source component mode assignment, the algorithm determines all combinations of control and component mode assignments that entail the original guard:

$$\left(x_{im} = v_{ij}\right) \wedge \Phi \models \left(cg \Rightarrow g\right)$$

Equation 5-4 - Entailment Question for Transition Compilation

where *cg* represents the compiled guard and *g* represents the original transition guard. This logical statement is equivalent to:

$$\left(x_{im} = v_{ij} \right) \wedge \Phi \wedge cl \wedge \neg l \text{ are inconsistent}$$

This requires the transition compilation to individually compile the transitions for each component mode assignment in the system model. The transition compilation algorithm must search for combinations of component mode assignments involving only the control and component mode assignments, and the negation of the assignments in the original label. The set of possible component mode variables to search over is decreased by one due to the source component mode assignment.

Transition compilation solves a similar constraint satisfaction problem as mode compilation. Combinations of control and component mode assignments are generated and tested for inconsistency with the system model. This is framed as an OPSAT problem so as to generate the minimal set of compiled guards for the transitions. Transition compilation instantiates an OPSAT problem for each component mode assignment and its associated transition guard from the original system model. The set '*x*' of the system variables are all variables within the system model, except the source component mode assignment. The source component mode assignment is added in as a constraint to the set of constraints G_x. This ensures that the source component mode assignment appears in the compiled result. Additionally, the transition guard, *g*, is negated and added as a constraint in the set of system model constraints, G_x of the OPSAT instantiation. The set of variables, '*y*', to be optimized are set to be the control and component mode variables in the system. Finally, the optimization function is given as the length of the candidates generated so that a candidate with fewer assignments has a better cost. Transition compilation generates the minimal set of compiled guards by performing a

subsumption check on a candidate with the current list of compiled guards. Transition compilation as an OPSAT problem is stated as follows:

$$OPSAT(s) \equiv \langle \bar{y}, f, CSP \rangle$$

$$CSP(s) \equiv \langle \bar{x}, D_x, G_x \rangle$$

where

$\bar{x} \equiv$ all variables in the system model, except the source x_{im} of the transition

$D_x \equiv$ the domains of the vector of variables, $\bar{x}$

$G_x \equiv$ the mode constraints to be unsatisfied, including $\neg l \land x_{im} = v_{ij}$

$\bar{y} \equiv$ the control variables, μ, and component mode variables, $x_{jm} \neq x_{im}$

$f \equiv$ minimize the length of assignments in a conflict

$OPSAT(s) \rightarrow$ an assignment to each variable $\vec{x}$

Figure 5-11 - Transition Compilation as OPSAT

Transition compilation is framed as an OPSAT problem that uses the unsatisfiability engine to determine inconsistency. Upon adding to the constraints, G_x, the negation of the original transition label and the source component mode assignment, the compiled result will contain these elements, along with the compiled label. The compiled result is given as:

$$\left(x_{im} = v_{ij} \right) \land \neg g \land cg$$

Transition compilation extracts the labels, cg, from the compiled result and returns the compiled transition including the original source and target component mode assignments, as well as the transition probability. The resultant transition compilation algorithm is given below:

Transition-Compilation(*Model-CCA*)

1   **create** a list $\mathbf{T}_c$ to hold the compiled transitions
2   **for each** $x_{im} = v_{ij}$ in *Model-CCA*
3   **for each** $\mathbf{T}_i^k (x_{im} = v_{ij} \rightarrow x_{im} = v_{in}) \in \mathbf{T}_i$
4   **extract** guard g, probability p, and target $x_{im} = v_{in}$ from $\mathbf{T}_i^k$
5   **add** $x_{im} = v_{ij}$ and $\neg g$ to constraints C_M of *Model-CCA*
6   **create** a queue, *Nodes*, that maintains the candidates of the search tree
7   **while** *Nodes* is *not empty*
8   *best-node* = extract shortest from *Nodes*
9   **if** *best-node* is not subsumed by *cl*, **then**
10   **if** **unsat**(*best-node*, C_M), **then** add *best-node* to *cl*
11   **otherwise**, **extend** *best-node* as follows:
12   **for** an $x_i = x_{ic}, x_{im}$ in *Model-CCA* unassigned in *best-node*

```
13              for each $v_{ij} \in \mathbf{D}(x_i)$
14                  new-node = best-node $\cup$ $x_i = v_{ij}$
15                      insert new-node in Nodes by length
16                  end for
17              end if
18          end if
19      end while
20      remove the constraint $\neg g$ from $C_M$
21      extract cg from the compiled result
22      create compiled transition $\mathbf{T}_{ci}$ using $x_{im} = v_{ij}$, $x_{im} = v_{in}$, cg, and p
23      $\mathbf{T}_c = \mathbf{T}_c \cup \mathbf{T}_{ci}$
24      return $\mathbf{T}_c$
```

The transition compilation algorithm described above iterates over the different source component mode assignments in the system model, performing several operations. First, the algorithm extracts the label, probability and target mode assignment of a particular transition. Then the source mode assignment and the negation of the label, *l* are added to the system constraints, C_M. The next phase is the "generate-and-test-loop" that determines the compiled label, *cl*. The algorithm creates a queue of candidates and extracts the shortest candidate from the queue. This candidate is first tested for subsumption with the existing compiled label. If the candidate is not subsumed, then the candidate is tested for inconsistency using the constraints, C_M. If the candidate is inconsistent it is added to the compiled label. If it is not, then the candidate is extended by expanding the tree using an unassigned control or component mode variable. The expansion is restricted to not include the mode variable in the source of the transition. Once the expansion occurs, the newly generated nodes are added to the queue in order of length. The generation of candidates terminates only when the entire search tree has been explored. Branches of the tree are pruned at the time of subsumption to increase efficiency.

Once the compiled label has been generated, the compiled transition is reconstructed using the source and target mode assignment, extracting the compiled label from the compiled result, and associating the original

transition probability to this compiled transition. The algorithm exits once all component mode assignments in the source of a transition have been used.

5.6.3 Transition Compilation Example

This section details an example to demonstrate the steps of the transition compilation algorithm. Consider the NEAR Power storage system of Chapter 1. This example focuses on the interaction of the battery and a charger in the system to compile the transitions of the battery. Figure 5-12 depicts the interactions between the *charger* and the *battery*.
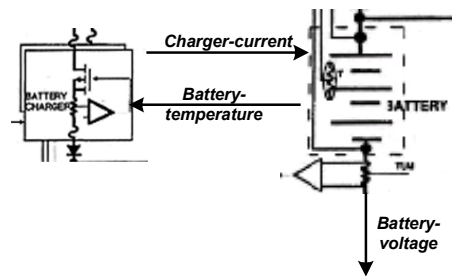
Figure 5-12 - Diagram of the Charger and Battery of NEAR

The *battery* and the *charger* communicate using the dependent variable '*charger-current*'. The battery uses this output in the transitions between component modes. For instance, the transition between the modes '*charging*' and '*full*' is determined when the '*charger-current = nominal*'. This value indicates that the charger has increased the current coming to the *battery*. However, in order for the '*charger-current*' to be '*nominal*', the *charger* can only be in the '*full-on*' mode.

The process of transition compilation determines the variable values that entail the same information as the '*charger-current*'. For the battery, the following list of transitions must be compiled. There is no other variable information associated with these transitions other than the '*charger-current*'.

1. source mode: (*battery = full*) destination mode: (*battery = charging*)

2. source mode: (*battery = full*) destination mode: (*battery = discharging*)

3. source mode: (*battery = charging*) destination mode: (*battery = full*)

4. source mode: (*battery = charging*) destination mode: (*battery = discharging*)

5. source mode: (*battery = discharging*) destination mode: (*battery = charging*)

Associated transition labels:

1. *l*(*full → charging*) = {*charger-current = nominal* }
2. *l*(*full → discharging*) = { *charger-current = zero* }
3. *l*(*charging → full*) = { *charger-current = trickle* }
4. *l*(*charging → discharging*) = { *charger-current = zero* }
5. *l*(*discharging → charging*) = { *charger-current = nominal* }

The remaining transitions of the battery all have an empty label since they are fault transitions. The full constraint automaton associated with the battery is given in Appendix A.

The transition compilation algorithm first identifies one of the battery modes. Assume that the algorithm chooses the component mode *battery = full* and compiles its transitions. The algorithm extracts the label, negates it and adds it to the constraints first. Assuming the algorithm is compiling the first transition, the associated label is *charger-current = nominal*. The

algorithm is capable of searching over control variables and component modes. In this example there are no control variables, and the available component modes are *switch.mode*, and *charger-1.mode*.

Focusing on the component mode *charger-1.mode*, the transition compilation algorithm would try different modes of this component to determine inconsistency. The possible component mode assignments are { *full-on*, *trickle*, *off*, *broken* }. In testing the first assignment *charger-1 = full-on* and the model constraints for inconsistency, the algorithm determines that this combination is inconsistent. The component mode *charger-1 = full-on* is then determined to be part of the compiled label, and added to *cl*. The algorithm proceeds to test the different modes of components, now using the *charger-1 = trickle* component mode. By testing this component mode, the algorithm predicts that *charger-current = trickle* for this component mode. However, this value is consistent with the model constraints and the negated label, so the component mode is not part of the compiled label. The transition compilation algorithm continues to try different values of the *charger-1.mode* and the *switch.mode*. However, only the component mode *charger-1 = full-on* is one that is inconsistent with the system model constraints. The algorithm would not test any superset of this component mode as it is not allowed by subsumption.

The remaining transitions of the *battery* are compiled in a similar manner. The resulting compiled transitions are then:

1. *battery = full → battery = charging l* : *charger-1 = full-on* p = 0.95
2. *battery = full → battery = discharging l* : *charger-1 = off* p = 0.04
3. *battery = charging → battery = full l* : *charger-1 = trickle* p = 0.95
4. *battery =charging → battery =discharing l* : *charger-1= off* p = 0.04
5. *battery=discharing → battery=trickle l* : *charger-1=full-on* p = 0.99

This example completes the development of model compilation. The process of model compilation has built upon the conflict-based algorithms of GDE, Sherlock, Livingstone and Mini-ME. Compiled Mode Estimation extends Livingstone by tracking multiple trajectories of mode estimates. It is enabled by the results of the compilation algorithms given in this chapter and Chapter 2. The algorithms of Compiled Mode Estimation are described in Chapter 4 and detailed in Chapter 5.

This page intentionally left blank.

6   Online Mode Estimation

6.1   *Architecture*

This chapter develops the second portion of the CME architecture, the process of determining online mode estimates of the spacecraft system. In the architecture shown in Figure 5-1, the dissents and compiled transitions are taken as an input to the online phase and, together with the observations and commands, are used to determine a set of current mode estimates that are consistent with these inputs. The mode estimate is determined by using the conflicts in the dissents to identify infeasible sets of component mode assignments. The compiled transitions are used to encode probabilities of component mode assignments, enabling diagnostic discrimination based on likelihood. Online-ME then tracks an approximated belief state over time by determining the most likely transitions from mode estimates in the previous belief state to mode estimates in the current belief state. Additionally, the current mode estimates must resolve all conflicts associated with the current observations.

To perform the process of mode estimation, the 'online' portion of CME is divided into two steps, shown in Figure 6-1. The first step, Compiled Conflict Recognition, determines the dissents and transitions that relate to the current observations and commands. The next step is to generate mode

estimates using the reachable component modes determined from the compiled transitions, and the conflicts transformed into constituent diagnoses. The Dynamic Mode Estimate Generation process uses the transformed conflicts to guide the choice of component mode assignments, using a modified conflict directed A* search.
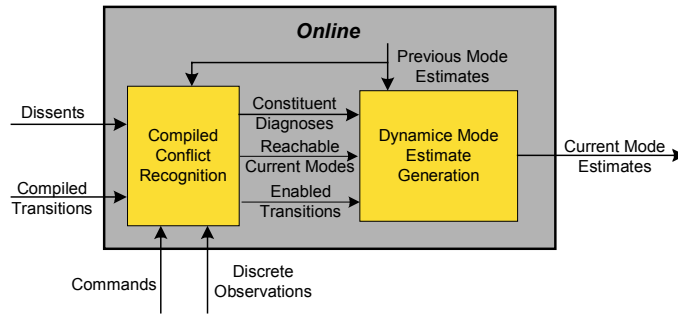
Figure 6-1 - Inputs/Outputs of Online Mode Estimation

The following section describes more formally the inputs and outputs of the online compiled mode estimation system, focusing on the 'Constituent Diagnoses', 'Reachable Current Modes', 'Enabled Transitions' and the 'Previous Mode Estimates'. Sections 6.3 and 6.4 discuss the 'Compiled Conflict Recognition' and the 'Dynamic Mode Estimate Generation' algorithms, respectively.

6.2   Inputs / Outputs

This section defines the inputs and outputs of the Online Mode Estimation process. All inputs to Online Mode Estimation have been defined earlier. The definition of the compiled model has been given previously in Section 5.6.1. This section then focuses on the definitions for the 'Constituent Diagnoses', the 'Reachable Component Modes' and the 'Enabled Transitions'.

Building on the example in the section 5.5, the definitions of the internal inputs and outputs are:

- Constituent Diagnoses $(cd) \equiv \{(x_{1m}=v_{1l_1}),...,(x_{pm}=v_{pl_p})\}$ where $x_{im} \in \Pi_m$ and $p \le n$, where n is the number of components in the system. The assignment $x_{im} = v_{ij}$ is a constituent diagnosis that resolves the conflict used to determine the constituent diagnoses of cd.

- Reachable Component Modes $(m^{(t+1)}) \equiv \{\langle(x_{1m}=v_{1l_1}),p_{1l_1}\rangle, \langle(x_{1m}=v_{1l_2}),p_{1l_2}\rangle,...,$

$$\langle(x_{nm}=v_{nl_1}),p_{nl_1}\rangle, \langle(x_{nm}=v_{nl_n}),p_{nl_n}\rangle\} \; \forall x_{im} \in \Pi_m$$

  For each assignment, $x_{im}=v_{ij}$, there is an associated probability, determined by the transition function, T_i^k. A variable, x_{im}, can have more than one assignment possible in the current time, $t+1$ as well.

- Enabled Transitions $(T_{EN}) \equiv \{T_i^k \mid T_i^k \in T_i(x_{im}=v_{ij}),$ and the guard of T_i^k is satisfied by a mode estimate at time 't'.} The set, T_{EN}, is the union of all enabled transitions for all component variables $x_{im} \in \Pi_m$.

Figure 6-2 - Input/Output Definitions for Online Compiled Mode Estimation

The constituent diagnoses, as described here, are a disjunction of component mode assignments, represented as a set. By choosing an assignment in the constituent diagnoses of a conflict, the conflict is then satisfied. The set of reachable component modes is a set of pairs consisting of a component mode variable assignment, and an associated probability. This probability is derived from the transition, T_i^k, that mentions the assignment, $x_{im} = v_{ij}$, as a target. The list of reachable component mode assignments is generated using the 'enabled transitions'. These 'enabled transitions' are the set of transitions whose source is in the previous mode estimates, and the guard is satisfied by the set of commands and previous mode estimates.

The final internal element of the Online Mode Estimation process that has not been described is the set of previous mode estimates. A mode estimate is defined as a pair $\langle S_i^{(t)}, P(S_i^{(t)}) \rangle$, where $S_i^{(t)}$ denotes a state of the system, and $P(S_i^{(t)})$ denotes the probability of that state. The set of these mode estimates is defined as a belief state, $B^{(t)}$. The belief state must be computed

at each time step to track the trajectories of the system. Recall the trellis diagram of Figure 2-2, that denoted sets of states at each time step, '*t*'. To calculate mode estimates, Compiled Mode Estimation in effect creates a moving window over the trellis diagram. This belief state stored at each time step is represented by the set of 'previous mode estimates' denoted on the architecture in Figure 6-1.

Mentioned previously, this mode estimation engine is an improvement on the Livingstone engine and its assumption of a single previous mode estimate. The Compiled Mode Estimation engine tracks a set of mode estimates at each time step to improve accuracy and hold to the theory of belief state update developed in Section 3.2 for Hidden Markov Models.

6.3   Compiled Conflict Recognition

This section describes the algorithm that maps the compiled model in the form of dissents and compiled transitions to a set of constituent diagnoses, a set of enabled transitions and a set of reachable component modes. Figure 6-3 denotes the architecture designed to map the compiled model to the desired outputs.
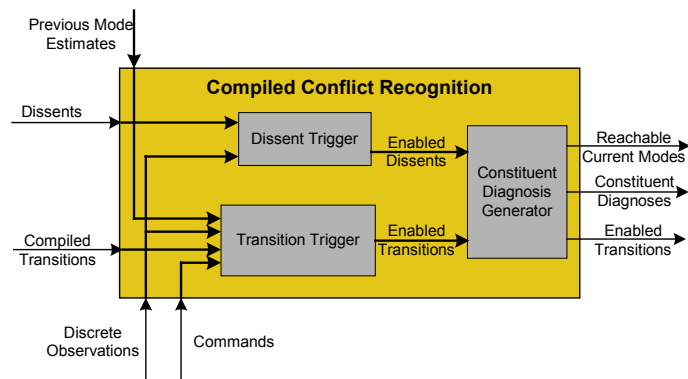
Figure 6-3 - Processes within the Compiled Conflict Recognition

Achieving Real-time Mode Estimation through Offline Compilation   139

The role of the Dissent Trigger is to trigger the appropriate dissents from the full list of dissents using the observations. Recall the form of a dissent, defined in Section 5.3. The examples show that the antecedent of the implication, the observation information, is all that is necessary to determine if a particular dissent needs to be enabled. For example, to determine if the dissent below is enabled, the observation '*bus-voltage = nominal*' must occur, then the dissent is triggered and added to the list of enabled dissents, D_{EN}.

```
[ BUS-VOLTAGE=NOMINAL ] ⇒ ¬[ SWITCH=CHARGER-2 ∧ CHARGER-2
=TRICKLE ]
```

The Transition Trigger performs the same operation, but for the set of 'compiled transitions'. Recall that a transition has a more complicated form involving component mode assignments as well as control variable assignments. However, each of these are known at the time that the mode estimates are determined. The process of triggering the proper compiled transitions is to determine if all the fields of a transition are in the list of previous component mode assignments, $m^{(t)}$, and commands, $\mu^{(t)}$.

The final step to the Compiled Conflict Recognition algorithm is the Constituent Diagnosis Generator. This algorithm maps the Enabled Dissents and Enabled Transitions to the output 'Constituent Diagnoses, 'Reachable Current Modes' and 'Enabled Transitions'. The Enabled Dissents map to the Constituent Diagnoses, and the Enabled Transitions map to the Reachable Current Modes.

6.3.1   Dissent and Transition Trigger Basics

The dissents and transitions are triggered incrementally, using the standard methods used for rule-based and truth maintenance systems. In particular,

the method employed is to maintain counters on the dissents and transitions that maintain a record of the unsatisfied antecedents. In the case of a dissent, there is a counter for the observations. For a transition, there are three different counters, one for the component mode assignment in the source of the transition, one for the control variable assignments and one for the component mode assignments in the constraint of the transition.

For the purposes of example and simplicity, the triggering process is described using dissents. The process is easily extended to transitions by simply repeating the process for the different types of variables in the transition.

As an example, consider a subset of the dissents generated from the system described in Chapter 1, with the full list of dissents given in Appendix A. The counters of the dissents are shown on the right, with the number of observations in the antecedent shown first, followed by the number of observation variables not in the current list of observations. So, the 1:1 is interpreted to mean that the dissent has one observation assignment, and that this assignment is not in the current list of observations. A 1:0 would indicate that the dissent has one observation assignment, and that the observation is in the current list of observations.

```
    [ ]  ⇒ ¬[ SWITCH = STUCK-CHARGER-1 ∧ CHARGER-2 = FULL-ON]
    [ ]  ⇒ ¬[ SWITCH = STUCK-CHARGER-1 ∧ CHARGER-2 = TRICKLE]
    [ ]  ⇒ ¬[ SWITCH = CHARGER-2 ∧ CHARGER-1 = FULL-ON]
    [ ]  ⇒ ¬ [ SWITCH = CHARGER-2 ∧ CHARGER-1 = TRICKLE]
    [ BATTERY-TEMPERATURE = HIGH ] ⇒ ¬ [ BATTERY = CHARGING ]
    [ BATTERY-VOLTAGE = ZERO ] ⇒ ¬ [ BATTERY = CHARGING ]
    [ BATTERY-TEMPERATURE = LOW ] ⇒ ¬ [ BATTERY = FULL ]
    [ BATTERY-TEMPERATURE = NOMINAL ] ⇒ ¬ [ BATTERY = FULL ]
    [ BATTERY-VOLTAGE = NOMINAL ] ⇒ ¬ [ BATTERY = DISCHARGING
]    1:1
    [ BUS-VOLTAGE = LOW ] ⇒ ¬ [ SWITCH = CHARGER-1 ∧ CHARGER-1
= OFF ]
```

```
    [ BUS-VOLTAGE = NOMINAL ]  ⇒  ¬[ SWITCH = CHARGER-1 ∧
CHARGER-1 = OFF ]
    [ BUS-VOLTAGE = ZERO ]  ⇒  ¬[ SWITCH = STUCK-CHARGER-2 ∧
CHARGER-2 = FULL-ON ]
    [ BUS-VOLTAGE = LOW ]  ⇒  ¬[ SWITCH = STUCK-CHARGER-2 ∧
CHARGER-2 = OFF ]
```

Figure 6-4 - Sampling of Dissents of the NEAR Power Storage System

A dissent is triggered by determining if each observable in the antecedent is
in the current list of observations. This is implemented efficiently using a
counter discipline. Each dissent is given a counter, initialized to the
number of its antecedents. For each observation assignment in the current
list of observations, the counter for all dissents that mention that
observation are decremented. If the counter on a dissent goes to zero, then it
is triggered. Given the observations:

(*bus-voltage = nominal*), (*battery-temperature = low*), (*battery-voltage = nominal*)

These observations would trigger the following dissents since their counters
go to zero.

```
    [ ]  ⇒  ¬[ SWITCH = STUCK-CHARGER-1 ∧ CHARGER-2 = FULL-ON]
    [ ]  ⇒  ¬[ SWITCH = STUCK-CHARGER-1 ∧ CHARGER-2 = TRICKLE]
    [ ]  ⇒  ¬[ SWITCH = CHARGER-2 ∧ CHARGER-1 = FULL-ON]
    [ ]  ⇒  ¬[ SWITCH = CHARGER-2 ∧ CHARGER-1 = TRICKLE]
    [ BATTERY-TEMPERATURE = LOW ]  ⇒  ¬[ BATTERY = FULL ]
    [ BATTERY-VOLTAGE = NOMINAL ]  ⇒  ¬[ BATTERY = DISCHARGING ]
    [ BUS-VOLTAGE = NOMINAL ]  ⇒  ¬[ SWITCH = CHARGER-1 ∧
CHARGER-1 = OFF ]
```

Figure 6-5 - Triggered Dissents from Observations

These dissents are placed in the list of enabled dissents, D_{EN}. The triggering
of the proper dissents is performed with efficiency in mind since the

Compiled Mode Estimation process is designed for real time systems. There are two outstanding issues. First is to know not just when to decrement the counts in a dissent or transition, but to also increment the counts. The second is to avoid iterating through all of the dissents and transitions when decrementing and incrementing the counts. The approach to handling these nuances is demonstrated using the above example.

A count is decremented or incremented only when an observation variable has changed its value from time step 't' to 't+1'. For example, if the *bus-voltage* had the value '*nominal*' at time 't', and then '*low*' at time 't+1', then any dissents mentioning the assignments '*bus-voltage = nominal*' would be incremented, and those mentioning '*bus-voltage = low*' must be decremented. Knowing when a variable has changed values then requires maintaining a previous truth value and a current truth value within the variable that signals if it has changed values. Then the algorithm can increment and decrement the dissent counters based on the truth-values of a particular assignment. To illustrate this, consider the two sets of observable values below.

   Previous: (*bus-voltage = low*), (*battery-temperature =nominal*), (*battery-voltage = nominal*)

   Current: (*bus-voltage = nominal*), (*battery-temperature = low*), (*battery-voltage = nominal*)

The truth values for these observations in the current time step would be:

Truth Value	*bus-voltage = nominal*	*bus-voltage = low*	*battery-temperature = low*	*battery-temperature = nominal*	*battery-voltage = nominal*
Previous	false	true	false	true	true
Current	true	false	true	false	true

Table 6-1 - Example of Truth values for Assignments

From this table, the algorithm would then increment any dissent that mentions the observable values (*bus-voltage = low*) and (*battery-temperature = nominal*), and decrement any dissents mentioning (*bus-voltage = nominal*) and (*battery-temperature =low*). The algorithm would not bother changing the counters for the observable variable '*battery-voltage*' since its value did not change from the previous time step to the next.

Finally, to update the dissents and transitions, it is inefficient to iterate through the complete list in a brute force fashion. Instead, only the dissents that mention the changed observation variables need to be updated. Assuming that an observation assignment has a link to the dissents that mention it, all that is required is to iterate through the list of changed observations, and increment or decrement the linked dissents.

This completes the description of the triggering process for dissents. This triggering is extended to transitions by simply updating the truth-values for control variables in the same way as for observation variables. For component mode variables, the truth-values are updated using the list of 'previous mode estimates'. The steps of the algorithm for triggering are described below.

1. Update truth values of
    a. all $x_{io} \in \Pi_o$ using the current set of observations
    b. all $x_{ic} \in \Pi_c$ using the current set of commands
    c. all $x_{im} \in \Pi_m$ using the previous mode estimates
    d. Create lists OBS, CMDS, and MODE$_{Prev}$, that represent the lists of assignments that have changed
2. For each $x_{io} = v_{ij} \in$ OBS

a. Increment or Decrement all OBS counters in dissents that mention $x_{io} = v_{ij}$

b. Increment or Decrement all OBS counters in transitions that mention $x_{io} = v_{ij}$

3. For each $x_{ic} = v_{ij} \in$ CMDS

a. Increment or Decrement all CMD counters in transitions that mention $x_{ic} = v_{ij}$

4. For each $x_{im} = v_{ij} \in$ MODE$_{Prev}$

a. Increment or Decrement all source mode counters in transitions that mention $x_{im} = v_{ij}$

b. Increment or Decrement all mode counters for constraints in transitions that mention $x_{im} = v_{ij}$

5. Determine which Dissents have '*counter = 0*', and put them in D$_{EN}$

6. Determine which Transitions have '*counter = 0*' for the source, observations, command and mode variable counters, and put them in T$_{EN}$

The above steps outline the Dissent and Transition triggering algorithms, creating the lists of enabled dissents, D$_{EN}$, and enabled transitions, T$_{EN}$. Along with the previous mode estimates, these outputs are used in the Constituent Diagnosis Generator to determine the constituent diagnoses, the reachable current modes and the enabled transitions.

6.3.2 Constituent Diagnosis Generator

The final step in Compiled Conflict Recognition is to use the enabled dissents and transitions from the dissent and transition triggers to create a list of constituent diagnoses and the set of reachable current modes. First the transformation of a dissent to a constituent diagnosis is presented,

followed by the mapping of enabled transitions and previous mode estimates to the set of reachable current modes.

The consequent of a dissent represents an infeasible space of assignments. This can be turned around to describe the remaining feasible assignments. The constituent diagnoses are generated by logically transforming the conflict. The logical transformation is as follows.

$$\left[\left(x_{1o}=v_{1l_1}\right)\wedge\left(x_{2o}=v_{2l_2}\right)\right] \Rightarrow \neg\left[\left(x_{1m}=v_{1l_1}\right)\wedge\left(x_{2m}=v_{2l_1}\right)\wedge\left(x_{3m}=v_{3l_2}\right)\right]$$

or by example

$$\left[\left(bus\text{-}voltage = nominal\right)\right] \Rightarrow \neg\left[\left(switch = charger\text{-}1\right)\wedge\left(charger\text{-}1 = off\right)\right]$$

Equation 6-1 - Logical Statement of a Dissent

Assuming that this dissent has been enabled, then the consequent is a conflict:

$$\neg\left[\left(x_{1m} = v_{1l_1}\right)\wedge\left(x_{2m} = v_{2l_1}\right)\wedge\left(x_{3m} = v_{3l_2}\right)\right]$$

or by example

$$\neg\left[\left(switch = charger\text{-}1\right)\wedge\left(charger\text{-}1 = off\right)\right]$$

In clausal form, these are equivalent to:

$$\neg\left(x_{1m} = v_{1l_1}\right)\vee\neg\left(x_{2m} = v_{2l_1}\right)\vee\neg\left(x_{3m} = v_{3l_2}\right)$$

or by example

$$\neg\left(switch = charger\text{-}1\right)\vee\neg\left(charger\text{-}1 = off\right)$$

These statements logically say that the variables cannot all have the values specified here. So, the '*switch*' cannot have the value *charger-1* at the same time that the '*charger-1*' is *off*. However, the variables can take on any other value in its domain. So, the following is the logical equivalent of the above statements.

$$\left(x_{1m} = v_{1l_2}\right)\vee\left(x_{2m} = v_{2l_2}\right)\vee\left(x_{2m} = v_{2l_3}\right)\vee\left(x_{3m} = v_{3l_1}\right)\vee\left(x_{3m} = v_{3l_3}\right)\vee\left(x_{1m} = v_{1l_3}\right)$$

or by example

$$\left(switch = charger\text{-}2\right)\vee\left(charger\text{-}1 = full\text{-}on\right)\vee\left(charger\text{-}1 = trickle\right)\vee\ldots$$

Equation 6-2 - Final Statement after Logical Transformation

The clause here is represented by the constituent diagnoses, defined as a set of component mode assignments in Equation 6-2. Each assignment in the set is referred to as a constituent diagnosis of the conflict because each

146   Achieving Real-time Mode Estimation through Offline Compilation

assignment resolves the conflict.   The set of constituent diagnoses represents a single conflict, so the '*Constituent Diagnoses*' is represented as a set of sets of constituent diagnoses of the form defined in Equation 6-2.

The final step of the Constituent Diagnosis Generator is to generate a list of reachable current modes using the enabled transitions, and to determine the likelihood of these assignments. This likelihood is taken from the transition probability specified on component mode assignments. After determining the enabled transitions, the set of reachable component modes is generated using the previous mode estimates and identifying the enabled transitions where a component mode assignment in the source is also in the previous mode estimate. A component mode assignment in the set of reachable component mode assignments is the target of these enabled transitions. Figure 6-6 depicts the calculation.
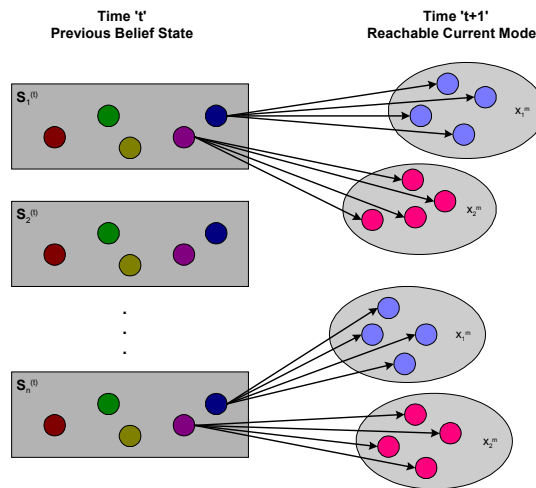
Figure 6-6 – Calculation of the Reachable Current Modes

The figure denotes different component mode assignments in the previous mode estimate $S_i^{(t)}$.   Shown are the transitions from two different components in each mode estimate, and from two different mode estimates

in the previous belief state. The Constituent Diagnosis Generator then adds the component mode assignments on the right of the figure to the set of *reachable current modes*. So, the component mode assignments for x_{1m} and x_{2m} that are reachable from the previous mode estimates $S_1^{(t)}$ and $S_n^{(t)}$ are added to the list of *reachable current modes*. There is a complication related to overlap of the reachable component modes generated from different previous mode estimates. In determining the *reachable current modes*, there is nothing to preclude two previous mode estimates from having transitions to the same current mode. When this occurs, the transitions are maintained separately. This enables the next phase of CME to compute the current belief state using the individually stored transitions. The approach to dealing with the overlap of reachable component modes is addressed in the detailed algorithms of Chapter 7.

Similarly to Livingstone, the set of reachable component modes is computed from each previous mode estimate using the enabled transitions. Each component mode assignment in the *reachable current modes* represents the transition using the probability of the transition and the previous mode estimate that is the source of this transition. The transition probability for the component mode assignment is given by the following equation:

$$P\left(x_{im} = v_{ij}\right) = P_{T_i^k\left(x_{im}=v_{ij}\right)} \bullet P_g\left(T_i^k\left(x_{im} = v_{ij}\right) \mid S_i^{(t)}\right)$$

Equation 6-3 - Probability Equation for Assignment Estimation

Here, the probability of a component mode assignment is dependent on the transition probability, P_T, and the guard probability, P_g. The guard probability is 1 or 0 depending on whether or not the guard is satisfied. The notation for 'P_g' is necessary to note that the transition probability is dependent on the entire state '$S_i^{(t)}$', including all, commands and previous

component mode assignments. The union of the pairs of component mode assignments and the associated probabilities, $\langle x_{im} = v_{ij}, p_{ij} \rangle$ where $x_{im} = v_{ij}$ is the target of the transition and p_{ij} represents the probability calculated in Equation 6-3 comprise the set of reachable current modes.

The following are the steps of the algorithm for the Constituent Diagnosis Generator.

1. For each 'dissent' in the Enabled Dissents
   a. Transform the consequent of each dissent to a constituent diagnosis, and place in the set CD
2. For each 'transition' in the Enabled Transitions
   a. Create a list of reachable current modes with the proper cost per Equation 6-3
3. Return the set CD, the Reachable Current Modes, and the Enabled Transitions

This completes the basic description of the Compiled Conflict Recognition algorithm design and computations that map the compiled knowledge to the set of constituent diagnoses, reachable current modes and the enabled transitions. The next step in the process of Online Mode Estimation is to use these to determine consistent mode estimates.

6.4   Dynamic Mode Estimate Generation

The previous sections have laid the foundation for Compiled Mode Estimation. Section 6.1 presented the overall architecture, and Section 6.2 gave the definitions of the inputs and outputs of the Online Mode Estimation process. Section 6.3 developed the approach to determining the conflicts relevant to the current observations, and the set of component modes that are reachable from the previous belief state. This section details the approach to tracking the approximate belief state over time. The

Dynamic Mode Estimate Generation (DMEG) algorithms track the approximated belief state by enumerating the most likely transitions from mode estimates in the previous belief state. DMEG uses the conflicts from the Compiled Conflict Recognition process to ensure that mode estimates are consistent with the current observations.

The Dynamic Mode Estimate Generation algorithms are developed by first presenting the architecture in Section 6.4.1 and then developing the general approach of DMEG in Section 6.4.2. Each phase of the DMEG process is described in Sections 6.4.3 through 6.4.5. The chapter concludes with a mapping of CME to the ME-CCA algorithm described in Chapter 4.

6.4.1 Architecture

Dynamic Mode Estimate Generation (DMEG) is broken into three pieces, Generate, CDA* and Rank. The architecture of DMEG is shown below in Figure 6-7. The description of the Dynamic Mode Estimate Generation algorithm then proceeds by describing the Generate algorithm, followed by Conflict-Directed A* Search, and then ending with the Rank algorithm. Interleaved in each section are examples to demonstrate the steps of the algorithm and show the mapping of inputs to outputs intuitively.
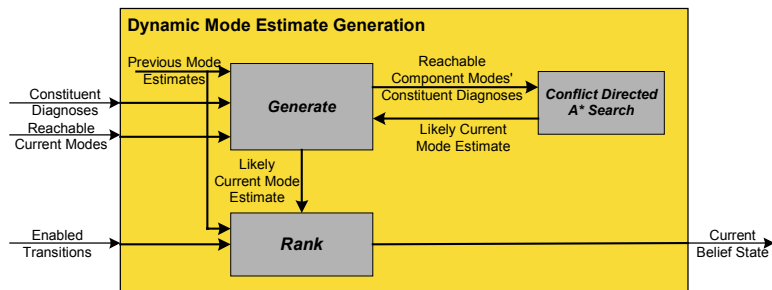
Figure 6-7 - Dynamic Mode Estimate Generation Architecture

The inputs of DMEG that have been previously defined include the constituent diagnoses, the reachable current modes, the enabled transitions, and the set of previous mode estimates. This section focuses on the remaining elements in the architecture, the 'current belief state', the 'likely current mode estimate', and the 'reachable component modes*'.

The current belief state is defined as the set of pairs, $\langle S_i^{(t+1)}, P^\bullet(S_i^{(t+1)}) \rangle$, where each $S_i^{(t+1)}$ is consistent with the observations at time '$t+1$' and commands given between time 't' and '$t+1$' and $P^\bullet(S_i^{(t+1)})$ is the posterior probability as given by the belief update equations. The 'likely current mode estimate' is defined as the pair $\langle S_i^{(t+1)}, {}^\bullet P(S_i^{(t+1)}) \rangle$. However, ${}^\bullet P(S_i^{(t+1)})$ denotes the probability of the mode estimate from CDA*. This probability is updated to the posterior probability, $P^\bullet(S_i^{(t+1)})$, in the Rank algorithm. The state, $S_i^{(t+1)}$ that is returned from the CDA* algorithm has the highest ${}^\bullet P(S_i^{(t+1)})$ of all states remaining in the search.

The set of '*reachable component modes*' is a mapping of the set of 'Reachable Current Modes' to a reduced set of component mode assignments. The Generate algorithm determines this reduced set of component mode assignments for the CDA* algorithm. The set is reduced to denote that not all component mode assignments in the set of Reachable Component Modes appear in the set of reachable component modes*.

6.4.2   Dynamic Mode Estimate Generation at a Glance

DMEG is tasked with determining a current belief state from a previous belief state, requiring tracking multiple mode estimates at every time increment. The approach to mapping the previous mode estimates to the current belief state is a 'generate-and-rank' approach where mode estimates are generated using the 'Generate' and 'CDA*' algorithms, and then ranked

by their posteriori probability in the current belief state by the 'Rank' algorithm.

The combination of the Generate and CDA* algorithms can be related back to the Livingstone approach for generating mode estimates. The Generate and CDA* algorithms combine to choose likely transitions from previous mode estimates to current mode estimates. This is exactly the Livingstone process of generating the likely mode estimate, without the need for satisfiability. So, the Generate and CDA* algorithms are considered as multiple instances of Livingstone, one for each previous mode estimate. Figure 6-8 demonstrates the desired calculation of Generate and CDA*, with the approximated belief state maintained by the DMEG algorithm in white.
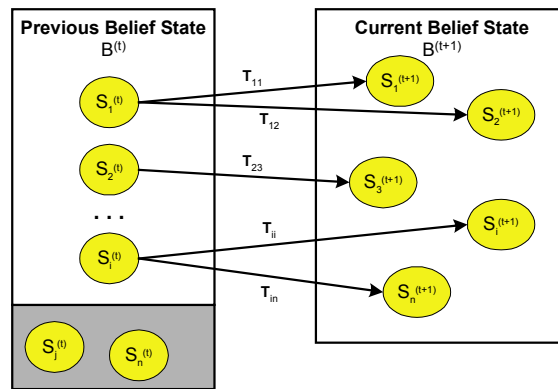
Figure 6-8 - Depiction of Generate and CDA* Result

The Generate and CDA* algorithms choose the transitions, T_{ij}, from mode estimates in the approximated previous belief state to mode estimates in the approximated current belief state. The approach is to choose a previous mode estimate from the previous belief state, and then determine its most likely transition to a current mode estimate in the current belief state. The resultant probability of a current mode estimate is then the probability of

the transition multiplied by the probability of the previous mode estimate. For instance, $P(S_3^{(t+1)}) = P(S_2^{(t)}) \bullet P(\mathbf{T}_{23})$.

The next step of the DMEG algorithm is to determine the probability of a current mode estimate from every previous mode estimate. This step is necessary to determine the posterior probability of the current mode estimate given by the belief update equations (Equations 3-1). The Generate and CDA* algorithms do not determine this. The calculation of the Rank algorithm is depicted below:
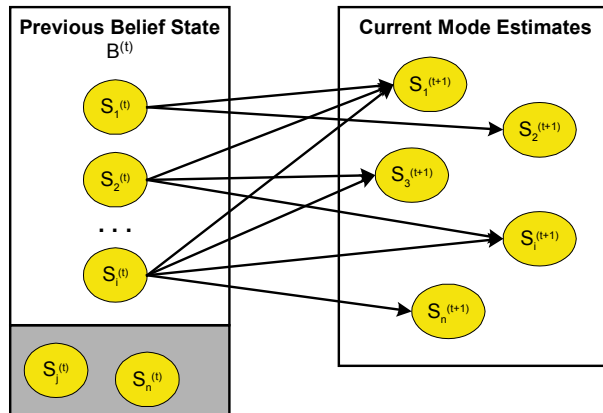
Figure 6-9 - Calculation of the Rank Algorithm

Denoted here, is the determination of all possible transitions to a current mode estimate from the previous belief state. The Rank algorithm determines the transitions from all $S_i^{(t)}$ to a particular $S_j^{(t+1)}$ to compute the posterior probability of $S_j^{(t+1)}$, given by the standard belief update equations. The posterior probability is then used to rank the current mode estimates in order of decreasing probability.

To summarize, the DMEG process of 'generate-and-rank' performs the following three steps to determine the current belief state:

6.1   Choose a previous mode estimate, $S_i^{(t)}$ in the previous belief state (Generate algorithm)

6.2   Choose the most likely transition from Si(t) to a current mode estimate, Sj(t+1) that resolves all conflicts (CDA* algorithm)

6.3   Determine all transitions from the previous belief state to the current mode estimate Sj(t+1) to calculate the posterior probability (Rank algorithm)

These three algorithms are the approach used within CME to calculate mode estimates and rank them in the current belief state. The following sections detail these algorithms, beginning with the Generate algorithm in Section 6.4.3, followed by the CDA* algorithm in 6.4.4 and concludes with the Rank algorithm in Section 6.4.5.

6.4.3   Generate Algorithm

The first step of Dynamic Mode Estimate Generation is the 'Generate' algorithm. The main task is to choose a previous mode estimate from the previous belief state. The goal of DMEG is to generate current mode estimates in a best-first order. In order to generate a mode estimate, a previous mode estimate is chosen, and then the most likely transition from the previous mode estimate is chosen by the CDA* algorithm. However, in order to find the current mode estimates, the Generate algorithm must choose previous mode estimates that lead to the likely current mode estimates.

One approach is to choose the previous mode estimates that have a high probability in the previous belief state. This could result in high probability current mode estimates. For instance choosing state $S_i^{(t)}$ with probability 0.7 results in $S_j^{(t+1)}$ with a transition probability of 0.7. However, this could

also result in low probability mode estimates. For example, choosing state $S_i^{(t)}$ with probability 0.7 could result in transitioning to $S_m^{(t+1)}$ with probability 0.01, but choosing state $S_k^{(t)}$ with probability 0.3 could result in transitioning to $S_p^{(t+1)}$ with probability 0.7. A better approach would be to have a metric that represents the likelihood of a previous mode estimate transitioning to the current mode estimates. This metric could then be used as a selection criterion to choose the previous mode estimates.

An additional role of the Generate algorithm is to pass along the set of constituent diagnoses to the Conflict-Directed A* algorithm, and to pass along the likely current mode estimate from the Conflict-Directed A* algorithm to the Rank algorithm. This section develops the approach the Generate algorithm uses to select the previous mode estimate, with the detailed algorithm and implementation details given in Chapter 7.

6.4.3.1 Generate Overview

Choosing the previous mode estimate is framed as a specialized tree search problem. The search tree is depicted in Figure 6-10. From the root of the tree, the previous mode estimates are expanded in the first level. From each previous mode estimate, $S_i^{(t)}$, a set of reachable current mode estimates, $S_j^{(t+1)}$ is expanded. The task of the Generate algorithm is to find a path from the root to a leaf that is the most probable.
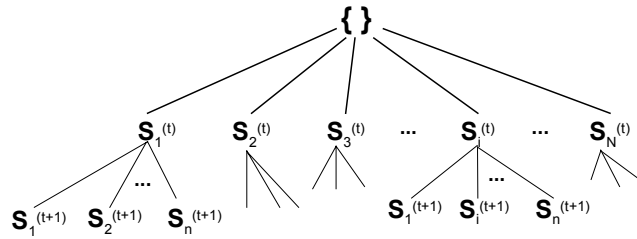
Figure 6-10 - Search Tree of Previous Mode Estimates

Achieving Real-time Mode Estimation through Offline Compilation   155

Choosing a previous mode estimate requires a cost that represents the probability of transitioning to reachable current mode estimates that have not been enumerated. The cost is associated with nodes in the tree, and is defined using the probability of the current mode estimate, so that a high cost represents a highly likely current mode estimate. If a previous mode estimate has generated high probability current mode estimates, then choosing that previous mode estimate may continue to generate high probability current mode estimates. Tree search offers a systematic way to choose the high cost node after calculating the cost of the nodes.

The cost of a node is the sum of the probability of transitioning to a current mode estimate plus a residual. The transition probability is a lower bound on the cost, while the residual is an upper bound. The residual represents the probability of transitioning to any current mode estimate in the belief state that has not been enumerated. The sum then represents the potential of the previous mode estimate to transition to high probability current mode estimates.

To calculate the lower bound of the cost, recall step 2 of DMEG in Section 6.4.2 where CDA* was used to choose the most likely transition from a previous mode estimate. This transition probability is multiplied by the probability of the previous mode estimate to give the lower bound. For instance, if the previous mode estimate $S_1^{(t)}$ has a probability of 0.5, and transitions to $S_{1-1}^{(t+1)}$ with a probability of 0.3, then the lower bound is 0.15.

The residual or upper bound is calculated using the results of the Rank algorithm. The Rank algorithm is called each time a current mode estimate is generated by the CDA* algorithm to determine transitions to the current mode estimate from all previous mode estimates. This probability is used to continually update the residual as current mode estimates are generated.

For instance, if the Rank algorithm updated the probability of $S_1^{(t+1)}$ to be 0.25, then the residual is $1 - 0.25 = 0.75$, assuming that $S_1^{(t+1)}$ is the only current mode estimate in the tree. Then the cost of the node for $S_1^{(t+1)}$ under $S_1^{(t)}$ is then 0.9. The cost is only associated with the previous mode estimate that was used to generate the current mode estimate.

The relevant formulae for calculating the cost of a node are given below. The first equation denotes the probability of a current mode estimate using the transition probability determined by CDA* and the probability of the previous mode estimate. The second value represents the posterior probability of the current mode estimate. This is used to calculate the residual probability remaining in the current belief state using the mode estimates that have been generated. The final equation is the cost, denoted as the sum of equations 1 and 3.

$$P\left(S_j^{(t+1)}\right) = P\left(S_i^{(t)}\right) \bullet P_{T_{S_i^{(t)} \to S_j^{(t+1)}}} \qquad \text{(from CDA* algorithm)}$$

$$P^\bullet\left(S_j^{(t+1)}\right) = \sum_{S_k^{(t)} \in B^{(t)}} P\left(S_k^{(t)}\right) \bullet P_{T_{S_k^{(t)} \to S_j^{(t+1)}}} \quad \text{(from Rank Algorithm)}$$

$$R = 1 - \sum_{S_j^{(t+1)} \in B^{(t+1)}} P^\bullet\left(S_j^{(t+1)}\right)$$

$$Cost\left(S_j^{(t+1)}\right) = P\left(S_j^{(t+1)}\right) + R$$

Equation 6-4 - Cost Equations for the Generate Algorithm

The Generate algorithm chooses the node in the search tree with the highest cost, representing the highest likely current mode estimate in the search tree. This guides the Generate algorithm to choose the previous mode estimate that is the parent of this node. For instance, from the above tree, if $S_2^{(t+1)}$ has the best cost of 0.9, the Generate algorithm chooses $S_1^{(t)}$ for CDA* to pick its next most likely transition. This results in generating node $S_3^{(t+1)}$ with a cost of 0.6. Next, suppose that this cost is less than the

cost of node $S_{4-1}^{(t+1)}$. The Generate algorithm would then choose $S_4^{(t)}$ for CDA* to pick its next most likely transition.

A consequence of choosing a previous mode estimate is that now, not all component mode assignments in the set of Reachable Current Modes are necessarily reachable from $S_i^{(t)}$. Recall Figure 6-6 that determined the Reachable Current Modes from all previous mode estimates as a union. The set of reachable component modes from any one previous mode estimate is a subset of this union. The component mode assignments that are not reachable from a previous mode estimate must be removed from the set of Reachable Current Modes. These are now stored in the set of '*reachable component modes*''. Consider the example mode estimates:

$S_1^{(t)}$: (*switch = charger-1*), (*charger-1 = trickle*), (*charger-2 = off*), (*battery = charging*)

with $P(S_1^{(t)}) = 0.9$

$S_2^{(t)}$: (*switch = stuck-charger-1*), (*charger-1 = trickle*), (*charger-2 = off*), (*battery = charging*) with $P(S_2^{(t)}) = 0.1$

The set of 'Reachable Current Modes' for these two mode estimates is then:

  (*switch = charger-1*), (*switch = stuck-charger-1*), (*switch = stuck-charger-2*),

  (*switch = broken*), (*switch = unknown*)

Only the switch modes are shown, as the rest of the reachable component modes would be the same. The set of Reachable Current Modes contains more component modes than are reachable from $S_2^{(t)}$. For instance, the component mode (*switch – charger-1*) is not reachable from the failure mode (*switch = stuck-charger-1*). The Generate algorithm would then reduce the set of mode assignments for the switch to (*switch = stuck-charger-1*) and (*switch = unknown*) for mode estimate $S_2^{(t)}$.

These are the key steps that enable DMEG, and CME, to track the approximated belief state over time. The Generate algorithm, by choosing a

previous mode estimate, enables the CDA* algorithm to choose the most likely transition from the previous mode estimate. The Generate algorithm is demonstrated through a simple example in the next section.

6.4.3.2   Generate Algorithm Example

The example in Figure 6-11 denotes a set of previous mode estimates, the transitions, and the current mode estimates. The probabilities associated with the previous mode estimates are shown to the left of the diagram, the transition probability is noted on the arc, and the probability of the current mode estimates are noted to the right of the figure. The current mode estimate probability was calculated using the standard belief update equation, which simplifies to the following for this example.

$$P\left(S_j^{(t+1)}\right) = P\left(S_i^{(t)}\right) \bullet P_{\tau_{S_t^i \to S_{t+1}^j}}$$

Equation 6-5 - Calculation of Current State Probability

For example, the first state, $S_1^{(t+1)}$ is calculated using $S_1^{(t)}$ and $S_2^{(t)}$. The probability of state $S_1^{(t+1)}$ is then $0.5 \times 1.0 + 0.3 \times 0.4 = 0.62$. The previous belief state, $B^{(t)}$ is ordered by decreasing probability, and the current belief state, $B^{(t+1)}$ is not ordered in any particular manner.
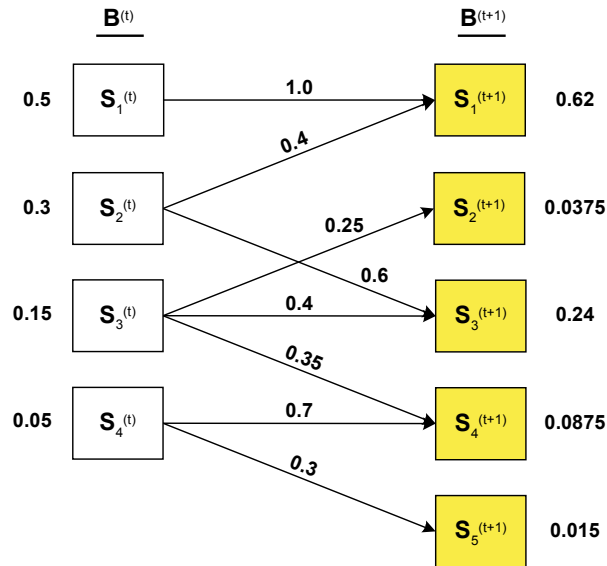
Figure 6-11 - Example of State Transitions for the Generate Algorithm

The previous mode estimates are used to expand the first level of the search tree in the Generate algorithm. Initially the tree is ordered according to the posterior probability of the previous mode estimate, depicted in Figure 6-12.
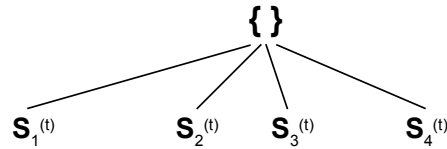
Figure 6-12 - Initial Ordering of the Search Tree in the Generate Algorithm

The Generate algorithm begins by choosing the most likely previous mode estimate, $S_1^{(t)}$ in this case, and chooses its most likely transition. This results in generating the mode estimate $S_1^{(t+1)}$ with a $P(\mathbf{T}_{11}) = 0.6$. The Rank algorithm then determines the posterior probability of the mode estimate to be 0.39, as shown in Figure 6-11. The Generate algorithm then calculates

the residual value, $R = 1 - 0.62 = 0.38$. The resulting cost of the node $S_1^{(t+1)}$ in the search tree is $C = P(S_1^{(t+1)}) + R = 0.50 + 0.38 = 0.88$. The search tree that results from this first iteration of the Generate algorithm is:

$$\{\}$$

$$S_1^{(t)} \quad S_2^{(t)} \quad S_3^{(t)} \quad S_4^{(t)}$$

$$S_1^{(t+1)}$$
$$L = 0.88$$
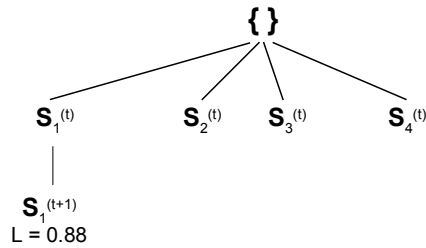
Figure 6-13 - Search Tree after 1st Iteration of the Generate Algorithm

The Generate algorithm then chooses $S_2^{(t)}$ as the previous mode estimate to generate its most likely transition, not $S_1^{(t)}$. The Generate algorithm first generates the most likely transition from each mode estimate in the approximated previous belief state so that the search is not biased towards highly likely previous mode estimates. The result of choosing $S_2^{(t)}$ is to choose its most likely transition, which results in generating $S_3^{(t+1)}$. This mode estimate is then ranked to give the posterior probability 0.24. The Generate algorithm then uses this value to update the residual to $R = 1 - 0.62 - 0.24 = 0.14$. The cost of the nodes are updated to obtain the search tree:

$$\{\}$$

$$S_1^{(t)} \quad S_2^{(t)} \quad S_3^{(t)} \quad S_4^{(t)}$$

$$S_1^{(t+1)} \quad S_3^{(t+1)}$$
$$L = 0.64 \quad L = 0.38$$
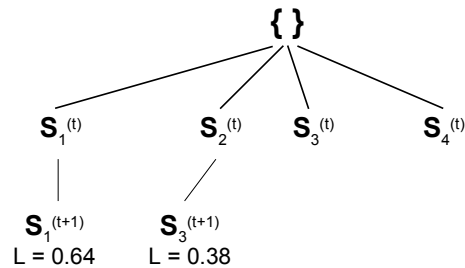
Figure 6-14 - Search Tree after 2nd Iteration of the Generate Algorithm

The Generate algorithm then proceeds to select the previous mode estimate $S_3^{(t)}$ to generate its most likely current mode estimate. This results in CDA* generating $S_3^{(t+1)}$ by taking the most likely transition $P(\mathbf{T}_{33}) = 0.4$. However, this mode estimate already exists in the current belief state, so the Generate algorithm only updates the cost of $S_3^{(t)}$ to obtain $C = R + P(S_3^{(t+1)}) = 0.14 + 0.06 = 0.2$. The resulting search tree is then:
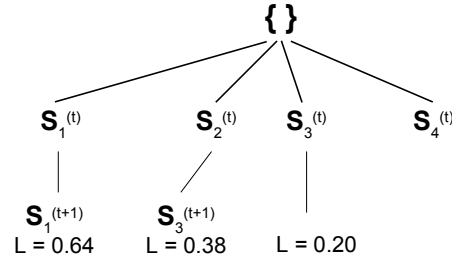
Figure 6-15 - Search Tree after 3rd Iteration of the Generate Algorithm

The Generate algorithm then proceeds to choose the previous mode estimate $S_4^{(t)}$ to generate the next current mode estimate. The result of choosing this mode estimate is to generate $S_4^{(t+1)}$ by choosing $P(\mathbf{T}_{44}) = 0.7$. The posterior probability of $S_4^{(t+1)}$ is updated by the Rank algorithm to obtain 0.0875. The Generate algorithm then updates the residual to obtain $R = 1 - 0.62 - 0.24 - 0.0875 = 0.0525$. The cost of the new node is then $C = P(S_4^{(t+1)}) + R = 0.035 + 0.0525 = 0.0875$. The remaining nodes in the search tree are also updated to obtain:
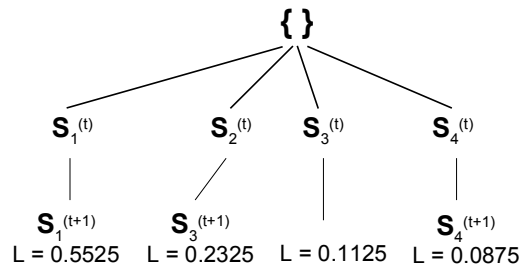
Figure 6-16 - Search Tree after 4th Iteration of the Generate Algorithm

The Generate algorithm then chooses the node with the highest cost to determine another consistent mode estimate. From Figure 6-16, the cost of $S_1^{(t+1)}$ is the highest, so the Generate algorithm chooses $S_1^{(t)}$. The result of choosing this does not generate a new current mode estimate since there is only one consistent current mode estimate from $S_1^{(t)}$ in Figure 6-11. The Generate algorithm then chooses the node with the next highest cost, in this case $S_2^{(t)}$. However, in choosing the most likely transition from $S_2^{(t)}$ results in $P(\mathbf{T}_{21}) = 0.4$. However, this transition results in generating $S_1^{(t+1)}$, which is in the current belief state. The Generate algorithm then only updates the cost of this node to $C = P(S_1^{(t+1)}) + R = 0.12 + 0.0525 = 0.1725$. The tree is updated to obtain:

{ }

$S_1^{(t)}$ $S_2^{(t)}$ $S_3^{(t)}$ $S_4^{(t)}$

$S_4^{(t+1)}$

L = 0.1725   L = 0.1125   L = 0.0875
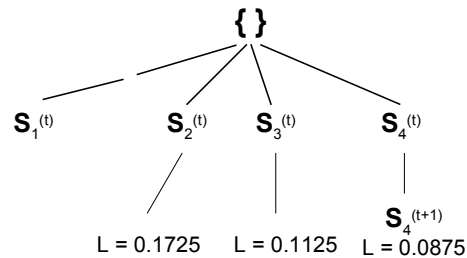
Figure 6-17 - Search Tree after 5th Iteration of the Generate Algorithm

The resulting tree no longer contains a link to the previous mode estimate $S_1^{(t+1)}$ because no more transitions to current mode estimates exists. The resulting search tree causes the Generate algorithm to choose $S_2^{(t)}$ as the highest cost node. When attempting to choose another likely transition, CDA* determines that there are no more consistent mode estimates from $S_2^{(t)}$. The result is to remove $S_2^{(t)}$ as a branch in the search tree. The Generate algorithm then chooses $S_3^{(t)}$ as the highest cost node and determines its most likely transition. This results in choosing the transition $\mathbf{T}_{34}$ and generating $S_4^{(t+1)}$. However, this current mode estimate has already been generated by $S_4^{(t)}$. The Generate algorithm then updates the cost of $S_3^{(t)}$ to obtain $C = P(S_4^{(t+1)}) + R = 0.0525 + 0.0525 = 0.105$. The resulting search tree is shown below.
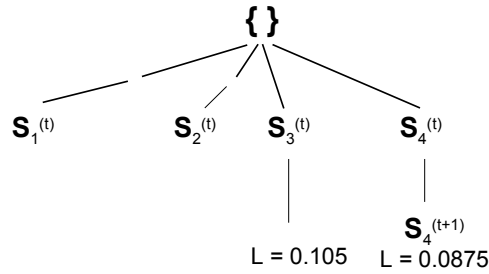
Figure 6-18 - Search Tree after 6th Iteration of the Generate Algorithm

The Generate algorithm again determines the node with the highest cost value, which is $S_3^{(t)}$ again. The result of choosing its next most likely transition, $\mathbf{T}_{32}$ results in generating the current mode estimate $S_2^{(t+1)}$. The Rank algorithm then determines the posterior probability of this mode estimate to be $P(S_2^{(t+1)}) = 0.0375$. The Generate algorithm then proceeds to update the residual value, resulting in $R = 1 - 0.62 - 0.24 - 0.0875 - 0.0375 = 0.015$. The node associated with $S_3^{(t)}$ and $S_4^{(t)}$ are updated to obtain $C = 0.0375 + 0.015 = 0.0525$ and $C = 0.035 + 0.015 = 0.0.05$, respectively. The search tree is updated to obtain:
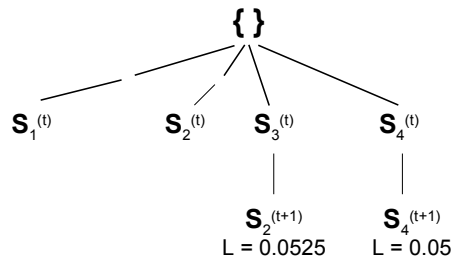
Figure 6-19 - Search Tree after 7th Iteration of the Generate Algorithm

The Generate algorithm determines that the highest cost mode estimate is again $S_3^{(t)}$. Upon determining a current mode estimate from $S_3^{(t)}$ results in CDA* discovering that there are no more transitions to consistent current mode estimates from $S_3^{(t)}$. As a result, the Generate algorithm removes $S_3^{(t)}$ from the tree, leaving only $S_4^{(t)}$. By choosing $S_4^{(t)}$, the CDA* identifies that

$S_5^{(t+1)}$ is the target of the most likely transition $\mathbf{T}_{45}$. The Rank algorithm then computes the posterior probability of $S_5^{(t+1)} = 0.015$, as shown in Figure 6-11. The Generate algorithm then updates the residual to obtain $R = 1 - 0.62 - 0.24 - 0.0875 - 0.0375 - 0.015 = 0.0$. The search tree is updated to obtain:
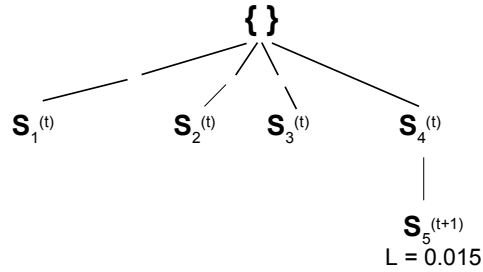
Figure 6-20 - Search Tree after 8th Iteration of the Generate Algorithm

The Generate algorithm then is only left to choose $S_4^{(t)}$. Upon attempting to determine its next most likely transition the CDA* cannot identify another consistent mode estimate from $S_4^{(t)}$. Without a new consistent mode estimate, the Generate algorithm removes $S_4^{(t)}$ from the search tree. There are no more nodes to explore in the tree, causing the algorithm to exit.

The example above demonstrates the steps of the Generate algorithm and its process of choosing a previous mode estimate by exploring the search tree. The Generate algorithm expands the most likely transition under each previous mode estimate first. This design choice enables the algorithm to track less likely trajectories of the system. This is beneficial since only the approximate belief state is tracked, so a less likely mode estimate may prove to be more likely in the future. Another characteristic of the Generate algorithm is that under each previous mode estimate, only a single node is maintained that represents the most recently generated current mode estimate from the previous mode estimate. This is done to reflect the likelihood of the remaining current mode estimates that are targets of the

previous mode estimate. For instance, $S_1^{(t)}$ has a high likelihood due to the high posterior probability of the current mode estimate $S_1^{(t+1)}$. It stands to reason that $S_1^{(t)}$ would produce more high probability mode estimates. So, the cost maintained for each node is designed to reflect this. Other methods for calculating the residual and updating the cost of nodes are discussed in Future Work.

6.4.3.3   Generate Algorithm

From the example above, an algorithm is extracted to perform these same steps. The full detail of the Generate Algorithm is given in Chapter 7. The following lists the steps of the algorithm.

1. Choose the highest cost node from the search tree. Nodes represent the current mode estimates

2. Choose the previous mode estimate, $S_i^{(t)}$, associated with the node.

3. Choose the most likely transition from $S_i^{(t)}$ using CDA*, giving a mode estimate, $S_j^{(t+1)}$ that satisfies all conflicts

4. Calculate the posterior probability of $S_j^{(t+1)}$ in the Current Belief state, $B^{(t+1)}$ using the Rank algorithm

5. Update the residual value, R, as described in the example above

6. Update the leaves in the tree, one for each previous mode estimate

Notice in step 6 that there is one branch maintained for each previous mode estimate. Any consistent mode estimates that have already been generated from a previous mode estimate are not considered. The cost is designed to reflect the likelihood of the remaining mode estimates that could be generated from a previous mode estimate. The algorithm then only needs to

consider the most recently generated branch from a previous mode estimate. Recall the basic premise of this algorithm was to choose a previous mode estimate until the likelihood of its most recently generated current mode estimate is lower than another previous mode estimate's most recently generated current mode estimate. In the example above, the algorithm chose '$S_3^{(t)}$' over '$S_4^{(t)}$' when its most recently generated mode estimate had a higher likelihood than the one generated from '$S_4^{(t)}$' ($L(S_2^{(t+1)}) = 0.0525$ vs. $L(S_4^{(t+1)}) = 0.050$).

The generate algorithm adheres to A* search by using an optimistic estimate to guide the ordering of nodes. The optimistic estimate is achieved through the use of the 'residual' probability to overestimate the true probability of a mode estimate. This overestimate guides the choice of a previous mode estimate to generate a current mode estimate. However, the search tree is not explored to completion, meaning that not all consistent current mode estimates are generated. The number of consistent current mode estimates is exponential, resulting in too many to track and calculate at each time increment. For instance, in the NEAR Power System, there are 4^{10} states ($\sim$ 1 million) states. To avoid this exponential search, the Generate algorithm uses halting conditions to stop the search. These conditions are detailed in Chapter 5.

The Generate algorithm makes use of other algorithms as well. In the steps of the algorithm above, the Conflict-Directed A* algorithm is used to generate consistent current mode estimates, and the Rank algorithm is used to determine the posterior probability of a mode estimate. These algorithms are detailed in sections 6.4.4 and 6.4.5, respectively.

6.4.4 Conflict-Directed A*

GDE, Sherlock and Livingstone all relied on the theory of conflict-directed A* search to solve the constraint satisfaction problem posed by model-based diagnosis. GDE and Sherlock used a modified A* search to determine diagnoses, while Livingstone used a modified A* search called Conflict Directed A* (CDA*) [*Williams, 2002*]. The search engine for Compiled Mode Estimation also uses conflict-directed A* search to solve the constraint satisfaction problem. In Compiled-ME, the constraints are represented by the set of dissents triggered by the Dissent Trigger. The role of the Conflict-Directed A* (DDA*) algorithm is to determine a set of component mode assignments that satisfy the conflicts encoded in the triggered dissents and that mode estimate generated are optimal solutions. CDA* offers fundamental theory to guarantee that solutions generated are optimal [*Williams, 2002*] and that the search guarantees systematicity [*Ragno, 2002*]

This section presents the formulation of Conflict Directed A* as a search, showing how the algorithm adheres to the theory of A* search. The heuristics for the A* search are presented first, followed by a description of the CDA* algorithm in Section 6.4.4.2. Section 6.4.4.3 then presents the algorithm, and the section ends with an example. The full algorithm is presented in Chapter 5.

6.4.4.1 CDA* Heuristics

Heuristics are the key to performing search. In order to gain the guarantees afforded by an A* search, the heuristics used must satisfy certain properties. The general equation for the A* search heuristic is represented in the following equation, from [*Russell, 1995*].

$$f(n) = g(n) + h(n)$$

Equation 6-6 – A* Heuristic Equation

The above equation represents the uniform cost heuristic, g(n), and the greedy cost heuristic, h(n). The uniform cost heuristic represents the best cost from the root of the tree to the node 'n'. The greedy cost heuristic is a value representing the best cost to the goal from the node 'n'.

Specific equations for these heuristics are dependent on the purpose and application of the search problem. In the case of mode estimation, the goal of the search is an assignment to each component mode variable in the system that is consistent with the system model and observations. Additionally, this set of component mode assignments maximizes the probability of each component mode variable. The search represents sets of assignments as paths through the search tree, linked by the branches. Recall from the development of the Compiled Conflict Recognition and Generate algorithms that the component mode assignments in the constituent diagnoses each have an associated cost, set to the transition probability. The search heuristic for mode estimation uses these probabilities to determine the likelihood of sets of component mode assignments. The CDA* search heuristic is based upon the same equation that is used in belief update for CCAs, shown below.

$$P_{\mathbf{T}}\left(S_i^{(t+1)}\right) = \prod_{\left(x_i = v_{ij}\right) \in S_i^{(t+1)}} P_{\mathbf{T}_i}\left(x_i = v_{ij}\right)$$

Equation 6-7 - CDA* Equation for Search Heuristics

As from belief update for CCAs, the probability of a mode estimate is determined from the probability of the transitions from a previous mode estimate to a current mode estimate. To note, this equation assumes that transitions between component mode assignments are independent of other components in the system. Since the goal of CDA* is to maximize the probability of Equation 6-7, if the search maximizes the probability of the

individual component mode assignments ($x_{im} = v_{ij}$), then this ensures that the highest estimate possible for the mode estimate is used. Using this, expressions for g and h are determined as follows.

$$g(n) \equiv \prod_{(x_{im} = v_{ij}) \in Node} P_{\mathbf{T}}(x_{im} = v_{ij})$$

$$h(n) \equiv \prod_{x_{im} \notin Node} \max(P(x_{im} = v_{ij}))$$

Equation 6-8 - CDA* Search Heuristics Defined

The above equations state in notation the following. The uniform cost heuristic is the probability of the assignments from the tree root to the node. This gives the lower bound on the probability of a node. The heuristic, $h(n)$, states that for all variables 'x_{im}' not currently assigned a value in the '*node*', choose its highest probability assignment 'v_{ij}' as the desired value. Then take the product of the probabilities of the assignments, $P((x_{im} = v_{ij}))$. This, along with $g(n)$ gives an upper bound on the probability of a node and includes an assignment to each component in the system. Take as an example the system described in Chapter 1.

$$Node : \left[(switch = Charger\text{-}1) \quad (charger\text{-}1 = Trickle) \right]$$
$$g(Node) = P(switch = Charger\text{-}1) \cdot P(charger\text{-}1 = Trickle)$$
$$h(Node) = P(charger\text{-}2 = Off) \cdot P(battery = Charging)$$

Figure 6-21 - Example Cost Calculation for a Node

The heuristic equations shown here are correct and adhere to the restrictions of heuristics for A* search. The $g(n)$ equation properly estimates the cost of a node from the root to a leaf in the tree, and the heuristic, $h(n)$ gives the desired over-estimate of the cost of the node to the goal. The heuristic is formulated to give the highest possible probability of the assignments in the node.

6.4.4.2   Conflict Direction and Systematicity

The CDA* algorithm relies on the input constituent diagnoses and the set of reachable component modes to enable the expansion of the search tree. At each level of the search tree, a set of constituent diagnoses is expanded. A set of constituent diagnoses corresponds one to one with each conflict, and choosing a component mode assignment from the constituent diagnoses resolves the conflict. The A* search is then conflict directed in the sense that the constituent diagnoses for a particular conflict are used to expand the nodes in the search tree. Recall from the example earlier in this chapter where the nodes expanded represented component mode assignments. The example expansion is given below.
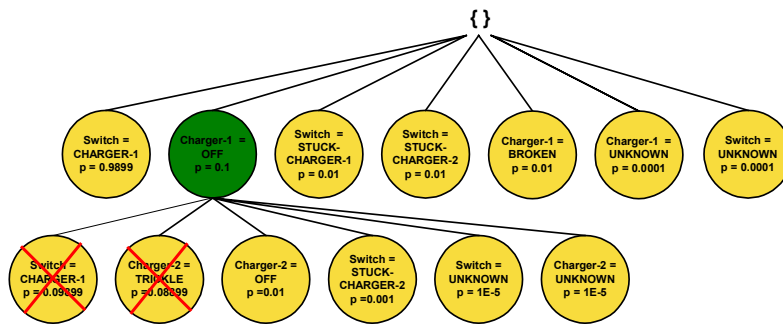
Figure 6-22 - Dissent Expansion of NEAR Power Storage System (Appendix C)

Figure 6-22 shows the expansion of the first constituent diagnosis of the example in Appendix C. The CDA* algorithm, by choosing the constituent diagnosis *charger-1 = off*, has satisfied the conflict associated with these constituent diagnoses. The next step of the CDA* algorithm is to determine other conflicts that this same assignment would satisfy. This requires determining if the constituent diagnosis *charger-1 = off* appears in other sets of constituent diagnoses triggered from Compiled Conflict Recognition. If it does appear as a constituent diagnosis, then the

Achieving Real-time Mode Estimation through Offline Compilation   171

constituent diagnoses do not need to be expanded under the branch. As an example, the constituent diagnosis *charger-1 = off* also satisfies the second conflict as it appears in the second set of constituent diagnoses. The relevant conflicts and constituent diagnoses are shown below.

Conflicts:

```
1. ¬ [ SWITCH = STUCK-CHARGER-2 ∧ CHARGER-1 = FULL-ON ]
2. ¬ [ SWITCH = STUCK-CHARGER-2 ∧ CHARGER-1 = TRICKLE]
3. ¬ [ SWITCH = STUCK-CHARGER-1 ∧ CHARGER-2 = FULL-ON]
```

Corresponding sets of constituent diagnoses:

```
1. [ SWITCH=CHARGER-1 ∨ SWITCH=CHARGER-2 ∨ CHARGER-1=TRICKLE ∨
   CHARGER-1=OFF ∨
   SWITCH=STUCK-CHARGER-1 ∨ CHARGER-1=BROKEN ∨ SWITCH=UNKNOWN
   ∨ CHARGER-1=UNKNOWN ]
2. [SWITCH=CHARGER-1 ∨ SWITCH=CHARGER-2 ∨ CHARGER-1=FULL-ON ∨
   CHARGER-1=OFF ∨
   SWITCH=STUCK-CHARGER-1 ∨ CHARGER-1=BROKEN ∨ SWITCH=UNKNOWN
   ∨ CHARGER-1=UNKNOWN ]
3. [ SWITCH=CHARGER-1 ∨ SWITCH=CHARGER-2 ∨ CHARGER-2=TRICKLE ∨
   CHARGER-2=OFF ∨ SWITCH=STUCK-CHARGER-2 ∨ CHARGER-2=BROKEN ∨
   SWITCH=UNKNOWN ∨ CHARGER-2=UNKNOWN ]
```

So, the CDA* algorithm does not expand these constituent diagnoses, and instead expands the third set of constituent diagnoses. This is denoted on Figure 6-22 as the expansion under the node *charger-1 = off*.

When the CDA* algorithm expands a set of constituent diagnoses, two operations are performed to guarantee systematicity. First, note on Figure 6-22 that the constituent diagnoses related to unreachable component mode assignments are not expanded from constituent diagnosis 3. For example, the assignment *switch = charger-2* and *charger-2 = trickle* are not allowed under the path for *charger-1 = off* because they are not in the set of reachable component modes.

Second, note on Figure 6-22 that the assignment *switch = charger-1* is not allowed under the *charger-1 = off* search path. This is because the assignment *switch = charger-1* is a sibling of *charger-1 = off* in the previous level of the search tree. CDA* maintains that siblings, and their children, cannot contain assignments to the left of the node. So, the result is that the children of the *charger-1 = off* mode assignment cannot contain the mode assignment *switch = charger-1* because it is a sibling on the left of *charger-1 = off*. Performing this computation enables the CDA* search to guarantee systematicity, as proven in [*Ragno, 2002*].

CDA* implements this by reducing the set of reachable component modes for each sibling node as the constituent diagnoses are expanded by placing assignments that are not allowed in a 'do-not-use' list of assignments. This 'do-not-use' list of assignments is then used to remove assignments from the reachable component modes that are associated with each node. As an example, the 'do-not-use' list of component mode assignments for the constituent diagnosis *charger-1 = broken* is:

{ *switch = charger-1*, *charger-1 = off*, *switch = stuck-charger-1*, *switch = stuck-charger-2* }

This list reduces the reachable component modes under the *charger-1 = broken* path to:

{ *switch = unknown*, *charger-2 = off*, *charger-2 = broken*, *charger-2 = unknown*, *battery =*

full, *battery = charging*, *battery = dead*, *battery = unknown* }

Note that the assignments for the *switch* have been reduced, and that assignments to *charger-1* are no longer allowed since it has been assigned a value. The 'do-not-use' list of component mode assignments is only used when constituent diagnoses are expanded. The list is cleared when all constituent diagnoses have been expanded.

CDA* must then compute the following at each expansion of a constituent diagnosis:

1. Use the 'do-not-use' list of component mode assignments to update the set of reachable component modes (note, initially the list is empty, but assignment are added as the constituent diagnoses are added to the search tree)

2. Determine if the constituent diagnosis is allowed for expansion by checking the set of reachable component modes

3. If the constituent diagnosis is allowed, then add it to the 'do-not-use' list of component mode assignments.

4. Determine all other conflicts that are satisfied by the constituent diagnosis.

Additionally, CDA* must compute the cost for each constituent diagnosis added to the search tree. The cost is calculated per the heuristic equations given in Equation 6-8. The calculation of the cost of each node guarantees that CDA* will find the optimal solutions with the fewest number of expansions [*Williams, 2002*]. Additionally, the expansion of constituent diagnoses described above guarantees systematicity [*Ragno, 2002*]. The CDA* algorithm that encompasses these capabilities is described in the next section.

6.4.4.3 CDA* Algorithm

The algorithm that explores the search tree described above for consistent sets of component mode assignments to constituent diagnoses is described in this section. The full algorithm description and implementation details are presented in Chapter 5. The search maintains the history and expansions of the tree by using a queue of nodes, where each node

represents the path from the root to the node. The algorithm is specified below:

CDA* (reachable component modes', set of constituent diagnoses)

1. Pop *node* from top of *queue*

2. Test *node*

    a. if assignments in the path from root to *node* resolve all current conflicts and make an assignment to all mode variables in the system, then return *node*

    b. if assignments in the path from root to *node* make an assignment to all mode variables but do not resolve all current conflicts, then explore siblings of *node*

    c. else GOTO 3

3. Expand *node*

    a. if there are no more *constituent diagnoses* to expand

        i. find a mode variable x_{im} that is unassigned in the path from root to *node*

        ii. expand node such that a *child* corresponds to a v_{ij} in the domain of x_{im}, and each *child* has a different v_{ij}.

        iii. for each *child* of *node*

            1. remove *child* $x_{im} = v_{ij}$ if not in the *reachable component modes'*

            2. if $x_{im} = v_{ij}$ is in the *reachable component modes*, then add $x_{im} = v_{ij}$ as a child of *node*

            3. calculate cost of *child* using Equation 6-6 and Equation 6-8

            4. insert *child* into *queue* in order of decreasing cost

    b. otherwise, choose a new set of constituent diagnoses, *cd*, and expand each constituent diagnosis as a *child* of *node*

c. for each *child*, constituent diagnosis ($x_{im} = v_{ij}$) of *node*

    i. remove assignments in '*do-not-use*' list of the current expansion from the *reachable component modes*

    ii. remove $x_{im} = v_{ij}$ if not in *reachable component modes*

    iii. add constituent diagnosis $x_{im} = v_{ij}$ to the '*do-not-use*' list

    iv. calculate the cost of *child* node $x_{im} = v_{ij}$ using Equation 6-6 and Equation 6-8

    v. insert *child* into *queue* in order of decreasing cost

d. return *queue*

e. GOTO 1

The algorithm as outlined above will first extract a node from the queue, the node with the highest cost, or best probability. The algorithm then tests the node to determine if it is complete, meaning that it has satisfied all conflicts and that it assigns to each component mode variable a value from its domain. If the set of assignments in *node* is not complete, the node is expanded. The expansion steps are as demonstrated previously. First a set of constituent diagnoses that remains is expanded. Each assignment in the constituent diagnoses is first checked to determine if it is allowable in this path of the tree. If the assignment is not in the *reachable component modes* list, then it is not expanded. If the assignment can be expanded, this is done by copying the node, adding the assignment to the node, and then updating the cost, or probability, of the node. This cost is calculated using Equation 6-8. Finally, the node is inserted in the queue by decreasing cost, or decreasing probability.

6.4.4.4   CDA* Example

The CDA* algorithm is best demonstrated by example using the NEAR Power storage system detailed in Chapter 1.   Consider the following previous mode estimate and observations:

   (*switch = charger-1*), (*charger-1 = full-on*), (*charger-2 = off*), (*battery = charging*)

   (*bus-voltage = nominal*), (*battery-voltage = nominal*), (*battery-temperature = nominal*)

The following is a sampling of the triggered dissents for this example, with the full list given in Appendix C.

```
1.  [ ]  ⇒ ¬ [ SWITCH = STUCK-CHARGER-2 ∧ CHARGER-1 = FULL-ON ]
2.  [ ]  ⇒ ¬ [ SWITCH = STUCK-CHARGER-2 ∧ CHARGER-1 = TRICKLE]
3.  [ ]  ⇒ ¬ [ SWITCH = STUCK-CHARGER-1 ∧ CHARGER-2 = FULL-ON]
4.  [ ]  ⇒ ¬ [ SWITCH = STUCK-CHARGER-1 ∧ CHARGER-2 = TRICKLE]

10. [ BATTERY-TEMPERATURE  =  NOMINAL  ]  ⇒  ¬  [ BATTERY  =
    DISCHARGING ]
11. [ BATTERY-VOLTAGE = NOMINAL ] ⇒ ¬ [ BATTERY = DISCHARGING ]
12. [ BATTERY-TEMPERATURE = NOMINAL ] ⇒ ¬ [ BATTERY = DEAD ]
```

The set of reachable component modes for the previous mode estimate are:
'*switch*' = { (*charger-1*, p = 0.9899), (s*tuck-charger-1*, p = 0.01), (s*tuck-charger-2*, p = 0.01),

            (*unknown*, p = 0.0001) }
'*charger-1*' = { (f*ull-on*, p = 0.8899), (o*ff*, p = 0.1), (*broken*, p = 0.01), (*unknown*, p = 0.0001) }
'*charger-2*' = { (*off*, p = 0.1), (*trickle*, p = 0.8899), (*broken*, p = 0.01), (*unknown*, p = 0.0001) }

'*battery*' = { (*full*, p = 0.499), (*charging*, p = 0.499), (*dead*, p = 0.001), (*unknown*, p = 0.0001) }

CDA* expands the constituent diagnoses from the first conflict, which result in:
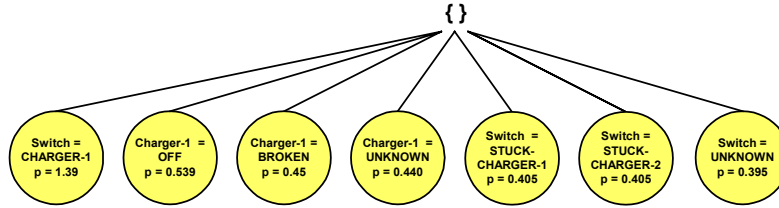
Figure 6-23 - CDA* Expansion of Constituent Diagnosis #1

The costs of each node are shown along with each assignment. As an example, consider the calculation of the cost of the *charger-1 = broken* node. The g(n) portion of the heuristic is given by the transition probabilities, so g(n) = 0.01. The h(n) portion is calculated using the highest probability mode assignments for the remaining components. So, h(n) uses *switch = charger-1*, *charger-2 = trickle*, and *battery = full* to determine that h(n) = 0.440. The resulting cost is the sum of g(n) and h(n) which is 0.450.

The CDA* algorithm chooses the highest cost node, which is *switch = charger-1*. This constituent diagnosis also satisfies conflicts 2, 3, and 4 shown above, as well as conflicts 5, 6 and 16 through 21 out of 21 conflicts, shown in Appendix C. Upon choosing to expand this node, CDA* determines that the next conflict to satisfy is conflict #7, given below.

```
7.   ¬ [ SWITCH = CHARGER-1 ∧ CHARGER-2 = FULL-ON]
```

Also, since it is the first node of the search tree, all assignments are allowed under the paths of this node, except for assignments to the *switch*. The resultant expansion of the constituent diagnosis for this conflict is shown below:
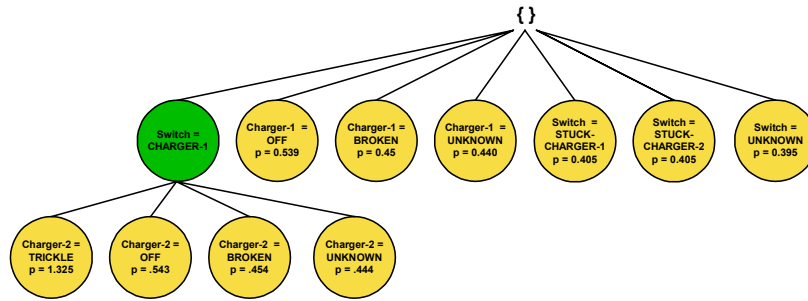
Figure 6-24 - Expansion of Constituent Diagnosis #7 for CDA*

The CDA* algorithm computes the costs associated with each node using the heuristic equations, which results in the best cost path being { *switch = charger-1, charger-2 = trickle* }. Upon going down this path, CDA* determines that it cannot satisfy the following conflict:

8. ¬ [SWITCH = CHARGER-1 ∧ CHARGER-2 = TRICKLE]

CDA* then chooses the next likely node in the search tree, which results in the path { *switch = charger-1, charger-2 = off* }. Additionally, when CDA* expanded the constituent diagnosis in Figure 6-24, the path *switch = charger-1* and *charger-2 = off* has a reduced set of reachable component modes due to the *charger-2 = trickle* sibling. CDA* determines that this path satisfies conflicts 7 and 8 using *charger-2 = off*, and conflicts 1 through 6 and 16 through 21 using *switch = charger-1*. CDA* then expands the constituent diagnosis related to conflict #9:

9. ¬ [BATTERY = FULL]

The resultant expansion of the related constituent diagnosis is shown below.
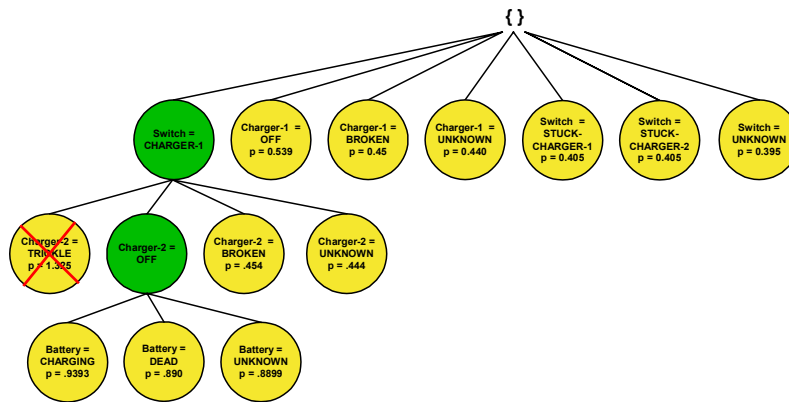
Figure 6-25 - CDA* Expansion of Conflict #9

The expansion shown above guides the CDA* search to follow the path { *switch= charger-1, charger-2 = off, battery = charging* } because the cost of this node is 0.9393, which is greater than the next highest cost node *charger-1 = off* with p = 0.539. CDA* determines that by adding the assignment *battery = charging* satisfies conflicts 9 through 13. The remaining conflicts to be satisfied from Appendix C are:

```
14. ¬ [ SWITCH = CHARGER-1 ∧ CHARGER-1 = TRICKLE ]
15. ¬ [ SWITCH = CHARGER-1 ∧ CHARGER-1 = OFF ]
```

CDA* expands conflict #14 under the best cost path { *switch = charger-1, charger-2 = off, battery = charging* } resulting in the following expansion in Figure 6-26. This expansion results in satisfying all conflicts by choosing the path { *switch – charger-1, charger-1 = full-on, charger-2 = off, battery = charging* }. However, the cost associated with this path is 0.0440. CDA* does not identify this as the highest cost node because the node *charger-1 = off* has cost of 0.539. CDA* would then expand constituent diagnoses under this node in the same process detailed here. The difference under this node is that the assignment *switch = charger-1* is

not allowed in any children of *charger-1 = off*, as depicted in Figure 6-22. The full example is given in Appendix C.
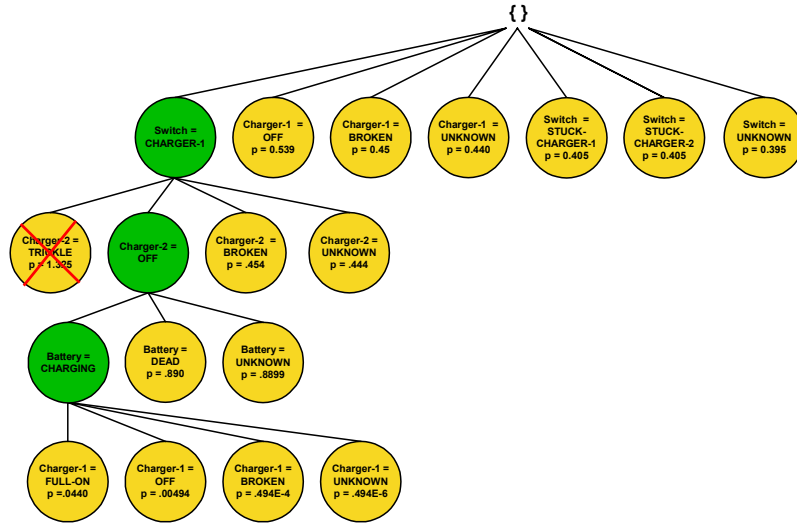
Figure 6-26 - Expansion of Constituent Diagnosis #14

Once the CDA* algorithm finds a node that is complete, it returns the node to the Generate algorithm. The final step of the Dynamic Mode Estimate Generation algorithm is to then call the Rank algorithm to determine the total probability of the state.

6.4.5   Rank Algorithm

The final step in determining the current belief state, $B^{(t+1)}$, is to rank each mode estimate. The Rank algorithm uses the current mode estimate generated from the Generate and CDA* algorithms, with the enabled transitions and previous belief state, $B^{(t)}$, to determine the posterior probability of the current mode estimate. Once the posterior probability has been calculated, the Rank algorithm places the current mode estimate in the

current belief state, $B^{(t+1)}$, in order of decreasing probability. The inputs and outputs of the algorithm are shown below.
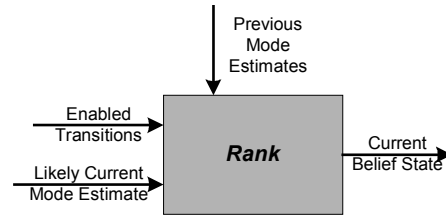
Figure 6-27 - Inputs and Outputs of the Rank Algorithm

The definitions of the inputs and outputs are as follows. The '*enabled transitions*' are the transitions from the Compiled Conflict Recognition whose source modes mentioned a component mode assignment in the previous mode estimates, and where all assignments in the guard were in the current commands and previous mode estimates. The '*previous mode estimates*' represent the approximate previous belief state, $B^{(t)}$, and map the previous set of states at time '*t*' to their respective probabilities. The '*likely current mode estimate*' is the mode estimate returned from the Generate and CDA* algorithms that is consistent with the current conflicts. Consistency of this mode estimate implies that the component mode assignments in the state of the mode estimate agree with the commands given and predict the observations made between time '*t*' and '*t+1*'. Finally, the current belief state, $B^{(t+1)}$, holds all mode estimates generated for time '*t+1*'.

6.4.5.1   Rank Algorithm Description

The Rank algorithm calculates the posterior probability of a mode estimate using mode estimates in the previous belief state that transition to the current mode estimate, $S_j^{(t+1)}$. This requires determining all transitions from the previous mode estimates to the current mode estimate.   The representation for this calculation is shown below. Noted on the figure is

the state and its associated probability, where $P(S_j^{(t+1)})$ is to be determined. The transition probabilities, P_T, are noted on the arcs between previous mode estimates and the current mode estimate.

The approach taken is to determine the enabled transitions that have in their targets, the component mode assignments in $S_j^{(t+1)}$, and then store the source component modes of these transitions. Using this list of source component modes, the Rank algorithm then iterates through the previous mode estimates in $B^{(t)}$, and determines if all component mode assignments in $S_i^{(t)}$ are in the list of source component mode assignments.
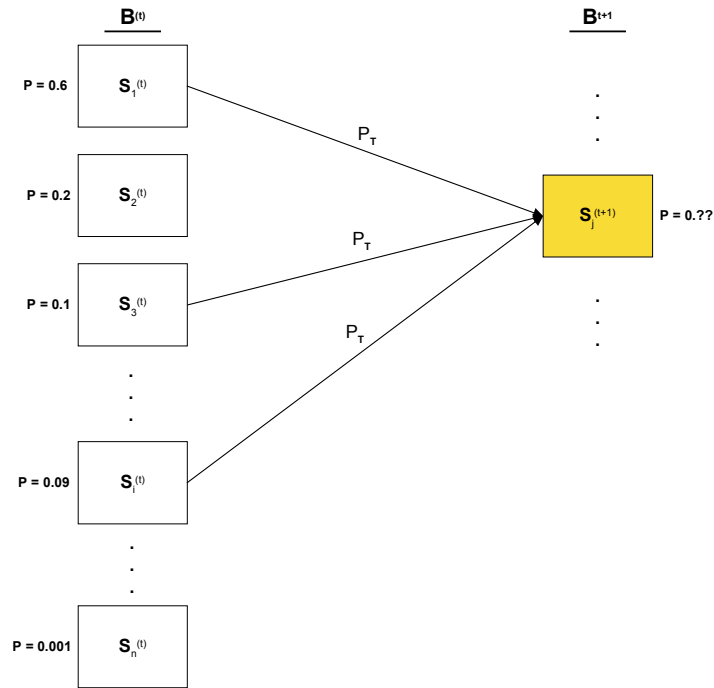
Figure 6-28 - Rank Algorithm Probability Calculation for a Mode Estimate

The Rank algorithm determines the transition, P_T, from a mode estimate, $S_i^{(t)}$, in the previous belief state to the current mode estimate, $S_j^{(t+1)}$. The determination of a transition between states is dependent on the individual

component transitions. The algorithm must then determine if the component mode assignments mentioned in state $S_i^{(t)}$ can transition to the component mode assignments mentioned in state $S_j^{(t+1)}$. This can be represented graphically as follows.
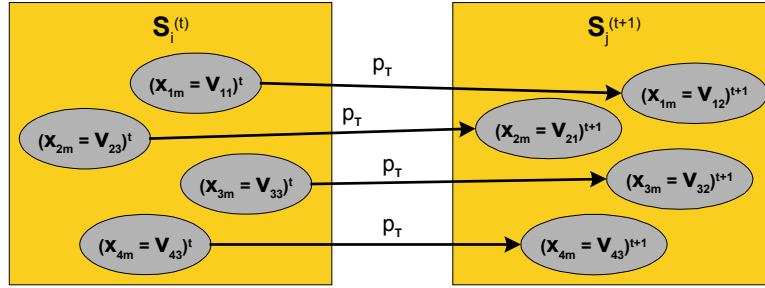
Figure 6-29 - Determination of Component Mode Assignment Transitions

The enabled transitions identified by the Transition Trigger are used to determine if the component mode assignments in state $S_i^{(t)}$ can transition to the component mode assignments in state $S_j^{(t+1)}$. The example in Figure 6-29 denotes the component transitions, but assuming that the transition from $(x_{3m} = v_{33})^t$ to $(x_{3m} = v_{32})^{t+1}$ has a probability, p_T, of zero, then the resulting transition probability from $S_i^{(t)}$ to $S_j^{(t+1)}$ is zero.

To determine these transition probabilities, the Rank algorithm identifies the enabled transitions that have in their targets the component mode assignments in $S_j^{(t+1)}$. The Rank algorithm then stores the component mode assignments that are in the source of the transitions in the list 'source-modes'. The transition probability p_T, is extracted and used in the determination of the overall transition probability, using the equation below

$$ P_{\mathbf{T}_{S_i^{(t)} \to S_j^{(t+1)}}} = \prod_{(x_{im}=v_{ij}) \in S_i^{(t)}} p_{\mathbf{T}\left(x_{im}=v_{ij}\right)^t \to \left(x_{im}=v_{ij}\right)^{t+1}} $$

Equation 6-9 - Probability Equation for Transitions Between States

This equation is the same used for mode estimation for CCAs, described in Chapter 2. To use this equation, the Rank algorithm must determine, for a

given $S_i^{(t)}$, if the component mode assignments are in the list of 'source-modes'. If all component mode assignments in $S_i^{(t)}$ are in the list 'source-modes', then P_T is non-zero, and can be calculated using Equation 6-9. This equation assumes that component mode transitions are independent of other transitions. This equation also assumes that the guards on these transitions are already satisfied. This then means that, from Equation 6-3, P_G is 1, for all transitions used by the Rank algorithm since they are 'enabled transitions'. Once the Rank algorithm has determined the transition from the previous mode estimate, it can then calculate the probability of the current mode estimate, $S_j^{(t+1)}$, given that source state, $S_i^{(t)}$ using:

$$P\left(S_j^{(t+1)} \mid T_{S_i^{(t)} \to S_j^{(t+1)}}\right) = P\left(S_i^{(t)}\right) \bullet P_{T_{S_i^{(t)} \to S_j^{(t+1)}}}$$

Equation 6-10 - Probability of a State Transition

This equation uses the probability of the previous mode estimate, $S_i^{(t)}$, and the transition probability determined by Equation 6-9. The final step in determining the total probability of the current mode estimate is to then sum all of the individual state probabilities from the previous belief state, $B^{(t)}$.

$$P\left(S_j^{(t+1)} \mid B^{(t)}\right) = \sum_{S_i^{(t)} \in B_t} P\left(S_i^{(t)}\right) \bullet P_{T_{S_i^{(t)} \to S_j^{(t+1)}}}$$

Equation 6-11 - Total Probability for a Mode Estimate

The equations given here describe the process of the Rank algorithm and the calculation of the total probability of a current mode estimate. This calculation is performed each time a consistent mode estimate is generated from the Generate and DDA* algorithms. The Generate algorithm then uses the total probability in its algorithm, as described in Section 6.4.3.

As described in the Generate algorithm, the generation of current mode estimates is an incremental process. As a result, the Rank algorithm must determine if a current mode estimate has already been generated and

ranked. This requires checking if the current mode estimate, $S_j^{(t+1)}$ is the same as any of the mode estimates that have been recorded in the current belief state, $B^{(t+1)}$. If the current mode estimate, $S_j^{(t+1)}$ is the same as a mode estimate already ranked in the current belief state then $B^{(t+1)}$ is not altered.

The steps described here are listed below, and a more thorough description is given in Chapter 7.

Rank($S_{j(t+1)}$, $B^{(t)}$, *Enabled Transitions*)

1. For each $S_k^{(t+1)}$ in $B^{(t+1)}$
   a. if $S_j^{(t+1)}$ is equal to $S_k^{(t+1)}$, then return $B^{(t+1)}$
2. For each $S_i^{(t)}$ in $B^{(t)}$
   a. Use Equation 6-9 and *Enabled Transitions* to calculate P_T
   b. Use Equation 6-10 to calculate $P(S_j^{(t+1)} \mid S_i^{(t)})$
   c. Use Equation 6-11 to keep calculate $P(S_j^{(t+1)} \mid B^{(t)})$
3. Insert $S_j^{(t+1)}$ in $B^{(t+1)}$ in order of decreasing probability

6.4.5.2   Rank Algorithm Example

The process of the Rank algorithm is best demonstrated using an example. Recall the example state transition system shown in Figure 6-11. Using this example and the steps of the Generate algorithm described in 6.4.3, the steps of the Rank algorithm are demonstrated as follows.

In step 2 of the example, the probability of the current mode estimate $\left\langle S_1^{(t+1)}, P\left(S_1^{(t+1)}\right) \right\rangle$ was determined to be 0.62. In the steps of the Generate and CDA* algorithms, only the probability of 0.5 was determined by using the previous mode estimate $\left\langle S_1^{(t+1)}, P\left(S_1^{(t+1)}\right)=0.5 \right\rangle$, and its most likely transition $P_T = 1.0$. The Rank algorithm updated the probability of the mode estimate by determining that the previous mode estimate

$\left\langle S_2^{(t+1)}, P\left(S_2^{(t+1)}\right) = 0.3 \right\rangle$ had a transition to $S_1^{(t+1)}$ with $P_T = 0.4$. This determination results in the following values.

$$P(S_1^{(t+1)} \mid S_1^{(t)}) = P(S_1^{(t)}) * P_T = 0.5 * 1.0 = 0.50$$
$$P(S_1^{(t+1)} \mid S_2^{(t)}) = P(S_2^{(t)}) * P_T = 0.3 * 0.4 = 0.12$$

The remaining previous mode estimates did not have any transitions to the current mode estimate, so the values of P_T for these were 0. The resultant total probability of the current mode estimate $S_1^{(t+1)}$ is then given by:

$$P(S_1^{(t+1)} \mid B^{(t)}) = P(S_1^{(t+1)} \mid S_1^{(t)}) + P(S_1^{(t+1)} \mid S_2^{(t)}) = 0.50 + 0.12 = 0.62.$$

This result is the same probability noted on Figure 6-11, and this example demonstrates how to arrive at that value.

Step 3 of the example in 6.4.3.2 demonstrates the need for the first steps of the Rank algorithm. In this step, the Generate algorithm has chosen $S_3^{(t)}$ as the source. This causes the DDA* algorithm to generate $S_3^{(t+1)}$ as the most likely mode estimate, but this mode estimate has already been generated by $S_2^{(t)}$. As a result, the Rank algorithm does not calculate the total probability of this mode estimate again. The Rank algorithm determined this by checking the mode estimate generated against the mode estimates already in the current belief state, which include $S_1^{(t+1)}$ and $S_3^{(t+1)}$. This determination then causes the Generate algorithm to proceed as described in the remainder of the example in 6.4.3.2.

6.4.5.3   Rank Algorithm and Belief Update

The equations used by the Rank algorithm are the same as those given in Chapter 2 for belief update of Hidden Markov Models. As such, the Rank algorithm enables Compiled Mode Estimation to perform full belief update. The equations describing belief update are repeated below.

$$\sigma^{(\bullet t+1)}\left[s_i\right] \equiv \sum_{j=1}^{n} \sigma^{(t\bullet)}\left[s_j\right] P_{\mathbf{T}}\left[s_j \mapsto s_i\right]$$

$$\sigma^{(t+1\bullet)}\left[s_i\right] \equiv \sigma^{(\bullet t+1)}\left[s_i\right] \frac{P_{\mathbf{O}}\left[s_i \mapsto o_k\right]}{\sum_{j=1}^{n} \sigma^{(\bullet t+1)}\left[s_i\right] P_{\mathbf{O}}\left[s_j \mapsto o_k\right]}$$

Equation 6-12 - Standard Belief Update Equations for Hidden Markov Models

To use the standard belief update equations, a transition function, $P_{\mathbf{T}}$, and an observation function, $P_{\mathbf{O}}$ must be defined for Compiled Mode Estimation. The transition function, $P_{\mathbf{T}}[s_j \rightarrow s_i]$, is described by the enabled transitions, with the probability of transitions between mode estimates defined in Equation 6-9. The Rank algorithm uses Equation 6-9 to calculate the transitions between previous and current mode estimates. The right hand side of the first belief update equation is then the same as Equation 6-10, with the posterior probability of a previous mode estimate, $\sigma^{(t\bullet)}[s_j]$, represented by $P(S_i^{(t)})$. The full apriori probability of a current mode estimate, $\sigma^{(\bullet t+1)}[s_i]$ is the same as the expression of Equation 6-11.

The observation function, $P_{\mathbf{O}}$, for each mode estimate generated is automatically 1. In using the dissents and compiled transitions, the mode estimates generated from the Dynamic Mode Estimate Generation are guaranteed to be consistent with the observations. This is guaranteed because the compilation process is complete and generates all conflicts and removes the need for any satisfiability of the system model and transitions. Recall from the definition of the observation function for CCA that the observation function value would change only if a mode estimate would not predict an observation. In these cases the $P_{\mathbf{O}}$ would be either 0 or $1/n$, where n represented the number of possible assignments to a particular observation value. However, by using the dissents, which represent the

compilation of the observation function, the mode estimates generated are guaranteed to be consistent with the observations.

The final piece missing from the Rank algorithm is the normalization performed by the second belief update equation in Equation 6-12. This step is performed once the current belief state has been completely generated, but is performed at the top level of the Online Mode Estimation algorithm.

6.5 Mapping Compiled Mode Estimation to ME-CCA

The steps of CME can be related to the mode estimation algorithm for Concurrent Constraint Automata presented in Chapter 2, ME-CCA. The steps of CME are slightly different because of the model used and the compilation process. The following comparison first describes the step of CME, followed by the corresponding step in ME-CCA.

Step 1:   CME
    NONE
Step 1:   ME-CCA
    Extracts constraints, $C_{Mi}^{(t)}$ from the previous mode estimates, $B^{(t)}$

The constraint extraction by ME-CCA performed in Step 1 is done so that these constraints can be used in determining the set of transitions that are enabled given those previous constraints. The allowable transitions are determined in Step 2 of the ME-CCA algorithm. In the case of CME, this is not necessary because of the dissents and compiled transitions. All that is needed is the previous mode estimates, not their constraints, and the commands to determine if a transition is enabled.

Step 2:   CME
    Calculates the set of reachable current modes using the previous mode estimates, $B^{(t)}$ and
    the control variables, $\mu^{(t)}$ to first determine the enabled transitions, $\mathbf{T}_{EN}$. The reachable

current modes are then the targets of $\mathbf{T}_{EN}$

    Step 2:   ME-CCA

    Calculates all reachable current mode estimates, $\langle S_j^{(t+1)}, p_{ij}\rangle$ using the previous mode estimates, $S_i^{(t)}$, constraints $C_{Mi}^{(t)}$ and the control variables, $\mu^{(t)}$

CME determines the set of reachable component modes from set of enabled transitions without performing any satisfiability determination. ME-CCA however requires satisfiability to check transition guards in order to generate all reachable current mode estimates. CME has removed the need for satisfiability by compiling the transitions through the process described in Section 5.6. CME does not determine all reachable current mode estimates, but instead maintains the representation of the individual component modes.

    Step 3: CME

        NONE

    Step 3 :   ME-CCA

        Calculates the apriori probability of each current mode estimate $\langle S_j^{(t+1)}, p_j\rangle$ using the standard belief update equation, $p_j = \Sigma \, \sigma^{(t^*)}[S_i^{(t)}] * p_{ij}$

CME does not calculate the posterior probability of mode estimates at this time since the current mode estimates have not been determined. ME-CCA performed this calculation because it has determined all reachable current mode estimates.

    Step 4:   CME

    Determines the current constraints, represented by the set of enabled dissents, $\mathbf{D}_{EN}$, using the current observations, $O^{(t+1)}$

    Step 4:   ME-CCA

        Extracts the current constraints, $C_{Mi}^{(t+1)}$ from the set of reachable current mode estimates,

            $\cup \, \langle S_j^{(t+1)}, p_j\rangle$

The CME step requires triggering the dissents to determine the enabled dissents through the triggering process described in Section 6.3.1. These dissents represent the constraints on the current mode estimates because

these conflicts must be resolved by the current mode estimate for it to agree with the observations. The constraints that the ME-CCA algorithm extracts are the mode constraints associated with the reachable current mode estimates determined in Step 2.

Step 5:   CME
Generates a mode estimate, $\langle S_j^{(t+1)}, p_j \rangle$ that resolves all conflicts of $\mathbf{D}_{Enabled}$ and is automatically consistent with the observations, $O^{(t+1)}$ using the Generate and DDA* algorithms described in Sections 6.4.3 and 6.4.4, respectively.

Step 5:   ME-CCA
Determines if the mode estimate, $\langle S_j^{(t+1)}, p_j \rangle$ is consistent with the constraints
$C_{Mi}^{(t+1)}$ and the current observations, $O^{(t+1)}$.

The overall goal of this step of CME and ME-CCA is the same, however the approach is very different. CME does not require satisfiability to determine consistent mode estimates. All that is required is for the mode estimate to resolve the conflicts. The conflicts generated through compilation are enough to reconstruct the diagnosis of the system online, removing the need for a satisfiability check of the mode estimates. ME-CCA however does require satisfiability to determine if a reachable current mode estimate is consistent with the observations and constraints of the system model. Additionally, CME incrementally generates current mode estimates, while ME-CCA determines if all reachable current mode estimates are consistent. This means that there are some reachable current mode estimates that are inconsistent. The time taken to test these is a point of wasted effort by ME-CCA. This step demonstrates the computational savings of CME because of the removal of the NP-hard problem of satisfiability.

Step 6:   CME
Calculates the apriori and posterior probabilities of a consistent current mode estimate,

$\langle S_j^{(t+1)}, p_j \rangle$ generated from Step 5 using the standard belief update equations with P_T

and P_O as defined in Section 6.4.5. Step 5 and Step 6 of CME are performed iteratively until the current belief state, $B^{(t+1)}$, is complete.

Step 6:   ME-CCA

Calculates the posterior probability by summing like states $S_i^{(t)}$ to $S_r^{(t)}$ and applying the

observation function values determined in Step 5 of ME-CCA.

The final step of the CME algorithm is to determine the posterior probability of a mode estimate generated by the Generate and CDA* algorithms using the Rank algorithm. The Rank algorithm determines all possible transitions from the previous belief state to a current mode estimate. ME-CCA calculates the posterior probability using the apriori probability calculated in Step 3.

By mapping the steps of CME to the ME-CCA algorithm, this highlights the major benefit of the computations of CME. Since the online algorithms are enabled by the compiled model, many of the computations that were necessary in ME-CCA are now removed from the algorithms of CME. The computational savings for CME are explicated when comparing the generation of consistent mode estimates to ME-CCA in step 5. ME-CCA tests consistency of many mode estimates, whereas CME only generates consistent mode estimates.

This chapter concludes the presentation of CME, with the implementation details in Chapter 7. To this point, the process of compilation has been developed that maps the system model to a CMPCA, which is a compact encoding of the system model as dissents and compiled transitions. The dissents are generated through the Enumeration algorithm given in Section 3.3. The process for generating compiled transitions was described in Section 5.6. This chapter first developed Compiled Conflict Recognition in Section 6.3 to determine the dissents and compiled transitions that pertain

to the current observations and commands. This process uses standard rule-triggering methods tailored to the dissents and compiled transitions. The second phase of Online Mode Estimation, Dynamic Mode Estimate Generation was developed in Section 6.4. This portion determines the likely transitions from previous mode estimates to current mode estimates, using the conflicts to guide the choice of component mode assignments in a conflict-directed A* search.

The CME engine and the Online Mode Estimation engine have been designed with several key attributes. The engine is capable of reconstructing mode estimates from conflicts in real-time using the Online-ME algorithms. CME reduces memory utilization through the compact encoding of the model constraints as dissents and compiled transitions. Additionally, the dissents express the diagnostic rules of the system model, encoded as "observations imply conflict". These enable inspection of the mode estimates for correctness by a human. Finally, CME is capable of using multiple sources of information to determine the most likely mode estimates, and track these mode estimates over time to diagnose complex system failures.

This chapter presented the underlying ideas of the CME engine. What remains is to present the algorithms of CME in Chapter 7, and validate these algorithms through experimentation in Chapter 8. Chapter 9 draws conclusions from the validation, and is followed by Future Work in Chapter 10.

7　Compiled Mode Estimation Algorithms

Chapter 6 presented the ideas and innovations for performing mode estimation using a compiled model. The online mode estimation algorithms have been described to convey the key ideas behind each algorithm. This chapter details the algorithms for all portions of Online Mode Estimation, and gives specifics for implementation. The methods described in this chapter have been used to generate results for the validation experiments described in Chapter 6.

The chapter begins with the detail of the Compiled Conflict Recognition algorithms in section 0, followed by the Dynamic Mode Estimate Generation algorithms in section 7.2. The chapter ends with a description of the top level Online Mode Estimation algorithm, which enables the Compiled Conflict Recognition and the Dynamic Mode Estimate Generation algorithms to work together to produce mode estimates.

7.1 Compiled Conflict Recognition

The Compiled Conflict Recognition algorithm maps the compiled model in the form of dissents and compiled transitions to a set of Constituent Diagnoses, Reachable Component Modes, and Enabled Transitions. These outputs are generated through the three top-level algorithms described in Chapter 3, the Dissent Trigger, the Transition Trigger and the Constituent

194　　Achieving Real-time Mode Estimation through Offline Compilation

Diagnosis Generator. This section only presents the Constituent Diagnosis Generator for brevity. The algorithm for the Dissent and Transition Triggers are presented in Appendix D.

7.1.1   Constituent Diagnosis Generator

The Constituent Diagnosis Generator uses the Enabled Dissents and Enabled Transitions to determine the set of Constituent Diagnoses and Reachable Current Modes. In addition, it passes on the Enabled Transitions. There are two distinct tasks within the Constituent Diagnosis Generator that produce the desired outputs. The first is to use the enabled transitions to determine the set of reachable current modes as described in Chapter 6. The second task is to map the enabled dissents to the set of constituent diagnoses, also described in Chapter 6. The inputs and outputs of the Constituent Diagnosis Generator are shown below.
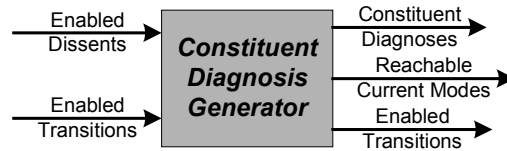
Figure 7-1 - Inputs and Outputs of Conflict Generator

A reachable current mode in the set of reachable current modes, $\Pi_m^{Current}$, stores:

1. Transition probabilities for a given Reachable current mode
2. List of previous mode estimates for a given Reachable current mode
3. $x_{im} = v_{ij}$ identifying this reachable current mode

A particular component mode may be the target of more than one transition, depicted in Figure 7-2.
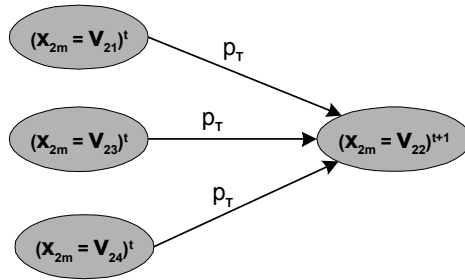
Figure 7-2 - A Reachable Current Assignment with Multiple Previous Sources

If a component mode is reachable from multiple previous mode estimates, then the probability of the component mode changes with respect to the previous mode estimate at time 't'. The component mode assignment, $(x_{2m} = v_{22})$, stores the previous mode estimates that mention $(x_{2m} = v_{21})$, $(x_{2m} = v_{23})$ and $(x_{2m} = v_{24})$, as well as the individual transition probabilities, p_T for each transition, giving the transition probability distribution. Storing this information is enabled by the previous list of component modes determined by the Compress-Mode-Estimates algorithm, given in Appendix D. All that is required is to go through the list of enabled transitions, and access the stored component modes in the source and the transition probabilities.

This computation results from the need to track the previous belief state, not just a single previous mode estimate. A reachable component mode stores the transition probability distribution and the previous mode estimates that are the sources of these transitions. This is used to simplify the calculations of the Dynamic Mode Estimate Generation algorithm.

The second step of the Constituent Diagnosis Generator algorithm transforms the Enabled Dissents into Constituent Diagnoses. This transformation uses the set of all component mode assignments, Π_m in the approximate belief state and the dissents to determine the set of constituent diagnoses for the conflict in each Enabled Dissent. The conflicts restrict the

component modes by specifying infeasible combinations. The algorithm then looks for all assignments of a particular component variable not in the conflict, and places these in the set of constituent diagnoses of the conflict. The set of constituent diagnoses corresponds one to one with the set of enabled dissents. The resultant Constituent Diagnosis Generator algorithm that captures these computations is given below.

---

function Constituent-Diagnosis-Generator(DS_{EN}, T_{EN}, Π_m)
    returns Reachable current modes, $\Pi_m^{Current}$, Constituent diagnoses, CD, and
enabled transitions, T_{EN}
        for each T_i in T_{EN}
            for $(x_{im} = v_{ij})$ in destination mode of T_i
                transition probability $\leftarrow$ P(T_i) for $(x_{im} = v_{ij})$
                mode estimate $\leftarrow$ *mode estimate* from source $(x_{im} = v_{ij})$ of T_i
                unless $(x_{im} = v_{ij}) \notin \Pi_m^{Current}$
                    $\Pi_m^{Current} \leftarrow (x_{im} = v_{ij}) \cup \Pi_m^{Current}$
        end

        for each d_i in DS_{EN}
            for each $(x_{im} = v_{ij})$ in d_i
                $cd \leftarrow cd \vee (x_{im} = v_{im}) \ \forall \ v_{im} \neq v_{ij} \in \mathbf{D}(x_{im})$
            end
            $CD \leftarrow cd \cup CD$
        end

        return $\Pi_m^{Current}$, CD, and T_{EN}

Figure 7-3 – Constituent Diagnosis Generator Algorithm

The algorithms given here for the Compiled Conflict Recognition map the compiled model, the current observations and commands to the Constituent Diagnoses, the Reachable Current Modes, and the Enabled Transitions. This information, along with the constituent diagnoses and enabled transitions, guide the search that produces the current mode estimates.

7.2   Dynamic Mode Estimate Generation

The presentation of the Online Mode Estimation algorithms now focuses on the algorithms in the Dynamic Mode Estimate Generation process that uses the information of the Compiled Conflict Recognition. The Dynamic Mode Estimate Generation process takes the constituent diagnoses, the reachable current modes and the enabled transitions and determines the current mode estimates of the system that are consistent with the observations and commands.

The Dynamic Mode Estimate Generation process is broken into three distinct functions: Generate, Conflict Directed A* and Rank algorithms. The Generate algorithm maps the reachable current modes to a reduced set, called the reachable component modes', which enables the Conflict Directed A* (CDA*) algorithm to search for the most likely mode estimate that satisfies the constituent diagnoses. The Rank algorithm then determines the probability of this mode estimate using the enabled transitions, and ranks it in the current belief state. This section details each of these algorithms and any supporting algorithms, beginning with the Generate algorithm, then specifying the CDA* algorithm and ending with the Rank algorithm.

7.2.1 Generate

The Generate algorithm performs several tasks to enable the Dynamic Mode Estimate Generation algorithm. Its main task, as described in Chapter 3, is to choose a previous mode estimate that reduces the set of reachable current modes, to the set of reachable component modes'. Each previous mode estimate has a corresponding *reachable component modes'* that is computed per Figure 3-14. The other important task of the Generate algorithm, is to enable the communication of the Conflict Directed A* algorithm and the Rank algorithm. This communication path sends the

'likely current mode estimate' from the CDA* algorithm to the Rank algorithm. This is passed through the Generate algorithm because it too must know and use the current mode estimates in its main task of choosing a previous mode estimate.

The inputs and outputs for the Generate algorithm are shown below.
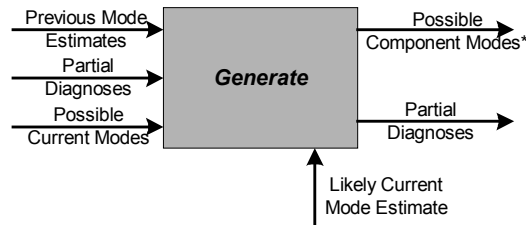
Figure 7-4 - Inputs and Outputs of the Generate Algorithm

The creation of the reduced set of component modes becomes a simple task using the stored information in each component mode assignment. Recall that the previous mode estimate and associated transition probability are stored in a reachable current mode. All that is required is to search the full list of reachable current modes for ones that are reachable from the chosen previous mode estimate, which is specified by the transition contained in the reachable current mode. This computation only has to be done once for a previous mode estimate and then recalled when the previous mode estimate is chosen again.

The exploration of the tree for the Generate algorithm is driven by a queue. This queue is comprised of nodes of the tree. Recall from Chapter 6, that the nodes of the tree represent the current consistent mode estimates generated from the CDA* algorithm. Also, only one node is maintained under each previous mode estimate, representing the most recently generated current mode estimate. The information contained in each node

is then the cost of the node, the current mode estimate, $S_j^{(t+1)}$, and the previous mode estimate, $S_i^{(t)}$. The cost of a node that has not been Ranked is given by $f(n) = g(n) + h(n)$:

$$g(node) = P\left(S_i^{(t)}\right) \bullet P_{T_{S_i^{(t)} \to S_j^{(t+1)}}}$$

$$h(node) = Residual$$

<div align="center">Equation 7-1 - Heuristics for the Generate Tree Search</div>

If a current mode estimate has been ranked, then the posterior probability is known. The Generate algorithm uses this probability as the cost for the previous mode estimate. Generate chooses a node in the search tree that has a high cost, representative of the probability that the previous mode estimate will transition to current mode estimates.   This cost and maintaining the proper ordering of the queue enable the Generate algorithm to properly explore the search tree and choose the appropriate previous mode estimate.

The computations described here and in Chapter 6 are captured in the Generate algorithm below.   The set of *reachable component modes'* is denoted by $\Pi_m^{RCM'}$.

---

function Generate($B^{(t)}$, $\Pi_m^{Current}$, *CD*, T_{EN})
    returns a likely current mode estimate $S_j^{(t+1)}$, or the current belief state, $B^{(t+1)}$
when exiting
    for each $S_i^{(t)}$ in $B^{(t)}$
        Nodes $\leftarrow$ *Nodes* $\cup$ $S_i^{(t)}$, with a cost of 1, ordered by $P(S_i^{(t)})$
    end
    Residual $\leftarrow$ 1
    loop do
        if *Nodes* is empty
            then exit
            else *node* $\leftarrow$ Remove-Best(*Nodes*)
        for $S_i^{(t)}$ in *node*
            if *previous CDA** output is empty
                then for each $(x_{im} = v_{ij})$ in $\Pi_m^{Current}$
                      for each $S_i^{(t)}$ in *mode estimate* of $(x_{im} = v_{ij})$
                          if $S_i^{(t)} = S_i^{(t)}$

$$\text{then } \Pi_m{}^{RCM'} \leftarrow (x_{im} = v_{ij}) \cup \Pi_m{}^{PCM'}$$

$$\textbf{end}$$
$$\textbf{end}$$
$$S_j^{(t+1)} \leftarrow \text{CDA*}(\Pi_m{}^{RCM'}, CD)$$
$$\textbf{else}$$
$$S_j^{(t+1)} \leftarrow \text{CDA*}(\Pi_m{}^{RCM'}, CD)$$

if $S_j^{(t+1)}$ is empty
    then remove $S_i^{(t)}$ from *Nodes*
else
    $\langle S_j^{(t+1)}, P(S_j^{(t+1)}) \rangle \leftarrow \text{Rank}(S_j^{(t+1)}, B^{(t)}, T_{EN})$
    Residual $\leftarrow$ *Residual* $- P(S_j^{(t+1)})$
    node-cost $\leftarrow P(S_j^{(t+1)})$
    for each *node* in *Nodes*
        node-cost $\leftarrow P(S_k^{(t+1)}) + $ *Residual*
    end
    Nodes $\leftarrow$ InsertInOrder(*node*, *Nodes*)

    highest probability $\leftarrow$ max($P(S_j^{(t+1)} \in Nodes)$)
    lowest probability $\leftarrow$ min($P(S_j^{(t+1)} \in Nodes)$)
    number current mode estimates $\leftarrow$ *number current mode estimates* + 1

  while($\neg$ *halting conditions*)
  return $B^{(t+1)} \leftarrow$ *Rank*

halting conditions $\equiv$ [*total probability* $\geq$ set value and
        comp-time $\geq$ set time   and
        highest probability $\geq$ Residual and
        lowest probability $\geq$ Residual and

        number next states $\geq$ factor * N_{poss}

    ]

Figure 7-5 - Generate Algorithm for Dynamic Mode Estimate Generation

The algorithm first sets up the queue by creating '*nodes*' that hold a previous mode estimate and an initial cost of 1. This is done to force the algorithm to generate a current mode estimate from each previous mode estimate. The next step initializes the Residual to 1, followed by the loop that executes the generation of mode estimates. The first step in this loop is to remove the best node from the top of the '*Nodes*' queue. Once extracted, the *node* is tested to determine if it has a child branch. If it does, then there is no need to generate the reduced set of component modes, $\Pi_m{}^{RCM'}$. If it

does not, then the algorithm proceeds to create this list by iterating through the full list of reachable current modes, $\Pi_m{}^{Current}$, and extracting those that are from the desired previous mode estimate. An example of this computation was given in 6.4.3. The algorithm then uses the list $\Pi_m{}^{RCM'}$ and the constituent diagnoses to generate a new current mode estimate. If there is no current mode estimate returned, then the algorithm removes the previous mode estimate from the queue so it is never used again.

Once a current mode estimate has been generated, the total probability must be updated, performed by the Rank algorithm. The Generate algorithm then uses this total probability, $P(S_j^{(t+1)})$, to update the residual value according to Equation 7-1. Using this updated residual, the algorithm then calculates the new cost for each node. Since the residual only decreases as more nodes are added, there is no need to reorder the queue. A better approach to calculating costs on *nodes* is given in Future Work (Chapter 9). Instead all that remains is to insert this new node into the queue in the appropriate order. This is done by making a call to the 'Insert-In-Order' function, detailed in Appendix D. Once the new node has been inserted in the queue, the loop can restart or terminate, if necessary.

The final step in this process is to test the halting conditions of the loop. The halting conditions shown above represent three different types of halting conditions, hard, soft and items that will always cause a halt. An example of the last type of halting condition is encoded in the algorithm itself. When there are no more items in the queue, representing the fact that there are no more mode estimates to generate, then the algorithm exits. An example of a hard halt is when the lowest probability of a mode estimate is greater than the residual. Using this halting condition gives the guarantee that the most likely 'N' mode estimates have been generated. Another hard halt is encoded to stop the task of Online Mode Estimation if the process is

taking too long to determine an estimate of the system behavior. The final condition, a soft condition, halts the mode estimate generation when a certain space of the consistent current mode estimates has been explored. This is represented as a factor multiplied by the number of Reachable current states, 'N_{poss}'. This condition is used to stop the search from going unnecessarily long should the total probability not reduce significantly with each newly generated current mode estimate. Once the algorithm exits, this forces the end of the Dynamic Mode Estimate Generation algorithm, and the current belief state is returned.

7.2.2 Conflict Directed A*

Called within the Generate algorithm is the Conflict-Directed A* that performs the search for an optimal mode estimate that satisfies the constituent diagnoses. This search is framed as an A* search as described in Chapter 6. The CDA* algorithm uses the constituent diagnoses to guide the search, and the probabilities of the reachable component modes to calculate the heuristic for the cost used in the search. This cost is given in Equation 4-9, and utilizes the transition and component mode probabilities. The inputs and outputs of the CDA* algorithm are shown in Figure 7-6. This section presents the detail of the CDA* algorithm and any supporting functions required.
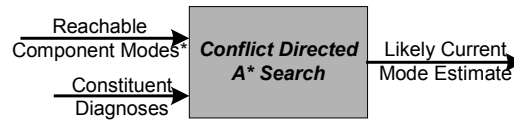
Figure 7-6 - Inputs and Outputs of the DDA* Algorithm

The CDA* algorithm used here is the Conflict Directed A* algorithm [*Williams, 2002*] with systematic search [*Ragno, 2002*] to guide the

expansion of nodes in the search tree using the constituent diagnoses. Guaranteeing systematicity requires storing the following for each node in the search tree.

1. All component mode assignments on the path from the root to the node
2. A list of allowable assignments
3. A list of unsatisfied constituent diagnoses
4. Cost of the node

Each time a new node is added to the search tree the fields stored in each node are updated as follows:

1. Add the new component mode to the previous list of modes
2. Using a 'do-not-use' list of component mode assignments, update the list of allowable component mode assignments
3. Determine the constituent diagnoses that the new component mode assignment satisfies and remove them from the list of unsatisfied constituent diagnoses
4. Update the cost of the node with the new component mode

The following is the CDA* algorithm and initialization algorithm. The initialization algorithm creates a single node in the tree that holds the set of reachable component modes*, $\Pi_m{}^{RCM'}$, and the Constituent diagnoses, *CD*. The set of reachable component modes is transformed into a list ordered by component mode variable, where the different component modes are sequentially ordered. This new list is noted as $\Pi_m{}^{VCM'}$, noting the *variable component modes'* list. The CDA* algorithm uses this node to expand the first constituent diagnosis, making a call to a supporting function, 'Expand-and-Insert'. The CDA* algorithm continues to expand nodes until the queue, *Nodes*, is empty or a node has generated a consistent mode estimate.

---

function Initialize-CDA*($\Pi_m^{\text{RCM'}}$, *CD*)

    returns initialized queue, *Nodes* that holds the transformed list, $\Pi_m^{\text{VCM'}}$

    for each ($x_{\text{im}} = v_{\text{ij}}$) in $\Pi_m^{\text{RCM'}}$

        $\Pi_m^{\text{VCM*}} \leftarrow \Pi_m^{\text{VCM*}} \cup (x_{\text{im}} = v_{\text{ij}})$, where $\Pi_m^{\text{VCM'}}$ is ordered by x_{im}

    end

    Nodes $\leftarrow$ $\Pi_m^{\text{VCM'}}$, *CD*

    return *Nodes*

function CDA*(*Nodes*, $\Pi_m^{\text{RCM'}}$, *CD*)

    returns current consistent mode estimate, $\langle S_j^{(t+1)}, P(S_j^{(t+1)})\rangle$

    if *Nodes* is empty

        then *Nodes* $\leftarrow$ Initialize-CDA*($\Pi_m^{\text{RCM'}}$, *CD*)

        for *node* in *Nodes*

          Nodes $\leftarrow$ ExpandAndInsert(*node, Nodes*)

    loop do

        node $\leftarrow$ Remove-Best(*Nodes*)

        if Node-Complete(*node*) is successful

          then return $\langle S_j^{(t+1)}, P(S_j^{(t+1)})\rangle$ in *node*, and *Nodes*

          else

            Nodes $\leftarrow$ ExpandAndInsert(*node, Nodes*)

    while *Nodes* is not empty

    return an empty $\langle S_j^{(t+1)}, P(S_j^{(t+1)})\rangle$ and *Nodes*

Figure 7-7 - Conflict Directed A* Algorithm

The CDA* algorithm detailed above gave the top-level description. First, the CDA* algorithm always returns not only the current mode estimate but also the queue remaining in the search tree. This enables the Generate algorithm to use the same previous mode estimate without having the CDA* algorithm regenerate the search tree. Second, CDA* uses several supporting algorithms, the 'Remove-Best', 'Node-Complete' and the 'Expand-and-Insert' algorithms. The 'Remove-Best' simply removes the node at the top of the queue, which represents the best cost node. The 'Node-Complete' algorithm determines if the mode estimate in the node contains a state, and that this state satisfies all constituent diagnoses. The 'Expand-and-Insert' algorithm performs the computations listed previously.

The next step is to detail the 'Expand-and-Insert' algorithm. This algorithm expands a constituent diagnosis and updates all of the fields within a node, as specified in the list of required computations above. This algorithm returns the updated queue to the CDA* algorithm.

---

function Expand-And-Insert(*node*, *Nodes*)
    returns an updated queue, *Nodes*
    if $\langle S_j^{(t+1)}, P(S_j^{(t+1)}) \rangle$ in *node* is empty
        then $cd_i \leftarrow$ first Constituent diagnosis in *CD*, stored in *node*

        else $cd_i \leftarrow$ ConstituentDiagnosis-To-Expand(*node*)
    if cd_i is empty
        then for a x_{im} in $\Pi_m^{VCM'}$ in *node* that has not been assigned
            for each $(x_{im} = v_{ij})$ that is allowed for x_{im}
                new node $\leftarrow$ copyNode(*node*)
                new node $\leftarrow$ update-Π_m^{VCM*} (*new node, do-not-use*)
                new node $\leftarrow$ add-Variable-Assignment(*new node*, $(x_{im} = v_{ij})$)
                if *new node* creation failed
                    then move to next $(x_{im} = v_{ij})$ in x_{im}
                  else
                        do-not-use $\leftarrow$ $(x_{im} = v_{ij}) \cup$ *do-not-use*
                        Nodes $\leftarrow$ insertNode(*new node, Nodes*)
            end
            return *Nodes*

    for each $(x_{im} = v_{ij})$ in cd_i
        new node $\leftarrow$ copyNode(*node*)
        new node $\leftarrow$ add-ConstituentDiagnosis-Assignment(*new node*, $(x_{im} = v_{ij})$)
        if *new node* failed to be created
            then move to next $(x_{im} = v_{ij})$ in cd_i
            else
                new node $\leftarrow$ update-$\Pi_m^{VCM'}$ (*new node, do-not-use*)
                do-not-use $\leftarrow$ $(x_{im} = v_{ij}) \cup$ *do-not-use*
                Nodes $\leftarrow$ insertNode(*new node, Nodes*)
    end
    return *Nodes*

Figure 7-8 - Expand and Insert Algorithm Supporting the CDA* Algorithm

The 'Expand-and-Insert' function performs the task of expanding a constituent diagnosis, or if all constituent diagnoses are satisfied, then

expands using an unassigned variable. The first step of the algorithm is to determine a Constituent diagnosis to expand. If the *node* does not have a mode estimate, then the algorithm chooses the first constituent diagnosis in the list, *CD*. Otherwise, the algorithm uses 'ConstituentDiagnosis-to-Expand' to determine the best constituent diagnosis to expand.

Once a constituent diagnosis has been chosen, then each component mode is expanded to new nodes. The 'Expand-and-Insert' algorithm makes use of several functions to enable this expansion. First, the algorithm copies the node before it adds a component mode, because there are normally more than one component modes mentioned in a constituent diagnosis. Once copied, the algorithm attempts to add the component mode by calling the 'add-ConstituentDiagnosis-Assignment'. This algorithm performs the computations associated with step 3 and step 4 specified above. The next task is to update the allowable list of component modes by removing any component modes previously expanded from its list of reachable component modes, $\Pi_m^{VCM'}$. This task is performed by the 'update-$\Pi_m^{VCM'}$' algorithm.

If there is not a constituent diagnosis to expand, then the algorithm expands any component variable that has not been assigned a value. The expansion places new nodes corresponding to the allowable component modes in the $\Pi_m^{VCM'}$ list for the chosen component variable. The algorithm that performs the addition of a component mode under these conditions is the 'add-Variable-Assignment' algorithm. Under this path, the '*do-not-use*' list is also used, but is computed in the same manner as when constituent diagnoses are expanded. The final task is to insert the *node* in order of decreasing probability into the queue, *Nodes*. The following details the 'add-ConstituentDiagnosis-Assignment' and the 'add-Variable-Assignment'.

function add-ConstituentDiagnosis-Assignment($node$, ($x_{im} = v_{ij}$))
    returns *node* with ($x_{im} = v_{ij}$) added if possible
    if ($x_{im} = v_{ij}$) $\notin \Pi_m^{VCM'}$ of *node*
        then mark *node* as a dead end
        return *node*
    if x_{im} is already assigned in $S_j^{(t+1)}$ of *node*
        then mark *node* as a dead end
        return *node*
    for each cd_i in CD_{Unsat} of *node*
        if ($x_{im} = v_{ij}$) $\in cd_i$
            then remove cd_i from CD_{Unsat} of *node*
        if $x_{im} \in cd_i$ & cd_i not removed
            then decrement the *counter* of usable assignments in cd_i

        if cd_i has only 1 variable remaining to be assigned
            then *next constituent diagnosis* $\leftarrow cd_i$ of the *node*
        if cd_i has no variables remaining to be assigned
            then mark *node* as a dead end
         end
    if *node* not marked as a dead end
        then $S_j^{(t+1)} \leftarrow S_j^{(t+1)} \cup (x_{im} = v_{ij})$ of *node*
            node-cost $\leftarrow P(S_j^{(t+1)}) \bullet P_T(x_{im} = v_{ij}) + \Pi \max[\ P_T(x_{im} = v_{ij})\ ]\ \forall\ x_{im}$ $\notin S_j^{(t+1)}$
    if *next constituent diagnosis* has not been updated
        next constituent diagnosis $\leftarrow$ first Constituent diagnosis in CD_{Unsat}
    return *node*

function add-Variable-Assignment($node$, ($x_{im} = v_{ij}$))
    returns *node* with ($x_{im} = v_{ij}$) added if possible
    if ($x_{im} = v_{ij}$) $\notin \Pi_m^{VCM'}$
        then mark *node* as a dead end
        return *node*
    if x_{im} is already assigned in $S_j^{(t+1)}$ of *node*
        then mark *node* as a dead end
        return *node*
    $S_j^{(t+1)} \leftarrow S_j^{(t+1)} \cup (x_{im} = v_{ij})$ of *node*
    node-cost $\leftarrow P(S_j^{(t+1)}) \bullet P_T(x_{im} = v_{ij}) + \Pi \max[\ P_T(x_{im} = v_{ij})\ ]\ \forall\ x_{im} \notin$ $S_j^{(t+1)}$
    return *node*

Figure 7-9 - Add Constituent Diagnosis and Add Variable Algorithms

The 'add-ConstituentDiagnosis-assignment' and the 'add-Variable-assignment' algorithms perform key operations enabling the CDA*

algorithm. These include early detection of dead ends in the search tree, adding a component mode assignment to the mode estimate of the node, and updating the cost of the node. The 'add-ConstituentDiagnosis-assignment' performs an essential operation of determining other constituent diagnoses that are satisfied by adding this assignment to the mode estimate.

The 'add-ConstituentDiagnosis-assignment' algorithm begins by performing several operations to determine if, by making the assignment, that the resultant mode estimate is a dead end. The first is to determine if the particular assignment is even in the allowable list of component modes, $\Pi_m^{VCM'}$. If it is, then the algorithm proceeds to check if the component mode variable, x_{im}, is already assigned a value in the mode estimate of the *node*. If it is not, then the algorithm proceeds to check the constituent diagnoses for a dead end. The algorithm uses the component mode assignment, $(x_{im} = v_{ij})$, to determine which constituent diagnoses it satisfies. The algorithm also checks if a constituent diagnosis mentions the component mode variable, x_{im}, but not the particular component mode assignment. In this case, the component mode assignments are reduced because no assignments associated with x_{im} can be used to satisfy that constituent diagnosis. Within each constituent diagnosis is a counter indicating the different component mode variables, x_{im}, it contains. The algorithm uses this to detect dead ends and near dead ends. If a constituent diagnosis does not have any more component mode variables it can use, then the mode estimate can never be a satisfying solution. This is represented when the counter in the constituent diagnosis is equal to zero. The detection of a near dead end is when this counter is 1, representing that the Constituent diagnosis only has one more component mode variable that it can use. The algorithm places this constituent diagnosis so that it is the next to be expanded.

Once the 'add-ConstituentDiagnosis-assignment' has determined that the component mode assignment does not make the *node* a dead end, it adds the component mode assignment to the mode estimate and updates the cost of the *node* using the mode estimate probability equation. The equation is simplified since the mode estimate is generated incrementally. All that is required is to multiply the current probability of the mode estimate, $P(S_j^{(t+1)})$ by the transition probability, $P_{\mathbf{T}}(x_{im} = v_{ij})$ of the component mode assignment. To complete the calculation of the CDA* heuristic developed in Chapter 6, the product of the highest transition probabilities of components not yet assigned a mode are used in the h(n) heuristic. This is added to the probability of the mode estimate, given by the heuristic, g(n). Calculating this gives the desired optimistic estimate for the search heuristic.

The next supporting algorithm used within the 'Expand-and-Insert' function is the 'update-$\Pi_m^{VCM'}$' algorithm. The task of this algorithm is to remove component mode assignments that are not allowed along a certain path of nodes. The 'Expand-and-Insert' algorithm builds up this list of component mode assignments as it expands from left to right. The 'update-$\Pi_m^{VCM'}$' algorithm uses the '*do-not-use*' list of assignments from the 'Expand-and-Insert' algorithm to perform this task. The algorithm is detailed below.

---

function update-Π_m^{VCM*}(*node*, *do-not-use*)
    returns the *node* after updating Π_m^{VCM*}
    for each $(x_{im} = v_{ij})$ in *do-not-use*
        node-Π_m^{VCM*} $\leftarrow$ remove $(x_{im} = v_{ij})$ from Π_m^{VCM*} of *node*
    end
    if $\exists\, x_{im}$ in Π_m^{VCM*} there are no more assignments & $x_{im} \notin S_j^{(t+1)}$ of *node*
        then mark *node* as a dead end
    return *node*

Figure 7-10 - Update Allowable Assignments Supporting DDA* Algorithm

The 'update-Π_m^{VCM}' algorithm not only removes component mode assignments, but also checks for a dead end. If by removing enough component mode assignments, it is possible that all Reachable component modes could be removed for a component mode variable, x_{im}. In this case, the node would not be able to ever be a complete mode estimate, so the algorithm marks it as a dead end.

Once the node has been updated by the 'add-ConstituentDiagnosis-assignment' or 'add-Variable-assignment', and the 'update' algorithms, the 'Expand-and-Insert' algorithm checks the *node* to see if it has been marked as a dead end. If it has, then the *node* is never added to the queue and is thrown out. However, if the *node* is not marked as a dead end, then it is ready to be inserted into the queue, *Nodes*. The 'Insert-Node' algorithm performs this task by iterating through the queue to determine the point where the *node* should be inserted. The algorithm maintains the queue in order of decreasing cost, as calculated by the heuristic equations given in Chapter 3. The 'Insert-Node' algorithm is specified below.

---

function Insert-Node(*new node, Nodes*)
    returns *Nodes*, updated with *new node*
    for each *node* in *Nodes*
        if cost(*new node*) = cost(*node*)
            then put *new node* after *node* in *Nodes*
        if cost(*new node*) < cost(*node*) & cost(*new node*) > cost(*node*+1)
            then put *new node* between *node* and *node*+1 in *Nodes*
    end
    return *Nodes*

Figure 7-11 - Insert Node Algorithm Supporting the DDA* Algorithm

The 'Insert-Node' algorithm is designed to be similar to the insert algorithm for the Generate algorithm. The first condition states that if two nodes have equal cost, then the tie goes to the node on the queue. This eliminates the

potential for greedy search. The second condition states that if the 'node' is between two values in 'Nodes', then it should be placed in between these two nodes.

The remaining algorithms that enable the 'Expand-and-Insert' algorithm of the CDA* algorithm are the 'copyNode', and 'ConstituentDiagnosis-to-Expand' algorithms. The 'copyNode' algorithm is rather straightforward. It copies every field within a *node* including the current set of component mode assignments, the list of remaining constituent diagnoses to be satisfied, and the list of allowable component mode assignments. The other algorithm, the 'ConstituentDiagnosis-to-Expand' simply extracts the next best constituent diagnosis stored within the node. Recall that the 'add-ConstituentDiagnosis-assignment' determined the best Constituent diagnosis to expand, as described in Figure 7-9.

These descriptions and the prior specifications complete the detail of the CDA* algorithm and all of its supporting algorithms. These enable the CDA* algorithm to perform the search for an optimal set of component mode assignments that satisfy the constituent diagnoses. Once the CDA* algorithm has determined this, it returns the mode estimate and the current queue, *Nodes*, to the Generate algorithm. The Generate algorithm will use the queue the next time it uses the previous mode estimate, $S_i^{(t)}$, associated with this queue. The CDA* algorithm is Conflict Directed A* algorithm used in OPSAT [*Williams, 2002*], with systematicity from [*Ragno, 2002*]. This algorithm guarantees the generation of only consistent mode estimates by using the constituent diagnoses. Through the framing of this algorithm as an A* search, the CDA* algorithm also guarantees that the fewest number of nodes are expanded.

7.2.3 Rank

The final phase in generating a current belief state is to rank the mode estimate generated by the CDA* algorithm. This requires calculating the posteriori probability of the mode estimate, as defined in Chapter 6. To perform this calculation, the Rank algorithm uses the current mode estimate, the enabled transitions and previous mode estimates, to calculate the posteriori probability using Equation 6-9 through Equation 6-11. Once the posteriori probability has been calculated, the current mode estimate can be appropriately inserted into the current belief state, $B^{(t+1)}$. The inputs and outputs of the Rank algorithm are shown below.
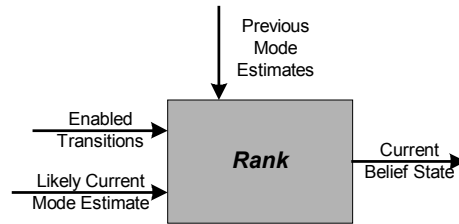
Figure 7-12 - Inputs and Outputs of the Rank Algorithm

The steps of the Rank algorithm, as explained in Chapter 6, begin with determining if the current mode estimate, $S_j^{(t+1)}$ already exists in the current belief state. To determine this, the algorithm iterates through the mode estimates in the current belief state, and compares the current mode estimate to these for equality. Equality is defined as containing the same, identical state. If the mode estimates are equal, then the current belief state is unchanged. If the current mode estimate does not exist in the current belief state, then the Rank algorithm proceeds to calculate the total probability of the current mode estimate.

The total probability calculation requires iterating through each previous mode estimate and determining if the component mode assignments in a

Achieving Real-time Mode Estimation through Offline Compilation   213

given previous mode estimate, $S_i^{(t)}$, can transition to the component mode assignments in the current mode estimate, $S_j^{(t+1)}$. Computing this, as specified in Chapter 6, requires identifying if there is an enabled transition for each pair of component mode assignments, where the source is the component mode in the previous mode estimate and the targer is the component mode in the current mode estimate. If there is an enabled transition for each pair, then the transition probability is non-zero, and is calculated by the Rank algorithm. The algorithm is detailed below.

---

function Rank($S_j^{(t+1)}$, $B^{(t)}$, $B^{(t+1)}$, T_{EN})
    returns $B^{(t+1)}$ when Generate exits, otherwise returns $\langle S_j^{(t+1)}, P(S_j^{(t+1)}) \rangle$, if
possible
    for each $S_m^{(t+1)}$ in $B^{(t+1)}$
        if $S_j^{(t+1)} = S_m^{(t+1)}$
            then return $\langle S_j^{(t+1)}, P(S_j^{(t+1)}) \rangle$ with $P(S_j^{(t+1)}) = 0$.
    end
    $P(S_j^{(t+1)} \mid B^{(t)}) \leftarrow 0$
    for each $S_i^{(t)}$ in $B^{(t)}$
        if $\forall\ (x_{im} = v_{ij}) \in S_i^{(t)}$ there exists a $T_i \in T_{EN}$ where a $(x_{im} = v_{in}) \in S_j^{(t+1)}$ is
the target
            then
                $P(S_j^{(t+1)} \mid S_i^{(t)}) \leftarrow P(S_i^{(t)}) \bullet \Pi\ P_T(\ (x_{im} = v_{ij}) \in S_i^{(t)} \rightarrow (x_{im} = v_{in})$
$\in S_j^{(t+1)})$
                $P(S_j^{(t+1)} \mid B^{(t)}) \leftarrow P(S_j^{(t+1)} \mid B^{(t)}) + P(S_j^{(t+1)} \mid S_i^{(t)})$
    end
    $B^{(t+1)} \leftarrow$ Insert-in-Order($B^{(t+1)}$, $S_j^{(t+1)}$)
    return $B^{(t+1)}$ when Generate exits, otherwise **return** $\langle S_j^{(t+1)}, P(S_j^{(t+1)}) \rangle$

Figure 7-13 - Rank Algorithm

The first steps of the Rank algorithm determine if the current mode estimate is equivalent to any mode estimate in the current belief state. If this is not the case, then the algorithm proceeds to calculate the posteriori probability by first initializing $P(S_j^{(t+1)} \mid B^{(t)})$, to be zero. Then, the algorithm iterates through the previous mode estimates and for an $S_i^{(t)}$, if a transition exists to the current mode estimate, then the transition probability is calculated per Equation 6-9 and Equation 6-10. This transition probability is then

summed to the running total $P(S_j^{(t+1)} \mid B^{(t)})$. Once the posteriori probability is calculated, the mode estimate is inserted in order of decreasing probability in the current belief state. The algorithm that performs this operation, the 'Insert-in-Order' algorithm, is the same as the 'Insert-Node' algorithm defined for the CDA* algorithm, given in Figure 7-11.

The specification of the Rank algorithm completes the algorithm definitions for the Online Mode Estimation process. These algorithms work together to map the compiled model, current observations and control variables to a set of consistent mode estimates, ordered from most likely to least likely. The final step is to tie the Compiled Conflict Recognition and the Dynamic Mode Estimate Generation algorithms together.

7.3 Online Mode Estimation

This algorithm drives the process of mode estimation during the time the spacecraft system is executing operations. The algorithms given thus far for the mode estimation process were designed to generate the current belief state between times 't' and '$t+1$'. The final phase of mode estimation is to perform these computations as time marches forward and track the system over time. The inputs and outputs of this process are shown below.
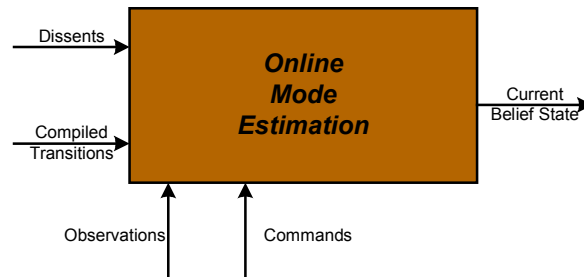
Figure 7-14 - Inputs and Outputs for Online Mode Estimation

The Online Mode Estimation algorithm ties together the algorithms of the Compiled Conflict Recognition and of the Dynamic Mode Estimate Generation. Online Mode Estimation calls the algorithms of the Compiled Conflict Recognition in a particular order. First, the truth-values of the observations and commands must be updated before any triggering can occur. Once this process is successful, the Dissent and Transition Trigger algorithms are invoked to create the lists of Enabled Dissents and Enabled Transitions. The Constituent Diagnosis Generator uses these inputs, along with the internal Previous Mode Estimates to determine the Constituent Diagnoses, the Reachable Current Modes, and passes along the Enabled Transitions.

The Online Mode Estimation algorithm then invokes the Generate algorithm that drives the computation of the current belief state from the Constituent Diagnoses, the Reachable Current Modes and the Enabled Transitions. The Online-Mode-Estimation algorithm is detailed below.

---

function Online-Mode-Estimation(*Dissents, $T_{COMPILED}$, $\Pi_o^{Current}$, $\Pi_c^{Current}$*)
    returns a current belief state, $B^{(t+1)}$
  [Π_m, Π_o, Π_c] $\leftarrow$ initialize assignment types once
  loop do
      if $B^{(t)}$ is empty
        then
          [$\Pi_o^{Changed}$, $\Pi_c^{Changed}$] $\leftarrow$ Initialize-Truth(Π_o, Π_c, $\Pi_o^{Current}$, $\Pi_c^{Current}$)
          $\Pi_m^{Previous} \leftarrow \Pi_m$
          $DS_{EN} \leftarrow$ Dissent-Trigger($\Pi_o^{Changed}$, *Dissents*)
          [$\Pi_m^{Current}$, *CD*, *empty*] $\leftarrow$ Constituent-Diagnosis-Generator(DS_{EN}, *empty*, $\Pi_m^{Previous}$)
          $B^{(t+1)} \leftarrow$ Generate(*empty*, $\Pi_m^{Current}$, *CD*, *empty*)
          return $B^{(t+1)}$

        else
          [$\Pi_o^{Changed}$, $\Pi_c^{Changed}$] $\leftarrow$ Update-Truth(Π_o, Π_c, $\Pi_o^{Current}$, $\Pi_c^{Current}$)
          $\Pi_m^{Previous} \leftarrow$ Compress-States($B^{(t)}$)
          $DS_{EN} \leftarrow$ Dissent-Trigger($\Pi_o^{Changed}$, *Dissents*)
          $T_{EN} \leftarrow$ Transition-Trigger($\Pi_o^{Changed}$, $\Pi_c^{Changed}$, $\Pi_m^{Previous}$, $T_{COMPILED}$)
          [$\Pi_m^{Current}$, *CD*, T_{EN}] $\leftarrow$ Constituent-Diagnosis-Generator(DS_{EN}, T_{EN}, $\Pi_m^{Previous}$)

$$B^{(t+1)} \leftarrow \text{Generate}(B^{(t)}, \Pi_m^{\text{Current}}, \mathit{CD}, T_{\mathit{EN}})$$
$$\textbf{return } B^{(t+1)}$$
while(*true*)

Figure 7-15 - Online Mode Estimation Algorithm

The algorithm shown here is only a skeleton that makes the appropriate invocations of the algorithms detailed previously. The Online Mode Estimation algorithm must be capable of interfacing with a real system. This results in the need for an interface for the 'Observations' and 'Commands'. This has not been specified because an interface of this type changes for each individual system.

The Online-Mode-Estimation algorithm is not necessarily an algorithm that ever exits under normal operation. The algorithm is executing in parallel with many other processes in the system and continuously determining mode estimates. When the system requires a mode estimate, the algorithm returns it to the system. This type of design enables the Online-Mode-Estimation algorithm to not only be used in a real time system, but to enable the architecture of the Model-based Executive presented in Chapter 1.

Compiled Mode Estimation performs the specified function within the Model-based Executive of providing mode estimates representative of the system behavior. It maps the system model, observations and commands to a set of mode estimates. Compiled Mode Estimation is able to use multiple sources of information to determine the mode estimates, is able to track multiple system trajectories at each time step, increasing the accuracy of the mode estimates. Additionally, CME is able to diagnose single and multiple faults. These are the desired capabilities specified in Chapter 1 of the next evolution of the mode estimation engine.

This brings to a close the description of the theory, and algorithms associated with Compiled Mode Estimation. Chapter 2 discussed the compilation process to obtain dissents and compiled transitions from a system model. Chapter 3 described in detail the use of this compiled information in performing online mode estimation, giving the main ideas and detailing the necessary computations. This chapter presented the formal algorithms that perform the Online Mode Estimation process. The formal algorithms that describe the compilation of the system model are presented in Appendix A.

The Online Mode Estimation produces consistent mode estimates that agree with the system model, observations and commands. The goal of this research was to not just develop a working mode estimation engine, but to also validate this approach to mode estimation. The validation of an algorithm of this type can only be done through experimentation and verification of the results against an existing system. The next chapter discusses the validation of the Compiled Mode Estimation approach using the NEAR spacecraft.

This page intentionally left blank.

8 Experimental Validation

The Compiled Mode Estimation system and algorithms have been developed through the presentation of previous mode estimation approaches and the process of compilation in Chapters 2, 3, 4 and 5. The algorithms that make use of the compiled model to perform mode estimation were given in Chapters 6 and 7. The next step is to validate CME through experimentation.

Our experiments include CME operating on scenarios of nominal operation and component failures. These scenarios specify sequences of observations and command values, while CME determines the expected behavior of the system. The experiments will demonstrate that CME correctly determines the expected behavior of the system. The experiments support the claim that the compiled model requires a smaller memory footprint than the full model. In addition, the set of dissents enable the diagnoses, which CME produces, to be inspectable for correctness by a human before they are needed by the system.

Our example is drawn from the NEAR spacecraft. An artist's depiction of the NEAR spacecraft is shown below.
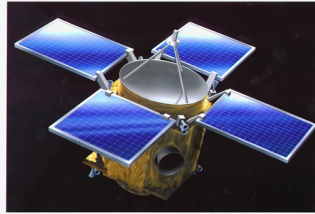
Figure 8-1 - Artist's Depiction of the NEAR Spacecraft

Recall that the Near Earth Asteroid Rendezvous mission was ground breaking for the Johns Hopkins University Applied Physics Lab. This spacecraft rendezvoused with the Eros asteroid appropriately on February 14, 2000. NEAR mapped the surface completely and performed experiments to determine the composition of the asteroid. The NEAR spacecraft provided a wealth of information over its mission lifetime of 2 years. Of the many systems on-board the spacecraft, the power system is one of the most essential to the operation of the spacecraft. Without the necessary power, the spacecraft would be rendered inoperable, so it is critical that the power system operate even in the face of failures.

The presentation of the Compiled Mode Estimation process has relied on the power storage system from the NEAR spacecraft to demonstrate the theory and algorithms. The validation experiments developed in this chapter use the entire NEAR power system. The models of the power system are presented in Section 8.1, followed by the compiled model in Section 8.2. The experiments designed to use these models to test the Compiled Mode Estimation and the results of these experiments are presented in Section 8.3. The chapter concludes with a discussion of the results in Section 8.4.

8.1   NEAR Spacecraft Power System

The use of existing systems enables the detailed modeling necessary for mode estimation. In using existing systems, the components, sensors and component interactions are understood, specified and well documented. Additionally, potential failures have been determined for existing systems, and there is a wealth of information for failures that have occurred in previous systems. The experiments focus on these failures for the NEAR Power system and test if the compiled mode estimation algorithms can properly estimate the modes of the components to diagnose these failures.

This section details some of the component models of the NEAR Power system by first presenting the power system block diagram, and then detailing some of the individual component models. Any that are not presented here are given in Appendix A. After presenting the component models, the compiled model is given in section 8.2.

8.1.1   System Block Diagram

The NEAR power system was designed as a direct energy transfer (DET) system. Scientific devices and spacecraft components are designed to use a specific voltage level, so the power system must regulate the incoming power to this level. The DET design uses mechanisms to dissipate power to regulate the voltage and current in the spacecraft. The schematic of the NEAR Power system is shown in Figure 8-2. This figure was presented earlier in Chapter 1, and is presented now for clarity, with all pertinent components labeled.
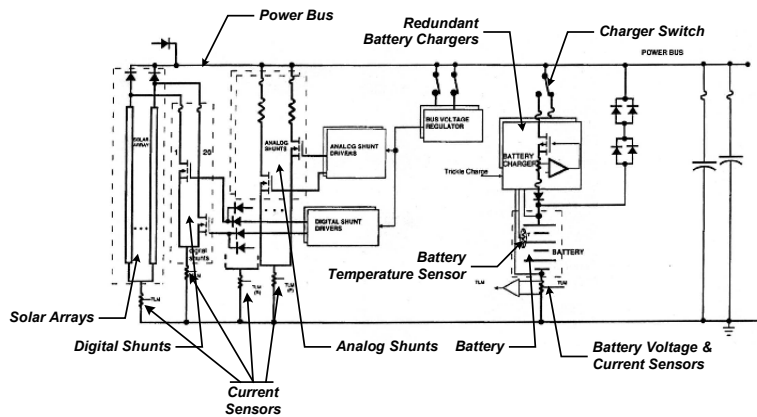
Figure 8-2 - NEAR Power System Schematic

Noted on the figure are the main components of the system, the solar arrays, the primary and redundant digital and analog shunts, the switch for the chargers, the redundant chargers and the battery. A shunt is the least intuitive component in the power system. It acts to dissipate power generated from the solar arrays. The two types, digital and analog, can be thought of as switches, that when closed dissipate power, and when open allow power to flow to the bus. Noted on the figure are the sensors in the system, the current sensors for the solar arrays, one for the primary and redundant digital shunts, one each for the primary and redundant analog shunts, and a temperature and voltage sensor for the battery. These sensors are used to extract the observation information from the power system.

The components in the schematic that are not referenced are the digital and analog shunt drivers, as well as the bus voltage regulator. The digital and analog shunt drivers send commands to the digital and analog shunts to open or close a certain number of shunts to dissipate the appropriate power. The bus voltage regulator is the source of these commands. This component can be thought of as a software process that determines which commands to send to the drivers to dissipate the appropriate amount of

Achieving Real-time Mode Estimation through Offline Compilation   223

power. These components are not modeled in this experiment. Instead the drivers and bus voltage regulator are abstracted away and the commands are an input to the system model, specifically the digital and analog shunts. Removing these components from the system model simplifies the model slightly, but does not take away from the complexities that the model expresses. The encoding of software processes is an extension to the modeling language used for this experiment.

8.1.2 Component Models

A new representation of the NEAR Power system is developed using the simplification described above. Figure 8-3 depicts the simplified schematic.
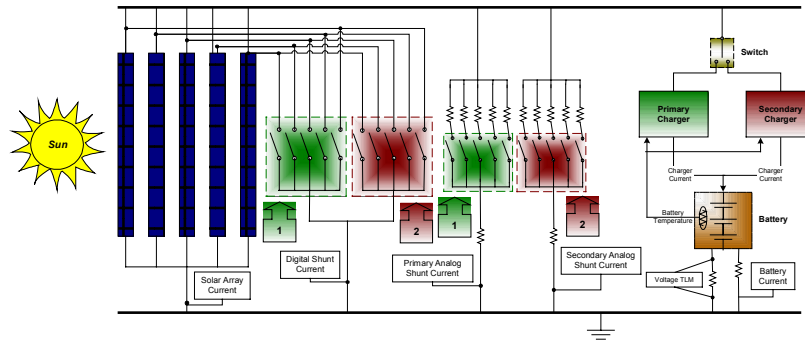
Figure 8-3 - Schematic of Simplified NEAR Power System

Noted on the schematic are the redundant digital and analog shunts. Drawn around the digital shunts and the solar arrays is a box that denotes a single solar array panel. The NEAR spacecraft has four solar panels, as shown in Figure 8-1, and each solar panel has five solar cell groups, depicted in Figure 8-3. This schematic is broken down into the following representation that shows the components, their inputs and outputs and all observation and commands that are within the system.
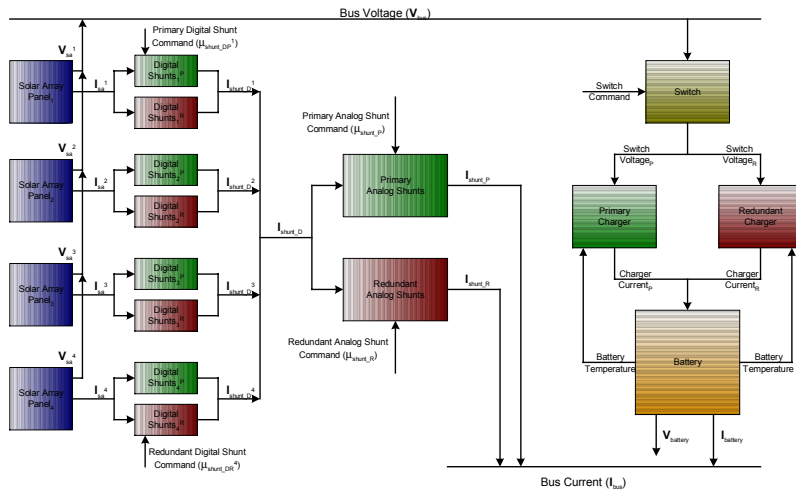
Figure 8-4 - NEAR Power System Block Diagram

Some of the components noted on the figure are detailed in the following sections with the remaining in Appendix A. By removing the digital and analog shunt drivers, the commands for the shunts must now be specified as inputs to the system. The figure denotes these commands for the analog shunts and digital shunts. The following sections detail the models of the battery and chargers. Their complex interaction results in a complex failure scenario. The reader is referred to Appendix A to review pertinent models to understand the specific scenarios and results presented in this chapter.

8.1.3 Charger

The power system chargers use the input voltage from the spacecraft power bus, noted as the *bus-voltage*, and transform it into a current to charge the battery. The internal pieces that perform this transformation are too complex to model individually. The chargers are not modeled to this level of detail because there is no observability into the operation of the chargers.

There is only the observable of the output of the charger, the *charger-current*. There is no direct observable of the input to the charger, the voltage coming from the *switch*. However, the additional information of the *battery-temperature* enables the models of the charger to use this information to determine its mode, or how it is charging the battery. This additional input allows the modeling of the charger at a high level, neglecting the internal specifics of the charger.

The modes of the charger are specified by determining the interaction between the charger and the battery. For instance, if the battery temperature is *nominal*, then the battery level of charge is not full, so the charger can continue to charge it. However, if the battery temperature is *high*, then this indicates that the battery charge level is full, so the charger only needs to trickle-charge the battery to keep it full. Using these characteristics, the model of the charger is specified below in Figure 8-5.
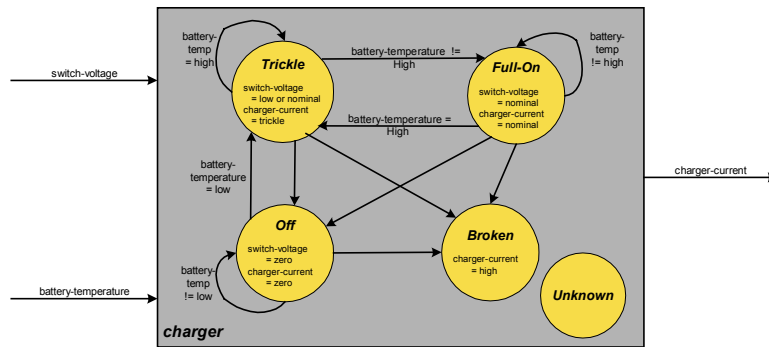
Figure 8-5 – Constraint Automaton of the NEAR Power System Chargers

The model uses the input *switch-voltage* and the output *charger-current* to constrain the modes of the charger. The *switch-voltage* has the domain {*zero, low, nominal*}, and the *charger-current* has the domain {*zero, trickle, nominal, high*}. The charger mode *trickle* is characterized by a *low* or *nominal* input *switch-voltage*, and a *trickle* output for the *charger-*

current. This mode is modeled to capture the behavior of normal operation on the spacecraft. The spacecraft should be using most of the power generated from the solar array and the battery should be fully charged most of the time. As a result, the battery only needs to be trickle charged to maintain its full charge. The next operational mode is denoted by the *full-on* mode for the charger. This mode models the charger having a higher amount of voltage to charge the battery, indicated by the *switch-voltage* being at *nominal*. The output current is then constrained to be *nominal* indicating that the battery requires more of a charge to get it back to the full level. The final operational mode, *off*, denotes that the charger has been turned off because there is no input voltage from the bus, indicated by the value *zero* for the *switch-voltage*. As a result, the charger can only have one output value, a *charger-current* of *zero*.

The failure modes for the charger include a *broken* and an *unknown* mode. The *broken* mode captures the behavior that the *charger* has a short in it that is causing the output current to be *high*. As a result, the charger has failed in some way, and cannot be used any more. When this happens, the redundant charger is then used to charge the battery. The automatic changing of the switch is expressed using the following constraints between the *switch-command* and the *charger-current*:

$$(\mathbf{if}\ (\textit{charger-current}^P = \textit{high}) \Rightarrow (\textit{switch-command} = \textit{to-charger-p}))$$
$$(\mathbf{if}\ (\textit{charger-current}^R = \textit{high}) \Rightarrow (\textit{switch-command} = \textit{to-charger-r}))$$

These constraints enable the *switch* to move to the *charger-p* or *charger-r* position automatically when a charger fails. When a charger has failed, then the *switch* can no longer be at that position, and the mode constrains this automatically. When a charger fails in the *broken* mode, no other operational modes are allowed ever again, which is restricted by the transitions. The *unknown* fault mode captures any other behavior of the battery charger that has not been considered.

The discussion of the transition system of the charger is detailed in Appendix A. The reader is referred to the section of the *charger* model for the expression of constraints between the output *charger-current* and the input to the *battery*.

8.1.4 Battery

The battery is the NEAR Power system's means to store excess power generated from the solar arrays for later use. It is also the NEAR spacecraft's means to operate the spacecraft in the event that the solar arrays cannot provide the necessary power for the spacecraft. This can happen on many occasions during the normal operation of the spacecraft. For instance, if the NEAR spacecraft flies into the shadow of an object, such as the Earth or the Eros asteroid, then the battery would provide power to the spacecraft. The battery has different levels of being charged, indicated by it either being full, charging, discharging and dead. These behaviors are captured in the model of the battery in Figure 8-6.

The battery uses the inputs of the *charger-current* and the outputs *battery-voltage*, *battery-current*, and the *battery-temperature* to constrain the modes. The input *charger-current* is used to transition between the modes of the battery, and the constraints on the modes are expressed using the outputs. Recall that in the previous modeling of the NEAR Power Storage system, only the battery voltage was considered as an output of the battery. However, having now included the remaining components of the NEAR Power system, it becomes necessary to include the battery current as an output because it adds to the output current of the analog shunts of the power generation components. The resultant component model is shown below in Figure 8-6.
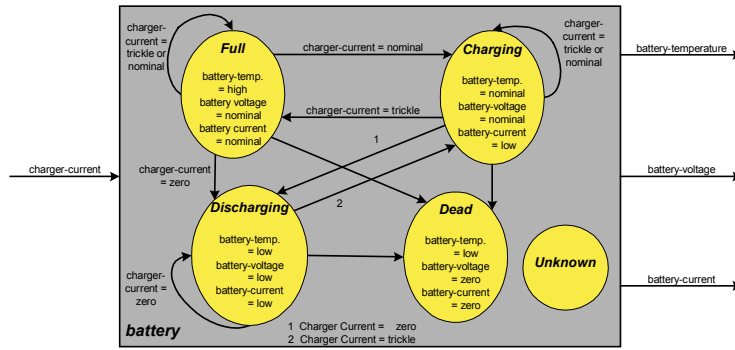
Figure 8-6 – Constraint Automaton of the NEAR Power System Battery

The operational modes of the battery are given as *charging*, *full*, and *discharging*. The component mode *charging* is characterized by a *nominal battery-voltage*, a *nominal battery-temperature* and a *low battery-current*. This combination of output values indicates that the level of charge in the battery is not where it should be, so it needs to be put on a full charge. The *full* mode is characterized by a *high* reading for the *battery-temperature*, a *nominal* reading for the *battery-current* and a *nominal* reading for the *battery-voltage*. This combination of values indicates that the battery level of charge is full and that it only needs to be kept at this level by the battery charger. Notice that the reading of the battery current has changed, but the voltage level stays the same. The battery always maintains the same voltage level, but the level of charge is indicated by the current and the temperature. In the case of the *discharging* mode, the output values are given by a *low* reading for the *battery-temperature*, a *low* reading for the *battery-voltage* and a *low* value for the *battery-current*. These values indicate that the battery temperature has dropped because the chargers are no longer heating it up through charging. Also, the *battery-voltage* and the *battery-current* have also dropped below the normal values to *low* because the level of charge in the battery has decreased.

The fault modes of the battery are given as a *dead* and an *unknown* mode. When the battery is dead, it no longer has any charge, resulting in the loss of the spacecraft. The output values given in the model for this component mode are a *low* value for the *battery-temperature*, a *zero* value for both the *battery-voltage* and the *battery-current*. These values characterize when the battery does not have charge remaining so it cannot discharge any voltage or current. The final mode of the battery, *unknown*, captures any behavior not modeled with these component modes.

The operational modes of the battery transition to other modes based on the value of the input *charger-current*. The transitions constrained by the *charger-current* are between the modes *charging*, *full*, and *discharging*. The battery transitions from the *charging* mode to the *full* mode when the input *charger-current* is at the *trickle* level. This constraint also characterizes the transition of the battery from *discharging* to *charging*. There is only one transition to an operational mode allowed from the *discharging* mode for the same reason expressed with the chargers. When the battery no longer needs to supply extra power to the spacecraft for its operations, there will not be an excess of power to allow for the full charging of the battery. As a result, the battery will only be able to begin charging using a trickle charge. The final operational transition allowed is between the *full* mode to the *charging* mode. This transition is allowed only if the input *charger-current* is *nominal*, because the battery level of charge is lower than full, requiring as much current as possible to get the level of charge back to full.

The final step of the model for the NEAR Power system is to constrain the inputs and outputs of the components to be the same. These constraints link the components together and are expressed in the concurrent constraint automaton that incorporates these individual constraint automata. The models given here capture the behaviors of the NEAR Power system and

are expressed as a concurrent constraint automaton. However, to develop the simulation for the Compiled Mode Estimation system, the CCA must be compiled into dissents and compiled transitions.

8.2   Compiled Model

The NEAR Power system having been developed using concurrent constraint automata must be transformed for the Compiled Mode Estimation system. The compiled model is presented below to show the compactness of the model. The uncompiled model specified above has not only individual component modes and their constraints, but also constraints on intermediary variables between components. The compactness of the compiled model allows for a human to determine correctness without requiring the need to reason over the entire system. The following figures denote the compiled transition systems of the individual components in the NEAR Power system.
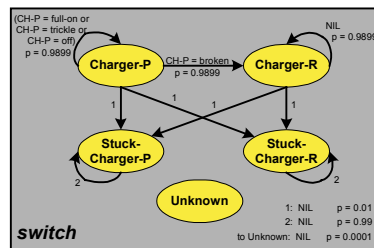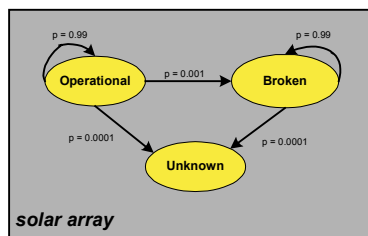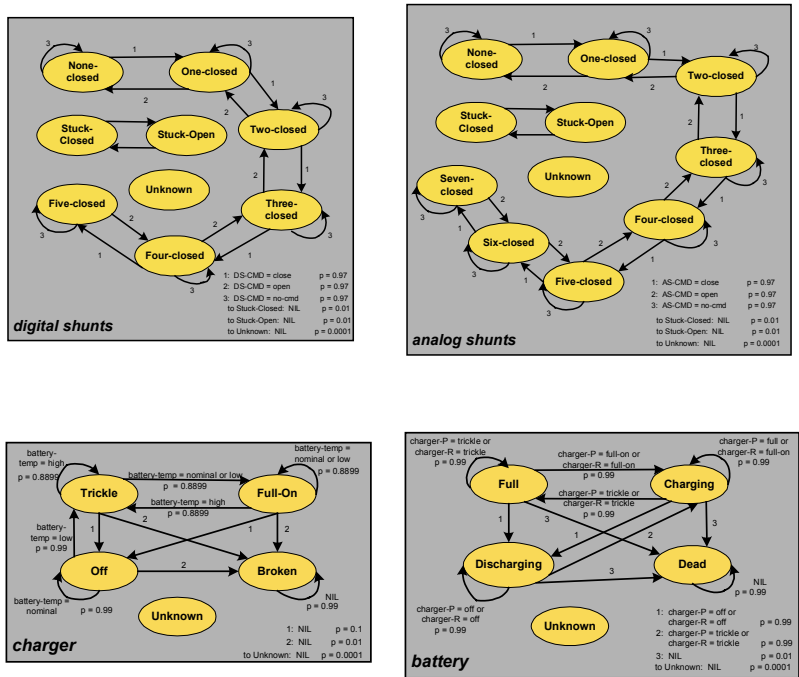
Figure 8-7 - Compiled Transition Function for Each Component

The full list of dissents for the compiled model is given in Appendix D. A sampling of these dissents are given below.

```
1. BUS-VOLTAGE=ZERO      -> ¬SOLAR-ARRAY-1.MODE=OPERATIONAL

2. BUS-VOLTAGE=LOW       -> ¬SOLAR-ARRAY-1.MODE=OPERATIONAL

3. SOLAR-ARRAY-CURRENT=ZERO -> ¬SOLAR-ARRAY-1.MODE=OPERATIONAL

4. SOLAR-ARRAY-CURRENT=LOW  -> ¬SOLAR-ARRAY-1.MODE=OPERATIONAL

5. BATT-CURRENT=ZERO      -> ¬BATTERY.MODE=CHARGING

6. BATT-CURRENT=NOMINAL -> ¬BATTERY.MODE=CHARGING

7. BATT-TEMPERATURE=LOW  -> ¬BATTERY.MODE=CHARGING

44. BUS-VOLTAGE=LOW -> ¬[ CHARGER-P.MODE=OFF,  SWITCH.MODE=
        CHARGER-P ]

45. BUS-VOLTAGE=NOMINAL -> ¬[ CHARGER-P.MODE=OFF, SWITCH.MODE=
        CHARGER-P ]

46. BUS-VOLTAGE=LOW -> ¬[ CHARGER-P.MODE=OFF, SWITCH.MODE=
        STUCK-CHARGER-P ]
```

```
47.  BUS-VOLTAGE=NOMINAL -> ¬[ CHARGER-P.MODE=OFF, SWITCH.MODE=
        STUCK-CHARGER-P ]
48.  BUS-VOLTAGE=ZERO -> ¬[CHARGER-P.MODE=FULL-ON, SWITCH.MODE=
        CHARGER-P ]
49.  BUS-VOLTAGE=LOW -> ¬[ CHARGER-P.MODE=FULL-ON, SWITCH.MODE=
        CHARGER-P ]
50.  BUS-VOLTAGE=ZERO  -> ¬[CHARGER-P.MODE=FULL-ON, SWITCH.MODE=
        STUCK-CHARGER-P ]
51.  BUS-VOLTAGE=LOW -> ¬[ CHARGER-P.MODE=FULL-ON, SWITCH.MODE=
        STUCK-CHARGER-P ]
52.  BUS-VOLTAGE=ZERO -> ¬[ CHARGER-P.MODE=TRICKLE, SWITCH.MODE=
        CHARGER-P ]
53.  BUS-VOLTAGE=ZERO  -> ¬[SWITCH.MODE=STUCK-CHARGER-R,
        CHARGER-R.MODE=TRICKLE ]
```

The dissents and compiled transitions shown here offer an intuitive way to verify the possible diagnoses of the system and to verify correctness of the model. Notice in the dissents that the conflicts are localized to only a few components and observations. This enables a human to verify the correctness of a conflict very easily by inferring what is meant by the set of infeasible mode assignments. For instance, in dissent 46, the observation *bus-voltage = low* implies the conflict between the mode *charger-p = off* and *switch = stuck-charger-p*. This conflict is correct, and upon reasoning over the behaviors of the component modes, it cannot be possible that the *charger* is off if it is receiving a non-zero voltage from the bus through the switch. If there is excess power being generated, this power must be used to charge the battery.

Doing this for each dissent however does require a substantial amount of time due to the large number of dissents. For instance, the dissents for the system modeled here Instead, to verify correctness of the model, a human develops scenarios that simulate spacecraft operations, where the result of the task is already known. The following section details several scenarios

developed using these models and the results of the CME engine on these scenarios.

8.3   Scenarios and Results

The NEAR spacecraft relied on a rule-based system to handle any failures in the spacecraft system.   This rule based system mapped sensor information to recovery actions.   The behavior of the system is implicit in this rule because a human modeler developed the rule by reasoning through the component interactions.   It is the aim of this validation experiment to show that the Compiled Mode Estimation diagnoses these failures, and combinations of these failures.   This will demonstrate that CME is capable of not only diagnosing failures in the NEAR rule set, but can diagnose multiple simultaneous failures.   In addition, CME can determine many more failures by reasoning about many different combinations of component modes and is not restricted to a specified set of failures as in a rule-based system.

The power system of the NEAR spacecraft has several associated rules to handle failures.   The complete NEAR rule set incorporates over 150 rules for its eight sub-systems, and the nine associated with the power system are listed in Figure 8-8.   The rules were designed to only handle critical component failures that have potential to cause the loss of the mission.

22	(Id>1A) AND (Ishunt_PA>0.8A) for 10 sec	Bad_Prim_Bus_Reg
23	(Id>1A) AND (Ishunt_RA>0.8A) for 10 sec	Bad_Sec_Bus_Reg
24	(Id>1A) AND (Ishunt_D>6A) for 10 sec	Try_Sec_Bus_Reg_Off
25	(Id>1A) AND (Ishunt_D>6A) for 10 sec	Find_Bad_Bus_Reg
26	(Vbus<23V) for 15 sec	Try_Sec_Bus_Reg_Off
27	(Red Battery Charger is On) for 5 sec	rule [28,29] stop
28	(Ichr>0.8A) for 10 sec	Switch_to_Sec_Charger
29	(Ichr<0.07A) AND (Vbus-Vbatt>01.6(11counts)V) for 10 sec	Switch_to_Sec_Charger
30	(Tbatt>30°) for 1 hour	Battery_Over_Temp

Figure 8-8 - Rules for the NEAR Power System

The notation of the rules above is as follows:

- 'Id' : regulated current level on the bus.

- 'Ishunt_PA' : current from the primary analog shunts

- 'Ishunt_RA' : current from the redundant analog shunts

- 'Ishunt_D' : current of the digital shunts

- 'Vbus' : the voltage level on the bus

- 'Ichr' : the output current of the charger

- 'Tbatt' : battery temperature

The validation experiments have been tailored to these rules. The same observations are input to CME, and the result is the component modes inherent in these rules. Although the behavior of the system is not explicit in the rule, the component modes can be inferred using the observations, the resultant repair action and the system model. The discussion to follow explains the rule, and the component modes that are deduced from the rule. It is these component modes that are the desired output by the CME engine.

The necessary inputs for the system that are specified for each test are:

- Initial mode of the system

- Sequence of observations

- Sequence of commands

The output presented for each test is a screen shot of the CME engine's output. This output represents the approximated belief state with mode estimates ordered by decreasing likelihood.

The tests were conducted with the following suite of components:

- 1 solar array (SA)

- 1 primary and 1 redundant set of digital shunts (DS-P, DS-R)

- 1 set of analog shunts (AS)

- 1 switch (S)

Achieving Real-time Mode Estimation through Offline Compilation   235

- 1 primary and 1 redundant charger (CH-P, CH-R)
- 1 battery (B)

The observation and command variables, with their respective domains, are:

- Solar Array Current (Isa)   { zero, low, nominal }
- Digital Shunt Current (Ishunt_D)   { zero, low, nominal, high }
- Analog Shunt Current (Ishunt_PA)   { zero, low, nominal, high }
- Charger Current (Ichr)   { zero, trickle, nominal, high }
- Bus Voltage (Vbus)   { zero, low, nominal }
- Battery Temperature (Tbatt)   { low, nominal, high }
- Battery Voltage (Vbatt)   { zero, low, nominal }
- Battery Current (Ibatt)   { zero, low, nominal }
- Prim. Digital Shunt Command (DS-P-CMD)

                       { open, close, no-command }

- Red. Digital Shunt Command (DS-R-CMD)

                       { open, close, no-command }

- Analog Shunt Command (AS-CMD)

                       { open, close, no-command }

Note that the suite of components is not the full NEAR Power system. The example system used only includes one solar array, its associated primary and redundant digital shunts, and a single set of analog shunts instead of a primary and redundant set. This simplification has been made for testing purposes but does not impact the goals of the validation. By demonstrating that CME can diagnose failures of one solar array means that it can be extended to the remaining solar arrays by incorporating each solar array and its associated set of digital shunts into the system model. The same holds for the analog shunts. Since the primary and redundant analog shunts are in parallel, the redundant mirrors the primary. So, showing that CME can diagnose the failure of the primary analog shunts translates to a similar diagnosis of the redundant shunts since they each have an individual sensor.

8.3.1 Nominal Operation

CME does not only determine faults, but provides current behavior of the system. This includes providing the correct mode estimate under normal operations. Examples of this include engaging digital or analog shunts when commanded, or determining that the charger switches modes based on the temperature of the battery. This section details these scenarios to demonstrate that CME provides the correct mode estimate for normal operation of the digital and analog shunts, and the charger and battery.

8.3.1.1 Digital Shunt Test

This test uses CME to confirm the opening and closing of the digital shunts when commanded. The system is assumed in the following initial mode:

> { SA = operational, DS-P = none-closed, DS-R = none-closed, AS = none-closed, S = CH-P, CH-P = trickle, CH-R = off, B = full }

with the following observations:

> { Isa = nominal, Ishunt_D = nominal, Ishunt_PA = nominal, Vbus = nominal, Ichr = trickle, Tbatt = high, Vbatt = nominal, Ibatt = nominal }

In order to induce the digital shunt to close, the NEAR system would relay the commands DS-P-CMD = close and DS-R-CMD = close, since the redundant shunts shadow the operation of the primary shunts. Additionally, the system gives the analog shunts no command, which is represented by AS = no-command being input to the simulation. Once the commands are given and observations collected, then the primary and digital shunts should each be in the mode *one-closed*. The observations are unchanged because

as long as normal operation ensues, which is the assumption of this test, then the output current of the digital shunts is *nominal*, and the remaining portions of the system are not affected. The desired output for the following observations:

> { Isa = nominal, Ishunt_D = nominal, Ishunt_PA = nominal, Vbus = nominal, Ichr = trickle, Tbatt = high, Vbatt = nominal, Ibatt = nominal }

is:

> { SA = operational, DS-P = one-closed, DS-R = one-closed, AS = none-closed, S = CH-P, CH-P = trickle, CH-R = off, B = full }

The following screen shots demonstrates this test:

The input mode estimate and the current observations and commands are shown above. The result of the CME algorithm is to produce a belief state from these inputs. The figure below denotes only the most likely mode estimate in the belief state. The full belief state for this experiment is given in Appendix E.

Figure 8-9 - CME Output for Digital Shunt Normal Operation

This demonstrates CME's ability to track the mode estimates from one time step to the next and use the command correctly.

8.3.1.2   Nominal Battery and Charger Operation

This nominal operation test involves the charger and the battery. To demonstrate the nominal operation of the charger and battery, the system is assumed operating normally as in the two previous nominal tests. However, the NEAR spacecraft requires more power from the power system than the solar arrays can provide. This means that the battery must be enabled to provide the necessary power. This necessitates the battery changing from the *full* mode to now *discharging*. In addition, since the spacecraft requires more power, then this means that there is no power to charge the battery. As a result, the primary charger turns off since there is no power coming in to it. This test will demonstrate CME's ability to estimate the behavior of multiple components and their interaction.

The initial mode estimate for this scenario assumes all components are operating normally as:

> { SA = operational, DS-P = none-closed, DS-R = none-closed, AS = none-closed, S = CH-P, CH-P = trickle, CH-R = off, B = full }

The observations that correspond to this scenario are then:

> { Isa = nominal, Ishunt_D = nominal, Ishunt_PA = nominal, Ichr = zero, Vbus = zero,   Tbatt = low, Vbatt = low, Ibatt = low }

The observations of interest are the charger current and battery temperature, voltage and current. These observations indicate that the battery is discharging. Since the charge level in the battery is dropping, then the current drops as well as the battery temperature since the charger is not charging it any longer. The resulting most likely mode estimate of the system is then:

{ SA = operational, DS-P = none-closed, DS-R = none-closed, AS = none-closed, S = CH-P, CH-P = off, CH-R = off, B = discharging }

The output from the CME engine for this scenario is as follows:

Using the inputs shown above, the CME engine produces the following mode estimates.

The mode estimates shown above have the exact same probability. The only difference between the two is the mode assignment for the *switch* is *charger-p* in one and and *charger-r* in the other. This results because since there is no incoming bus voltage, the switch could be at either position.

Figure 8-10 - CME Engine Output for Nominal Charger and Battery Operation

8.3.2   Primary Analog Shunt Failure

The first failure scenario considered involves the analog shunts. This test will demonstrate CME's ability to use commands and the conflicts to correctly identify faulty behavior. A shunt can either fail in the open position or in the closed position and will remain that way. This failure scenario involves a shunt failing in the open position. A shunt that fails in this manner causes the output shunt current to be higher than expected because the system believes that the shunt should be closed, thus dissipating power. However, if the shunt remains open, the power is not dissipated, causing a higher output current than expected.

This scenario corresponds to rules 22 and 23 of the NEAR fault management system. This rule states that if the bus current, Id, is greater than 1.0 A and the analog shunt current, Ishunt_PA or Ishunt_RA, is greater than 0.8 A, then the group of shunts has failed. The symptom then states that if the analog shunt current is high, then the bank of analog shunts has failed. Due to a lack of observability of the shunts, the symptom only identifies the entire bank of shunts as failed, but cannot identify any one particular shunt.

The experiment for this scenario begins with all components operating in the modes below:

{ SA = operational, DS-P = none-closed, DS-R = none-closed, AS-P = none-closed, S = CH-P, CH-P = trickle, CH-R = off, B = full }

The following commands are then issued to the system:

{ DS-P-CMD = no-command, DS-R-CMD = no-command, AS-CMD = close }

The resultant observations of the system, denoting the high analog shunt current, are:

{ Isa = nominal, Ishunt_D = nominal, I_shunt_PA = high, Ichr = nominal, Vbus = nominal, Tbatt = high, Vbatt = nominal, Ibatt = nominal }

The most likely mode estimate of the system is then:

{ SA = operational, DS-P = none-closed, DS-R = none-closed, AS-P = stuck-closed, S = CH-P, CH-P = trickle, CH-R = off, B = full }

This set of component modes demonstrates that by observing a high current from the analog shunt output means that an analog shunt has failed in the *stuck-open* position. The resulting output from the CME engine for this is given below:

The observations, commands and initial mode estimate then generate the likely mode estimates:

Figure 8-11 - CME Output for a Failed Analog Shunt

242   Achieving Real-time Mode Estimation through Offline Compilation

The figure above shows that the CME engine has correctly determined that the analog shunts have failed in the stuck-open position. The remaining mode estimates in the belief state are given in Appendix C.

8.3.3   Failed Charger

The next failure considered involves the charger in the NEAR Power storage system.   A charger failure is indicated by the output current exceeding a threshold.   Rule 28 in the NEAR fault management system is associated with this type of failure.   If the charger current exceeds 0.8 A, then there is a short within the charger causing a high output current.   In the discrete modeling, this is indicated by the observation *Ichr = high*.   The charger failure offers an interesting characteristic.   A result of the charger failing, is the switch immediately is moved to the *charger-r* position and *charger-r* is turned on so it can charge the battery.   So, this demonstrates that the CME engine is capable of determining if multiple components changed modes at the same time.

The experiment for this scenario begins with the components in the following modes:

> { SA = operational, DS-P = none-closed, DS-R = none-closed, AS
> = none-closed, S = CH-P, CH-P = trickle, CH-R = off, B = full }

The following observations are made, with no commands being given to the shunts.

> { Isa = nominal, Ishunt_D = nominal, Ishunt_PA = nominal, Ichr =
high, Vbus = nominal, Tbatt = high, Vbatt = nominal, Ibatt = nominal }

> { DS-P-CMD = no-command, DS-R-CMD = no-command, AS-
CMD = no-command }

The diagnosis of the failed charger using the above observations first identifies the failed charger, CH-P. Since it is the only charger that is on then the observation *Ichr = high* reflects the behavior of this component. Next, CME determines that because the primary charger has failed, that the *switch* must be moved to position *charger-r* and that *charger-r* must be turned on and begin trickle charging the battery. The constraint that at least one charger must always be on if the incoming *bus-voltage* is greater than zero was encoded in the original model, and carried through to the compiled model and the dissents. The following is a sampling of the relevant dissents.

```
29.    [ ]    ->    ¬[    SWITCH.MODE=CHARGER-P,    CHARGER-
R.MODE=FULL-ON ]
30.    [ ]    ->    ¬[    SWITCH.MODE=CHARGER-P,    CHARGER-
R.MODE=TRICKLE ]
31.    [ ]    ->    ¬[ SWITCH.MODE=STUCK-CHARGER-P, CHARGER-
R.MODE=FULL-ON ]
32.    [ ]    ->    ¬[ SWITCH.MODE=STUCK-CHARGER-P, CHARGER-
R.MODE=TRICKLE ]
33.    [ ]    ->    ¬[                CHARGER-P.MODE=FULL-ON,
SWITCH.MODE=STUCK-CHARGER-R ]
34.    [ ]    ->    ¬[                CHARGER-P.MODE=FULL-ON,
SWITCH.MODE=CHARGER-R ]
```

The expected diagnosis of the system with these observations is:

> { SA = operational, DS-P = none-closed, DS-R = none-closed, AS = none-closed, S = CH-R, CH-P = broken, CH-R = trickle, B = full }

The output of the CME engine for this scenario is given below.

These observations and commands result in the following mode estimate.

Figure 8-12 - CME Output for Failed Charger

The most likely mode estimate given above identifies the primary charger as being broken, and has placed the switch at the redundant charger position. The redundant charger is then turned on to the trickle mode and charges the battery. CME correctly estimates the modes of all three components using multiple observations to identify the failed *charger* and to choose the correct modes for the *switch* and *charger-r*.

8.3.4   Digital Shunt Failure

Another critical failure of the power system involves a failure of the digital shunts. In this case, if a shunt fails stuck open, then the resulting current is going to be higher than expected. This failure is similar to the analog shunts. However, in this case, the diagnosis is much more difficult because there is only a single observable for the digital shunt current, Ishunt_D. So,

if the shunt current is higher than expected, the failure could be in either the primary or redundant shunts.

This failure is captured in the NEAR rules under numbers 24 and 25. If the digital shunt current exceeds 6 A, then one of the banks of digital shunts has failed. The NEAR rule set automatically determines whether the primary or redundant charger has failed by executing the recovery actions "Find_Bad_Bus_Reg" and then "Try_Sec_Bus_Reg_Off". By executing "Find_Bad_Bus_Reg", the power system disables the primary digital shunts, and as a result the output of these shunts does not appear in the Ishunt_D output. This leaves the redundant digital shunts enabled, and thus its output is reflected in Ishunt_D. Then, the system waits for another reading of the observation, and if it exceeds 6 A again, then the fault is isolated in the redundant digital shunts. However, if the Ishunt_D was not greater than 6 A on the second reading, the recovery action "Try_Sec_Bus_Reg_Off" is executed which enables the primary digital shunts and disables the redundant digital shunts. If the observation is 6 A this time, then the fault is isolated in the primary digital shunts.

This scenario offers a prime example to demonstrate the utility of CME and its tracking of multiple mode estimates. By tracking multiple mode estimates, CME determines in different mode estimates that either the primary or redundant digital shunts has failed. However, the most likely mode estimate may not be the correct one. This is disambiguated by the second observation though because if the observation persists, then one trajectory becomes highly unlikely, while the other one becomes very likely. The following figure visualizes this.
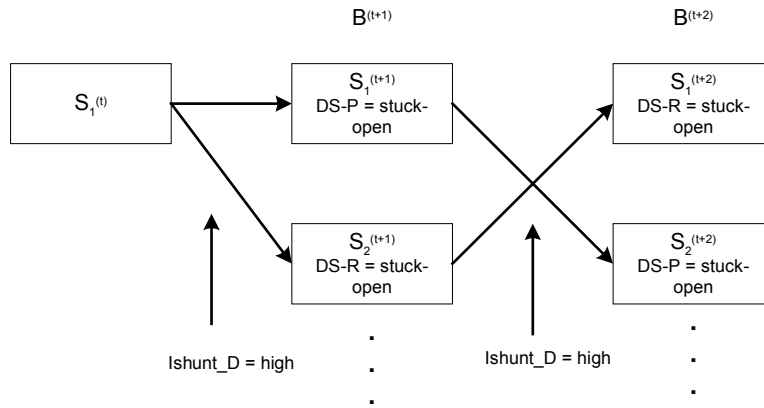
Figure 8-13 - CME Diagnosis of the Digital Shunt Failure

For instance, if the most likely mode estimate contains the component mode *DS-P = stuck-open*, but in reality the mode estimate containing *DS-R = stuck-open* is correct. When the next observation is made that the digital shunt current is still high, then the likelihood that *DS-P = stuck-open* drops considerably, while the likelihood of *DS-R = stuck-open* increases.

In diagnosing the digital shunt failure, the scenario begins with all components in their normal operation, with one digital shunt closed. The initial mode estimate is then:

> { SA = operational, DS-P = one-closed, DS-R = one-closed, AS = none-closed, S = CH-P, CH-P = trickle, CH-R = off, B = full }

The set of observations and commands for the scenario are:

> { Isa = nominal, Ishunt_D = high, Ishunt_PA = nominal, Ichr = trickle, Vbus = nominal, Tbatt = high, Vbatt = nominal, Ibatt = nominal }

> { DS-P-CMD = close, DS-R-CMD = close, AS-CMD = no-command }

The first step of determining the mode estimate for this scenario results in identifying the primary set of digital shunts failing. This mode estimate is followed closely by the mode estimate containing the redundant digital shunts as failing.

> { SA = operational, DS-P = stuck-open, DS-R = one-closed, AS = none-closed, S = CH-P, CH-P = trickle, CH-R = off, B = full }

followed by:

> { SA = operational, DS-P = one-closed, DS-R = stuck-open, AS = none-closed, S = CH-P, CH-P = trickle, CH-R = off, B = full }

The next step in this scenario asserts that the digital shunt current is still high to test if CME actually does isolate the failure to the appropriate bank of digital shunts. This should result in identifying the redundant set of digital shunts as being the source of the failure. The observations and commands input to the system are:

> { Isa = nominal, Ishunt_D = high, Ishunt_PA = nominal, Ichr = trickle, Vbus = nominal, Tbatt = high, Vbatt = nominal, Ibatt = nominal }

> { DS-P-CMD = no-command, DS-R-CMD = no-command, AS-CMD = no-command }

These observations assert that the digital shunts still in operation are causing the fault. The result of these inputs should be to diagnose the other bank of digital shunts as faulty. Depending on which bank of digital shunts failed in the first step above, then the other should be isolated as the failed component. The resulting mode estimate should be:

> { SA = operational, DS-P = one-closed, DS-R = stuck-open, AS = none-closed, S = CH-P, CH-P = trickle, CH-R = off, B = full }

with the following mode estimate being much less likely,

> { SA = operational, DS-P = stuck-open, DS-R = one-closed, AS = none-closed, S = CH-P, CH-P = trickle, CH-R = off, B = full }

The results of this scenario are shown in the following screen shots.

This initial set of observations results in the following mode estimates:

Notice that CME has identified that the primary digital shunts and the redundant digital shunts have failed with high probability, with the primary failing with a slightly higher likelihood. This causes the disabling of the primary digital shunts. However, given the next set of observations:

CME gives the following mode estimates. Notice that the two have changed positions and that the mode estimate containing DS-P = stuck-open

becomes less likely because the observations persist, thus identifying DS-R = stuck-open as the failed component mode. The full belief state returned by CME for this experiment is given in Appendix D.

Figure 8-14 - CME Output for a Failed Digital Shunt

8.3.5   Failed Charger and Failed Analog Shunt

The final scenario developed for the validation of the CME engine involves diagnosing two simultaneous component failures. The failures chosen are the difficult diagnosis of the failed charger, and the diagnosis of an analog shunt. This scenario demonstrates CME's ability to diagnose multiple component failures, in this case, in different parts of a system. The diagnosis of a charger is independent of the diagnosis of an analog shunt, even though they are both in the power system. This scenario uses a combination of observation and command values from the scenarios detailed in sections 8.3.2 and 8.3.3.

The initial mode estimate for the system begins with the components in normal operating modes:

{ SA = operational, DS-P = none-closed, DS-R = none-closed, AS = none-closed, S = CH-P, CH-P = trickle, CH-R = off, B = full }

The commands then given to the system are:

{ DS-P-CMD = no-command, DS-R-CMD = no-command, AS-CMD = close }
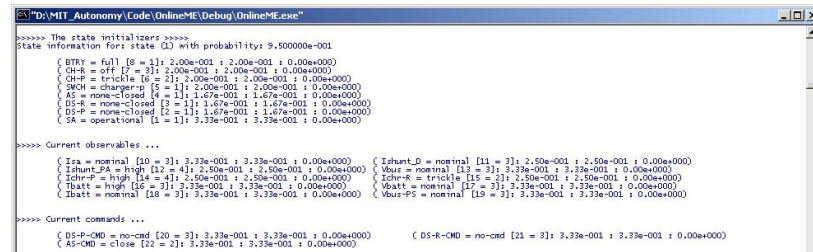
The resulting observations for the scenario:

{ Isa = nominal, Ishunt_D = nominal, Ishunt_PA = high, Ichr = high, Vbus = nominal, Tbatt = high, Vbatt = nominal, Ibatt = nominal }

The resultant diagnosis for this set of commands and observations should be:

{ SA = operational, DS-P = none-closed, DS-R = none-closed, AS = stuck-open, S-CH-R, CH-P = broken, CH-R = trickle, B = full }

The results of the CME engine are given below:

These observations result in the following mode estimate determined by CME:

Figure 8-15 - CME Results on Double Failure with the Analog Shun and Charger

This most likely estimate reflects the desired result of the stuck-open analog shunts and the broken primary charger. The approximate belief state returned by CME is given in Appendix D.

8.4   *Discussion*

The suite of tests in Section 8.3 demonstrates several of the important capabilities of the CME engine. These include diagnosing single and multiple failures using multiple pieces of information, ranking the diagnoses, and the benefits of using a compiled model. These benefits include a smaller memory footprint for the model and the mode estimation engine, diagnoses that are inspectable for correctness before spacecraft operation, and more intuitive modeling of components.

The model of the NEAR Power system used for this experiment before being compiled had a memory footprint of 107 KB as tabulated on a Microsoft Windows 2000 operating system. The compiled model of dissents and transitions requires approximately 30% less memory. Additionally, the CME program has a footprint of only 250 KB, but could easily be reduced with better coding techniques. In comparison, the Livingstone engine has a footprint of 500 KB, in addition to the size of the model. The combination of these two gives a mode estimation capability that takes up little space in systems where it is so precious. This result is encouraging, however more data points need to be collected to verify this.

The experiments above began with three tests of nominal behavior, where these included commanding digital shunts and analog shunts to close, followed by the test using the charger and battery where the battery was discharging. These tests demonstrated that CME could identify nominal behavior of the system accurately and ensure that it identifies normal operations of the components, not just faulty behavior. The tests using the digital and analog shunts demonstrated that CME uses the commands and observations properly to track mode estimates from one time step to the next. The scenario to note in these nominal tests is the operation of the

charger and battery. This demonstrated CME's ability to track nominal behavior correctly of multiple components and their interactions.

The failure scenarios developed and tested highlight the capabilities of CME. The first failure experiments involving the digital and analog shunts required the use of several pieces of information. The commands given to the shunts and the resultant observations of the output currents being higher than expected are used to determine that the *digital-shunts = stuck-open*.

The next scenarios to highlight are the failing of the charger and the failure of the digital shunts. These two experiments are of prime interest because they demonstrate the power of CME. The models for these two scenarios express complex faults simply by developing the model constraints appropriately. In the case of the failed charger, CME is able to not only identify the failed component, but also identify that the switch must change modes to the charger-r mode, and that charger-r must be turned on to trickle charge the battery. This diagnosis is made possible through the simple constraints that at least one charger must be on if the incoming bus-voltage is greater than zero. The dissents shown in Section 8.3.3 present the conflicts that express these same constraints. The conflicts are very intuitive because they are expressed with component mode assignments in proximity to one another making verification of the correctness of conflicts easier for a human. For instance the conflict ¬[SWITCH.MODE=CHARGER-P, CHARGER-R.MODE=TRICKLE] is expressed using switch and charger-p modes, which are two components in sequence. This makes reasoning about the conflict for verification focused. Most of the conflicts for the NEAR Power system have this property, and are given in Appendix D.

The true benefit of the CME engine is demonstrated using the failure of the digital shunts. This experiment showed the benefit of tracking the belief state over time because the proper behavior could not be determined

immediately. In this scenario, CME identified the primary digital shunts as having failed due to the reading of *Ishunt_D = high*. This mode estimate would cause the NEAR system to disable the primary shunts. However, the sensor reading persists in the scenario, indicating that the redundant digital shunts are the true faulty components CME determines this without any problems because it tracked this mode estimate in the previous belief state, and the observation only increases its likelihood, as demonstrated in Section 8.3.4. CME also reduces the likelihood of the previous mode estimate that indicated that the primary digital shunts were faulty. This diagnosis may not have been possible with Livingstone because it only tracked the most likely trajectory of the system. Livingstone would have identified the primary digital shunts as having failed and thrown away the less likely diagnosis involving the redundant digital shunts. When the observation persists into the next time step, Livingstone may not have been able to identify that the redundant digital shunts had failed and that the primary digital shunts are still working. This is the key benefit of CME to identify when less likely trajectories of the system become the most likely mode estimate.

The final experiment presented demonstrates the capability of CME to identify multiple simultaneous component failures. The experiment involved observations that indicate a failed analog shunt and a failed charger. The CME engine correctly diagnoses that these components have failed using the multiple sources of observation information.

These experiments derived from the NEAR rules demonstrate that CME is capable of diagnosing the same failures. Rule 22 and 23 involve the determination that the analog shunts have failed if the current output is high. CME correctly determined that the primary analog shunts have failed in 8.3.2. Rule 24 and 25 relate to a failure of the digital shunts. CME correctly identifies that the redundant digital shunts have failed in 8.3.4.

Rule 26 relates to a failure of the solar arrays, which is discussed in Appendix E. Rule 27 relates to stopping the monitoring of rules 28 and 29 if the redundant battery charger is on. Rules 28 and 29 relates to failures of the primary charger, so if it has failed, indicated by the redundant charger being on, then the rules are no longer useful. Rules 28 and 29 are covered by CME in 8.3.3 where it correctly determined that the charger was failed and that the switch and charger had to change modes. The final rule, Rule 30 relates to the automatic switching of the charger based on the temperature of the battery. If the battery temperature is high, this indicates that the battery is full, so the charger only needs to trickle charge it. This rule is covered by the nominal behavior of the charger and battery discussed in Section 8.3.1.2.

The validation experiments detailed here have demonstrated the various capabilities of the CME engine. CME gives savings in the memory footprint because of the compiled model and the online portion of the algorithm. Additionally, the experiments demonstrate CME's ability to diagnose single, and multiple component failures, as well as the benefits of tracking the belief state instead of most likely trajectories.

9   Conclusions

This thesis has developed an improvement to mode estimation that unifies the rule-based and model-based approaches to fault management. We have developed a system, Compiled Mode Estimation that compiles a system model to a Compiled Concurrent Automata (CMPCA). CMPCA encodes the system model as a set of conflicts, encoded as dissents, and compiled transitions. The CMPCA is used online to determine a set of mode estimates that are consistent with the observations. Compiled Mode Estimation (CME) tracks multiple mode estimates at each time step to increase accuracy of the mode estimate. This enables CME to diagnose a multitude of faults, including multiple component failures, and diagnose complex spacecraft behavior and component interactions. The results of the previous chapter highlight these benefits through the experiments.

9.1   Results

The experiments of the previous chapter involved a suite of nominal and failure scenarios using the NEAR power system. These scenarios were developed from the rules used by the NEAR spacecraft to diagnose failures. The experiments to note are the failures of a charger, a digital shunt, and the combination of a failed charger and an analog shunt. CME was able to diagnose all of the failed components correctly. Each of these scenarios highlights a key capability of CME. The failure of the charger highlights

CME's ability to determine a failed component from multiple sources of information. In this failure scenario, the charger current was high and the remaining observations were all nominal. CME uses this information to determine that no other components are faulty, and identified only the charger. Additionally, CME is able to determine that the switch and redundant charger have changed modes as a result of the failed charger. This highlights the capability of CME to identify the changed behavior of multiple components.

The next scenario of interest is the failure of the digital shunts. This experiment exploited CME's tracking of mode estimates and demonstrated the significant benefit of the approach. The primary digital shunts were first diagnosed as being the most likely fault by CME. However, when the observation persisted in the next time step, CME was able to determine that the failed component was the redundant digital shunts. CME was only able to determine this because it tracked additional mode estimates with the most likely mode estimate.

The final scenario involved the failure of a charger and the primary analog shunts. CME determined the correct mode estimate for this set of observations, demonstrating that it is capable of diagnosing multiple, simultaneous failures in different subsystems. This is a key capability for a mode estimation system to be able to discriminate diagnoses and focus in on the most likely ones. Even though by probability, single faults are more likely than multiple faults, CME was able to determine the correct diagnosis using the conflicts.

These experiments validate the CME engine and the compiled model. The CME engine has demonstrated that the conflicts are indeed sufficient to reconstruct the diagnoses of the system [*deKleer, 1987*]

9.2   *Compiled Mode Estimation*

Recall the initial capabilities list for a mode estimation system for spacecraft:

> "A fault management engine must be capable of detecting single and multiple failures, using multiple sources of information to determine system behavior, and have the ability to rank diagnoses of the system. Additionally, as available resources, including time, computational power and storage space, for fault management on board a spacecraft dwindle it becomes necessary to require faster response times and smaller memory allocation for these software processes."
>
> - Introduction

CME has been developed with this list of capabilities in mind. CME is able to detect single and multiple failures of components by using conflicts in an online process to choose the correct component modes that are consistent with the observations. CME is able to rank the diagnoses it generates using the probabilities of transitions in the system model.   The addition of transitions enables CME to track mode estimates over time as well. CME is able to give real time guarantees when determining mode estimates. The compiled model enables the design of any-time algorithms for the online process of generating mode estimates. By removing satisfiability from the online determination of mode estimates, CME only requires a minimal set covering of the current conflicts to determine mode estimates that are consistent with the observations and the system model. Not only does this enable CME to give real-time performance, it also reduces the memory footprint in the system. The compiled model is a compact encoding of the original mode constraints and transitions. Furthermore, the algorithms for Online-ME are simplified by exploiting the properties of the compiled model, requiring less space for the actual executable.

These benefits of CME are essential for spacecraft as missions continue to push deeper into the solar system. CME has the ability to determine the system behavior accurately and efficiently, which is a necessity for space explorers tasked with venturing further out into the solar system. CME could provide this capability as a standalone in order to give the spacecraft the ability to determine system behavior. Alternatively, CME has a much more powerful use within a larger autonomy architecture, as described in Chapter 1. As a piece in the Model-based Executive, CME enables the spacecraft to be reactive to diagnose and repair failures, reconfigure the spacecraft to achieve goals and be more robust. For space exploration to overcome the hurdles of failures due to spacecraft complexity and tackle difficult missions, enhanced fault management, and possibly larger autonomy systems, will be required. CME is an advancement to enhance the capabilities of fault management through the use of common sense models, but to also allow the spacecraft engineer the ability to inspect the diagnostic results of the engine before the operation of the spacecraft. By unifying the rule-based and model-based approaches to fault management, CME has combined the strengths of an explicit representation of the diagnostic results from rule-based systems, and the benefits of automated reasoning of component interactions from model-based systems to deliver a fault detection engine that spacecraft will need if they are to be successful in future missions.

10 Future Work

The CME mode estimation engine addresses the problems of tracking an approximated belief state over time. However, the engine as developed and implemented here can be made more efficient. This chapter details a few extensions to the algorithms in the online portion of CME that may have an impact on the performance and accuracy of the mode estimates.

10.1   Compiled Conflict Recognition

The most expensive computation in this portion of Online-CME is the triggering of dissents and transitions. This step requires determining those dissents and transitions that are enabled by the observations, commands and component mode assignments in the previous mode estimates. As detailed previously, the algorithm iterates through the dissents and compiled transitions of a changed observation, command or component mode assignment. While this is a standard efficient indexing method, the process could be designed to require fewer computations on average.

A SAT solver, Chaff [*Moskewicz, 2001*] has been developed to perform satisfiability very efficiently. Its approach to solving the expensive cost of Boolean constraint propagation is to monitor only particular literals in a clause, and if the variable becomes false, chooses another literal in the clause to monitor. Once all but one of the values in a clause become false, then the remaining literal is true. This approach has provided an order of

magnitude speed up in finding a solution to the SAT problem. This idea of monitoring particular literals is extended to the Dissent and Transition Triggering to monitor only particular assignments.

By focusing on particular observations or command assignments, this requires only determining if the dissents and compiled transitions associated with these assignments are triggered. This could save many computations by not testing dissents and compiled transitions that would not be enabled. To use this technique, a method must be developed to choose assignments to monitor. This could be based on the probability of an assignment, where least likely assignments are monitored before more likely assignments. This is a good approach because unlikely assignments are hardest to satisfy.

This approach could speed up computations by never looking at dissents or transitions because the assignment focused on is not in the current set of observations, commands or previous component modes.

10.2   *Dynamic Mode Estimate Generation*

The Dynamic Mode Estimate Generation algorithms for CME have been built upon A* search. The CDA* algorithm leverages conflicts to direct the A* search to find the optimal solution. The Generate algorithm uses tree search to choose a previous mode estimate so that CDA* can determine its most likely transition to a current mode estimate. The choice of the previous mode estimate is guided by the heuristics described in Section 6.4.3.1. However, this heuristic does not add as much guidance to the search as desired. The calculation of the residual has a minimal effect on the ordering of the nodes. A different approach to calculating an admissible heuristic is presented in this section.

The previous heuristic used the residual plus the transition probability from a previous mode estimate to put an upper bound on the probability that a previous mode estimate would transition to a current mode estimate. However, this did not effect the ordering of the tree very much. What has not been incorporated into the calculation is the probability of a previous mode estimate's transitions that have not been enumerated. This probability is useful to get a better estimate on the upper bound. To better describe this, consider the following figure:
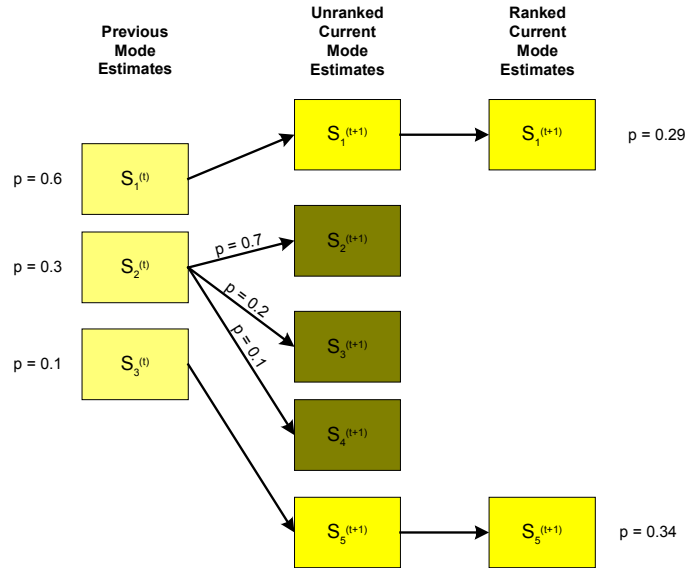
Figure 10-1 - Example Transition System for New Heuristic

Noted on this figure is the previous mode estimates to the left. In the middle of the figure are the current mode estimates that have not been ranked, and to the left are current mode estimates that have been ranked, and their probability is noted to the right. The example focuses on the state transitions from $S_2^{(t)}$ to states two through four at '$t+1$'. Consider that the state $S_2^{(t+1)}$ has been generated and the Generate algorithm must calculate the cost of this node. First, the residual is calculated as $R = 1 - \Sigma\ P(S_r^{(t+1)})$,

where $S_r^{(t+1)}$ are mode estimates that have been ranked by the Rank algorithm. The residual represents the probability of all current mode estimates that have not been enumerated. In the current Generate algorithm, this probability was added to the transition probability of the current mode estimate to arrive at the cost.

To tighten the cost, we use the knowledge that current mode estimates from the same previous mode estimate are distinct from one another. For instance, $S_3^{(t+1)}$ and $S_4^{(t+1)}$ are necessarily distinct from $S_2^{(t+1)}$, but current mode estimates from $S_1^{(t)}$ may not necessarily be distinct from $S_2^{(t+1)}$. The cost defined for the Generate algorithm for $S_2^{(t+1)}$ should not include transitions from the previous mode estimate, $S_2^{(t)}$, to other current mode estimates, $S_3^{(t+1)}$ and $S_4^{(t+1)}$. For example, having generated the current mode estimate $S_2^{(t+1)}$, the residual is $R = 1 - 0.34 - 0.29 = 0.37$. Adding this residual to $P(S_2^{(t+1)}) = 0.7 \times 0.3 = 0.21$, results in 0.58. Notice that this value includes transitions to current mode estimates $S_3^{(t+1)}$ and $S_4^{(t+1)}$. These should not be incorporated in the cost of $S_2^{(t+1)}$, because they are all generated from the same previous mode estimate. Instead, these should be subtracted from the residual to obtain a tighter upper bound on the cost of $S_2^{(t+1)}$.

The key issue is to be able to subtract these probabilities without explicitly enumerating them. Using the example above, the probability of $S_3^{(t+1)}$ is given as $P(S_2^{(t)}) \times P_\mathbf{T}(S_2^{(t)} \rightarrow S_3^{(t+1)}) = 0.3 \times 0.2$. Similarly for $S_4^{(t+1)}$, $P(S_4^{(t+1)}) = 0.3 \times 0.1$. These transition probabilities represent the probability that the previous mode estimate, $S_2^{(t)}$ does not transition to the state $S_2^{(t+1)}$. This is the same as saying that the probability of an element is the same as taking one minus the probability of things that are not that element. So, the probability that $S_2^{(t)}$ does not transition to $S_2^{(t+1)}$ is given as $P(S_2^{(t)}) \times (1 -$

$P_T(S_2^{(t)} \to S_2^{(t+1)})$). This equation results in: $0.3 \times (1 - 0.7) = 0.09$. This is the same values as the sum of $P(S_3^{(t+1)})$ and $P(S_4^{(t+1)})$.

The new residual incorporates the probability of remaining transitions from a previous mode estimate that are not to the current mode estimate. To estimate the cost of the state $S_2^{(t+1)}$ with a tighter upper bound, the transition probability for states $S_3^{(t+1)}$ ($p = 0.2 \times 0.3 = 0.06$) and $S_4^{(t+1)}$ ($p = 0.1 \times 0.3 = 0.03$) are subtracted away. This results in the calculation for a new upper bound cost for $S_2^{(t+1)}$ as:

$$C(S_2^{(t+1)}) \qquad = R - p(S_3^{(t+1)}) - p(S_4^{(t+1)}) + p(S_2^{(t+1)})$$
$$= 0.37 - 0.06 - 0.03 + 0.21 = 0.49.$$

Substituting $P(S_2^{(t)}) \times (1 - P_T(S_2^{(t+1)}))$ for the sum of $P(S_3^{(t+1)})$ and $P(S_4^{(t+1)})$ results in the following for the cost:

$$C(S_2^{(t+1)}) = R - P(S_2^{(t)}) \times (1 - P_T(S_2^{(t)} \to S_2^{(t+1)})) + P(S_2^{(t+1)})$$
$$= 0.37 - 0.3 \times (1 - 0.7) + 0.21 = 0.49$$

This shows that the same value for the cost is determined, but the benefit of the second calculation is that $S_3^{(t+1)}$ and $S_4^{(t+1)}$ did not have to be enumerated.

This demonstrates the computations of the new cost of a current mode estimate in the Generate search tree, but also highlights what it incorporates. The new cost calculation encompasses not just the likelihood of the current mode estimate, but also the probability of the transitions that are not to the current mode estimate. This new computation could enable the Generate algorithm to explore the search tree with a little more guidance.

This page intentionally left blank.

References

[Cadoli, 1997] Cadoli, M. and Donini, F. "A survey on Knowledge Compilation", *The*
  European Journal for Artificial Intelligence, Vol. 10, pp. 137 – 150, 1997.

[Chandra, 2000] Chandra, A. and Simon, L. "Multi-resolution on Compressed Sets of Clauses",
  Proceedings of the International Conference on Tools with Artificial Intelligence, pp.
  449-454, 2000.

[Chung, 2001] Chung, S., Van Eepoel, J. and Williams, B.C., "Improving Model-based Mode
  Estimation through Offline Compilation", *Proceedings of the 6th International*
  Symposium on Artificial Intelligence, Robotics & Automation in Space, 2001.

[deKleer, 1987] deKleer, J and Williams, B.C., "Diagnosing Multiple Faults", *Journal of*
  Artificial Intelligence, Vol. 32, pp. 97 – 130, 1987.

[deKleer 2, 1987] deKleer, J. "An assumption-based truth maintenance system", *Readings in*
  Nonmonotonic Reasoning, Morgan Kaufmann, Los Altos, Ca, 1987.

[deKleer, 1989] deKleer, J and Williams, B.C., "Diagnosis with Behavioral Modes",
  Proceedings of International Joint Conference on Artificial Intelligence, pp 1324-1330,
  1989.

[deKleer, 1992] deKleer, J., Mackworth, A.K., Reiter, R., "Characterizing Diagnoses and
  Systems", *Journal of Artificial Intelligence*, Vol. 56, pp. 197-222, 1992.

[Elliott, 1995] Elliott, R.J., *Hidden Markov Models: Estimation and Control*, Springer, 1995.

[Hamscher, 1992] Hamscher, W, Console, L. and deKleer, J. *Readings in Model-based*
Diagnosis, Morgan Kaufmann, San Mateo, CA, 1992.

[Ingham, 2001] Ingham, M., Ragno, R., Williams, B.C., "A Reactive Model-based Programming
Language for Robotic Space Explorers", *Proceedings of the 6th International Symposium on Artificial Intelligence, Robotics & Automation in Space*, 2001.

[JHUAPL, 2002] Johns Hopkins University Applied Physics Lab, *The MESSENGER Spacecraft*,
http://messenger.jhuapl.edu/, JHU APL, Laurel, MD, 2002.

 [Kurien, 2000] Kurien, J. "Back to the Future for Consistency-based Trajectory Tracking",
Proceedings of the 17th National Conference on Artificial Intelligence, 2000.

[Moskewicz, 2001] Moskewicz, M. et. al., "Chaff: Engineering an Efficient SAT Solver", 39th
Design Automation Conference, Las Vegas, June 2001

[Ragno, 2002] Ragno, R. *Clause Directed A**, MIT M.Eng Thesis in Electrical Engineering and
Computer Science, May 2002.

[Russell, 1995] Russell, P., Norvig, P., *Artificial Intelligence: A Modern Approach*, Prentice
Hall, 1995.

[Wertz, 1999] Wertz, J., *Space Mission Analysis and Design*, Microcosm Books, 3rd ed., 1999.

[Williams, 1996] Williams, B.C., Nayak, P. "A Model-based Approach to Reactive Self-
Configuring Systems", *Proceedings of the 13th National Conference on Artificial Intelligence*, 2000.

[Williams, 1998] Williams, B.C., and Millar, B., "Decompostional Model-based Learning and Its
Analogy to Model-based Diagnosis", *Proceedings of the 15th National Conference on Artificial Intelligence,* 1998.
Achieving Real-time Mode Estimation through Offline Compilation   267

[Williams, 2002] Williams, B.C. and Ragno, R. "Conflict-directed A* and its Role in Model-
  based Embedded Systems", *accepted for publication in Special Issue on Theory and Applications of Satisfiability Testing, Journal of Applied Math*, 2002.

[Williams 2, 2002] Williams, B.C., Ingham, M., Chung, S., Elliott, P., "Model-based
  Programming of Intelligent Embedded Systems and Robotic Space Explorers", *accepted for publication in the IEEE Proceedings Special Issue on Embedded Software,* July 2002.

Appendix A.   NEAR Power System Models

A.1 NEAR Power Generation

A.1.1 Solar Arrays

The solar arrays are the means by which the spacecraft harnesses the energy of the sun and turns it into usable power. The four solar arrays are divided into five solar cell groups, and each solar cell group has its own digital shunt. There is only a single observable for the current produced by the solar array, noted as $\mathbf{I}_{sa}{}^{i}$. The solar array voltage is a fixed value, chosen by the system designers to be 12 volts. The behavior of the solar array that must be captured is when the solar array produces a lower current than expected. Solar arrays are a passive power generation method, meaning that it does not use any mechanical or moving parts to transform the energy of the sun into usable power. The solar arrays merely absorb the energy from the sun, and through a chemical reaction produce current and voltage.

The passive behavior of solar arrays requires that the model capture when the solar array is degraded, thus impacting production of power. Additionally, the solar array may have broken in some way impacting power production. Each of these behaviors manifest themselves in the same way: the current produced is lower than expected. The model is depicted in below.
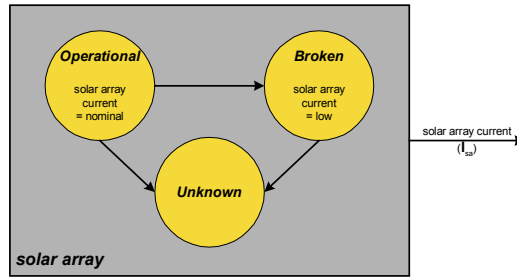
Figure A-1 - Constraint Automaton for the NEAR Power System Solar Arrays

The solar array is expressed using *constraint automata* given in Chapter 2, with discrete modes, constraints on these modes, and constraints between modes. The model shown in Figure A-1 expresses the model of a single solar array. A solar array is modeled using three modes, *operational*, *broken*, and *unknown* modes. The *operational* mode captures the normal behavior of the solar array where the output current, $\mathbf{I}_{sa}^{i}$, is *nominal*. The *broken* mode captures any fault behavior that manifests itself with an output current equal to *low*. These fault behaviors include the degradation of the solar array. Since solar arrays are passive, then the current can never be higher than nominal. Recall that for the charger, the current could increase above any normal value due to a short in the wiring. However, for a solar array, if a short occurs it only acts to reduce the current produced by the solar array. Finally, the *unknown* mode captures any behavior not already modeled. The output of the solar arrays, $\mathbf{I}_{sa}^{i}$, can take on the values {*zero, low, nominal*}. There are no constraints on the transitions in the solar array because they are passive, the sun being the only input required to produce power.

The entire bank of solar arrays is built by duplicating this model four times, one for each solar array on the NEAR spacecraft. The resultant current, $\mathbf{I}_{sa}$, is determined by summing the individual currents from each solar array. This constraint captures the behavior of the overall solar array current since

the solar arrays are connected in parallel. This knowledge is very useful in planning tasks on the spacecraft so that the power required does not exceed the power the solar arrays provide.

A.1.2 Digital Shunts

The digital shunts are a device that removes a solar array from adding to the power in the spacecraft. They are considered to be like a switch, that when open, allow the power from the solar array to be used in the spacecraft. However, if the total power becomes too high for the spacecraft to handle, the digital shunts are commanded to close to short out a solar array. Each solar array has its own bank of digital shunts, as shown in the schematic in Chapter 8. There are five digital shunts associated with each individual solar array. Recall that there are five solar cell groups per solar array, making a single digital shunt connected with a single solar cell group.

The digital shunts are necessary only when the power produced by the solar arrays is too high for the spacecraft to handle. The digital shunts are used to short out, or shunt the power, produced from a solar cell group. This is a method to give coarse adjustments to the power produced from the solar arrays. By shorting out a whole solar cell group, the power generated is significantly reduced. This type of power control is necessary when the spacecraft is near Earth. The solar arrays are designed to produce the necessary power when the spacecraft is orbiting the asteroid, and is further away from the sun than Earth. So, the digital shunts provide a means to reduce the power generated by the solar arrays.

The digital shunts are modeled as a single unit. As shown in the schematic, there is only a single input, the solar array current, $\mathbf{I}_{sa}^{i}$, and single output, the digital shunt current, $\mathbf{I}_{shunt\_D}^{i}$, that give insight into the behavior of the digital shunts. As a result, the digital shunts are modeled as a single unit,

similar to the solar array groups. The constraint automaton for a group of digital shunts is shown below in Figure A-2.
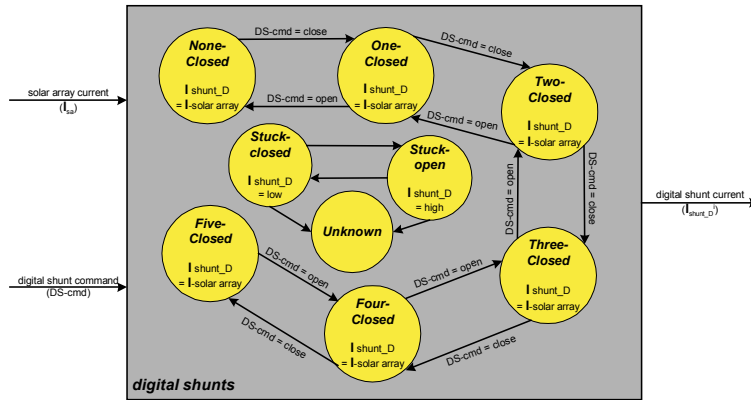
Figure A-2 - Constraint Automaton for the NEAR Power System Digital Shunts

The figure denotes the different modes of the group of digital shunts. The component has modes that denote how many of the digital shunts are closed. In each case, the modeling constraint only states that the output current is *nominal*. This relays the behavior that the shunts are operating normally. The fault modes *stuck-open*, *stuck-closed* and *unknown* are modeled in the component. The *stuck-open* mode denotes that the digital shunts do not close when commanded. Since the digital shunts are modeled as a group, it is impossible to tell if one, two or five are stuck open. This mode is characterized by an output current higher than expected because a shunt commanded to close has remained open, making more power available. The mode *stuck-closed* captures the behavior that a shunt has not closed when commanded. As a result, the output current is lower than expected because the shunt staying closed reduces the available power. The *unknown* mode captures any behavior not considered that does occur.

The transitions between the modes of the digital shunt are conditioned on the input command, $\mu_{shunt\_D}^i$. Note that idle transitions and transitions to the

272   Achieving Real-time Mode Estimation through Offline Compilation

fault modes are not shown for clarity. There are commands associated with the primary and redundant digital shunts. Each group is commanded independently, giving a total of eight input commands to the system. The transitions are designed in a way to restrict the opening and closing of the digital shunts. Only one digital shunt can be opened or closed at any time. The allowable commands for the digital shunt are {*open*, *close*, *no-command*}.

A complication arises because of the single output, $\mathbf{I}_{shunt\_D}^{i}$, for the two groups of digital shunts, the primary and redundant, for each solar array. The individual outputs of the digital shunt groups must be constrained to output a single value. This constraint specifies that if the two outputs, $\mathbf{I}_{shunt\_D\_P}^{i}$ and $\mathbf{I}_{shunt\_D\_R}^{i}$, are the same, that the $\mathbf{I}_{shunt\_D}^{i}$, is this value. When the two values are different, the constraint must define which output to use. Since the output indicates when the digital shunt group has failed in some way, then the output, $\mathbf{I}_{shunt\_D}^{i}$, should indicate this as well. The current can only have values of {*low*, *nominal*, *high*}, so the different combinations are enumerated as follows.

$\qquad$ (**if** ($\mathbf{I}_{shunt\_D\_P}^{i}$ = *nominal*) $\wedge$ ($\mathbf{I}_{shunt\_D\_R}^{i}$ = *nominal*) $\Rightarrow$ ($\mathbf{I}_{shunt\_D}^{i}$ = *nominal*))
$\qquad$ (**if** ($\mathbf{I}_{shunt\_D\_P}^{i}$ = *low*) $\wedge$ ($\mathbf{I}_{shunt\_D\_R}^{i}$ = *low*) $\Rightarrow$ ($\mathbf{I}_{shunt\_D}^{i}$ = *low*))
$\qquad$ (**if** ($\mathbf{I}_{shunt\_D\_P}^{i}$ = *high*) $\wedge$ ($\mathbf{I}_{shunt\_D\_R}^{i}$ = *high*) $\Rightarrow$ ($\mathbf{I}_{shunt\_D}^{i}$ = *high*))
$\qquad$ (**if** ($\mathbf{I}_{shunt\_D\_P}^{i}$ = *nominal*) $\wedge$ ($\mathbf{I}_{shunt\_D\_R}^{i}$ = *low*) $\Rightarrow$ ($\mathbf{I}_{shunt\_D}^{i}$ = *low*))
$\qquad$ (**if** ($\mathbf{I}_{shunt\_D\_P}^{i}$ = *nominal*) $\wedge$ ($\mathbf{I}_{shunt\_D\_R}^{i}$ = *high*) $\Rightarrow$ ($\mathbf{I}_{shunt\_D}^{i}$ = *high*))
$\qquad$ (**if** ($\mathbf{I}_{shunt\_D\_P}^{i}$ = *low*) $\wedge$ ($\mathbf{I}_{shunt\_D\_R}^{i}$ = *nominal*) $\Rightarrow$ ($\mathbf{I}_{shunt\_D}^{i}$ = *low*))
$\qquad$ (**if** ($\mathbf{I}_{shunt\_D\_P}^{i}$ = *low*) $\wedge$ ($\mathbf{I}_{shunt\_D\_R}^{i}$ = *high*) $\Rightarrow$ ($\mathbf{I}_{shunt\_D}^{i}$ = *low*))
$\qquad$ (**if** ($\mathbf{I}_{shunt\_D\_P}^{i}$ = *high*) $\wedge$ ($\mathbf{I}_{shunt\_D\_R}^{i}$ = *nominal*) $\Rightarrow$ ($\mathbf{I}_{shunt\_D}^{i}$ = *high*))
$\qquad$ (**if** ($\mathbf{I}_{shunt\_D\_P}^{i}$ = *high*) $\wedge$ ($\mathbf{I}_{shunt\_D\_R}^{i}$ = *low*) $\Rightarrow$ ($\mathbf{I}_{shunt\_D}^{i}$ = *high*))

For each solar array, there is a primary and redundant set of digital shunts, each with a single output current, $\mathbf{I}_{shunt\_D}^{i}$. These individual outputs are summed to obtain the output, $\mathbf{I}_{shunt\_D}$.

A.1.3 Analog Shunts

The analog shunts are another mechanism of dissipating power generated from the solar arrays. These shunts behave differently than the digital shunts because they dissipate the power through resistors instead of just short-circuiting the solar array. These resistors are shown in the schematic of Chapter 8. The schematic shows that these resistors are connected to switches that enable or disable them. When the switch is closed, it enables the resistor, and allows power to be dissipated. The analog shunts are used to fine-tune the power generated from the solar arrays to the level necessary for the spacecraft. The resistors only dissipate power, so they do not completely remove a group of solar cells.

In each set of analog shunts, there are seven resistors, each with their own switches. It is the mechanics of the switch that determine if an analog shunt is used or not. The inputs to the analog shunts are the overall current from the digital shunts, I_{shunt_D}, and the command, μ_{shunt_P} and μ_{shunt_R}, denoting commands for the primary and redundant analog shunts, respectively. The output from the analog shunts is the current, I_{shunt_P} and I_{shunt_R}, denoting the primary and redundant again. Since there is only a single output, then this leads to modeling the analog shunts as a single entity, similar to the digital shunts. The model is given below in Figure A-3.
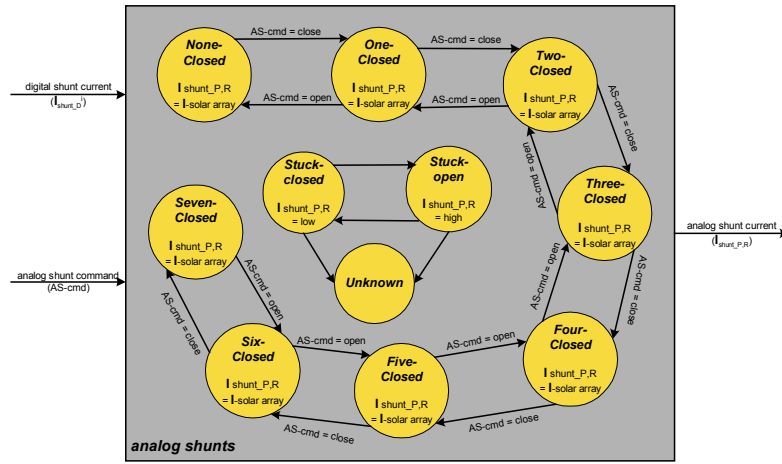
Figure A-3 - Constraint Automaton for the NEAR Power System Analog Shunts

The analog shunts are modeled with modes denoting how many analog shunts are closed. The constraint for these modes denotes that the output current is *nominal*. This denotes that the analog shunts are operating normally. The fault modes used in this model are *stuck-open*, *stuck-closed*, and *unknown*. These modes denote the same behavior as the digital shunts. If the output current is lower than expected, then this means that an analog shunt did not open when it was commanded, so the overall analog shunts are *stuck-closed*. Since there is only a single observable value for the analog shunts, then only the group of analog shunts can be pinpointed as having failed, and not individual shunts.

The component model describes a single group of analog shunts. The NEAR Power system has primary and redundant, each with their own output current, denoted as I_{shunt_P} and I_{shunt_R} respectively. Ideally, as redundant systems, the two groups of analog shunts behave identically. So, when the primary group closes a shunt, the redundant group of shunts does the same. However, the two groups are maintained completely separately

so that if one fails, the other is not adversely affected. For the system model to acquire this behavior, two separate groups of analog shunts are created, each with their own command, μ_{shunt_P} and μ_{shunt_R}, and their own current output.

A.2   NEAR Power Storage

A.2.1   Switch

The first component is a switch that toggles between the primary and redundant chargers in the NEAR Power system. The switch changes between these positions when commanded by an outside source, ordinarily the spacecraft computer. The behavior of the switch is captured using the inputs *bus-voltage* and the command *switch-command*, and the outputs *charger-p-voltage* and *charger-r-voltage*. The constraint automaton for the *switch* is shown below in Figure A-4.
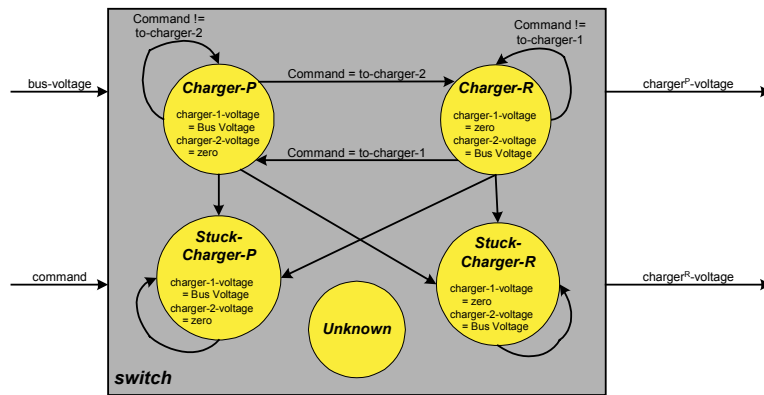
Figure A-4 – Constraint Automaton for the NEAR Power Storage Switch

The behavior of the *switch* is modeled with two operational modes, *charger-p* and *charger-r*, and the fault modes *stuck-charger-p*, *stuck-charger-r*, and *unknown*. The modes constrain the outputs to be particular

276   Achieving Real-time Mode Estimation through Offline Compilation

values. In the case of the mode *charger-p*, the output voltage *charger-p-voltage* is constrained to be equal to the input voltage, and the *charger-r-voltage* is constrained to be zero. This constraint captures the behavior of the *switch* only being able to route the input *bus-voltage* to one charger only. The constraints are similar for the other component modes. The *unknown* mode captures any behavior outside of the specified modes as it has no constraints.

The *switch* transitions between operational modes only. In order to transition from the mode *charger-p* to *charger-r*, the *switch* must receive the input command, *to-charger-r*. Unless the *switch* receives this command, it will remain in the *charger-p* mode. This constraint is expressed similarly between modes *charger-r* and *charger-p*, with the command *to-charger-p*. Under most cases the *switch* remains at the *charger-p* position since it is the primary charger. However, if the primary charger fails, the *switch* automatically changes position to *charger-r*. This behavior is captured using constraints between the primary charger and the *switch-command* that are discussed in the next section.

A.2.2 Charger
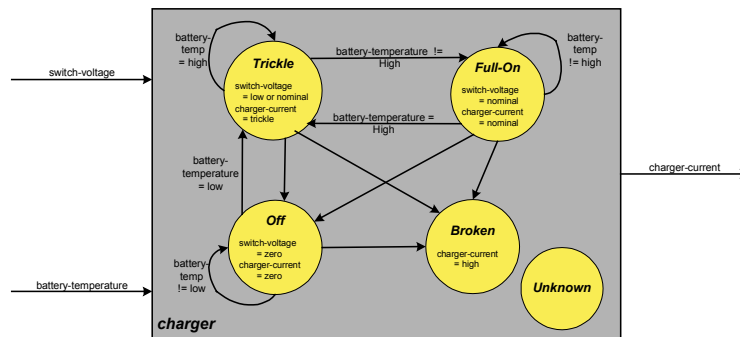
The transitions of the *charger* use the input *battery-temperature* to determine when to change modes. This is consistent with the physical interactions of the charger and battery. The charger would not begin to trickle charge the battery unless the battery was full. The *battery-temperature* allows the charger to determine if the battery is full. As a result, the charger changes between the operational modes *trickle* and *full-on* only when the *battery-temperature* changes to *high*. A *high* battery temperature indicates that the battery charge is full and has heated up due to excess charging. The charger transitions between the modes *off* and *trickle* only if the *battery-temperature* is *low*. This indicates that the battery has been discharging, and the temperature has dropped below the *nominal* level. As a result, the charger begins to charge the battery with a trickle charge, not a full charge. If the battery has been discharging, then the voltage is not high enough for the charger to give a full charge to the battery. Instead the charger trickle charges the battery until the *switch-voltage* increases enough to allow a full charge from the charger. This behavior is captured in the transitions from the *off* mode to the *trickle* mode, and then from the *trickle* mode to the *full-on* mode. This cascading captures the physical behavior of the power system with the battery and charger interactions.

To fully characterize the NEAR Power storage system and the chargers, there must be constraints between the outputs of the two chargers that give a single output, *charger-current*. There are two chargers in the system, a primary and a redundant charger, each with associated output currents, *charger-currentP* and *charger-currentR*. The resultant *charger-current* should take on the higher value of these two as that indicates the true *charger-current* from the one that is on and working. However, should both output charger currents be zero, this only indicates that both are off. The constraints that express this behavior are as follows.

    (**if** (*charger-current^P* = *nominal*) $\land$ (*charger-current^R* = *zero*) $\Rightarrow$ (*charger-current* = *nominal*)

$$(\textbf{if}\ (charger\text{-}current^P = trickle) \wedge (charger\text{-}current^R = zero) \Rightarrow$$
$(charger\text{-}current = trickle)$

$(\textbf{if}\ (charger\text{-}current^P = high) \wedge (charger\text{-}current^R = zero) \Rightarrow (charger\text{-}current = high)$

$(\textbf{if}\ (charger\text{-}current^P = zero) \wedge (charger\text{-}current^R = nominal) \Rightarrow$
$(charger\text{-}current = nominal)$

$(\textbf{if}\ (charger\text{-}current^P = zero) \wedge (charger\text{-}current^R = trickle) \Rightarrow$
$(charger\text{-}current = trickle)$

$(\textbf{if}\ (charger\text{-}current^P = zero) \wedge (charger\text{-}current^R = high) \Rightarrow (charger\text{-}current = high)$

$(\textbf{if}\ (charger\text{-}current^P = zero) \wedge (charger\text{-}current^R = zero) \Rightarrow (charger\text{-}current = zero)$

These constraints capture the behavior of the interactions between the two chargers and the input to the battery, the *charger-current*.

A.2.3 Battery
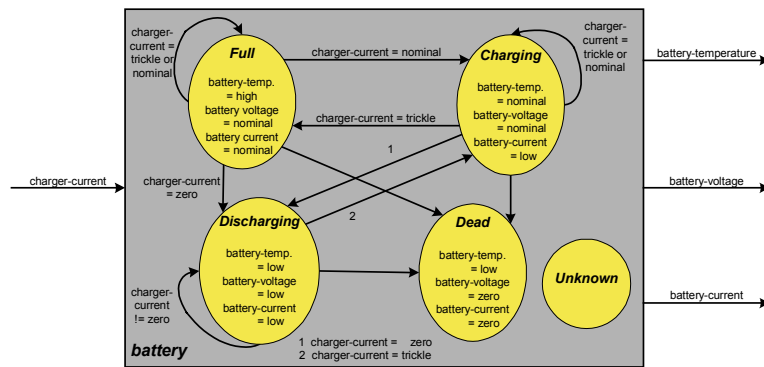
This page intentionally left blank.

Appendix B. NEAR Power Storage Dissents & Transitions

B.1 Dissents

[ ] $\Rightarrow$![SWITCH = STUCK-CHARGER-2 $\wedge$ CHARGER-1 = FULL-ON]

[ ] $\Rightarrow$![SWITCH = STUCK-CHARGER-2 $\wedge$ CHARGER-1 = TRICKLE]

[ ] $\Rightarrow$![SWITCH = STUCK-CHARGER-1 $\wedge$ CHARGER-2 = FULL-ON]

[ ] $\Rightarrow$![SWITCH = STUCK-CHARGER-1 $\wedge$ CHARGER-2 = TRICKLE]

[ ] $\Rightarrow$![SWITCH = CHARGER-2 $\wedge$ CHARGER-1 = FULL-ON]

[ ] $\Rightarrow$![SWITCH = CHARGER-2 $\wedge$ CHARGER-1 = TRICKLE]

[ ] $\Rightarrow$![SWITCH = CHARGER-1 $\wedge$ CHARGER-2 = FULL-ON]

[ ] $\Rightarrow$![SWITCH = CHARGER-1 $\wedge$ CHARGER-2 = TRICKLE]

[BATTERY-TEMPERATURE = LOW] $\Rightarrow$![BATTERY = CHARGING]

[BATTERY-TEMPERATURE = HIGH] $\Rightarrow$![BATTERY = CHARGING]

[BATTERY-VOLTAGE = ZERO] $\Rightarrow$![BATTERY = CHARGING]

[BATTERY-VOLTAGE = LOW] $\Rightarrow$![BATTERY = CHARGING]

[BATTERY-TEMPERATURE = LOW] $\Rightarrow$![BATTERY = FULL]

[BATTERY-TEMPERATURE = NOMINAL] $\Rightarrow$![BATTERY = FULL]

[BATTERY-VOLTAGE = ZERO] $\Rightarrow$![BATTERY = FULL]

[BATTERY-VOLTAGE = LOW] $\Rightarrow$![BATTERY = FULL]

[BATTERY-TEMPERATURE = NOMINAL] $\Rightarrow$![BATTERY = DISCHARGING]

[BATTERY-TEMPERATURE = HIGH] $\Rightarrow$![BATTERY = DISCHARGING]

[BATTERY-VOLTAGE = ZERO] $\Rightarrow$![BATTERY = DISCHARGING]

[BATTERY-VOLTAGE = NOMINAL] $\Rightarrow$![BATTERY = DISCHARGING]

[BATTERY-TEMPERATURE = NOMINAL] $\Rightarrow$![BATTERY = DEAD]

[BATTERY-TEMPERATURE = HIGH] $\Rightarrow$![BATTERY = DEAD]

[BATTERY-VOLTAGE = LOW] $\Rightarrow$![BATTERY = DEAD]

[BATTERY-VOLTAGE = NOMINAL] $\Rightarrow$![BATTERY = DEAD]

[BUS-VOLTAGE = ZERO] $\Rightarrow$![SWITCH = CHARGER-1 $\wedge$ CHARGER-1 = TRICKLE]

[BUS-VOLTAGE = NOMINAL] $\Rightarrow$![SWITCH = CHARGER-1 $\wedge$ CHARGER-1 = TRICKLE]

[BUS-VOLTAGE = ZERO] $\Rightarrow$![SWITCH = CHARGER-1 $\wedge$ CHARGER-1 = FULL-ON]

[BUS-VOLTAGE = LOW] ⟹ ![SWITCH = CHARGER-1 ∧ CHARGER-1 = FULL-ON]

[BUS-VOLTAGE = LOW] ⟹ ![SWITCH = CHARGER-1 ∧ CHARGER-1 = OFF]

[BUS-VOLTAGE = NOMINAL] ⟹ ![SWITCH = CHARGER-1 ∧ CHARGER-1 = OFF]

[BUS-VOLTAGE = ZERO] ⟹ ![SWITCH = CHARGER-2 ∧ CHARGER-2 = TRICKLE]

[BUS-VOLTAGE = NOMINAL] ⟹ ![SWITCH = CHARGER-2 ∧ CHARGER-2 = TRICKLE]

[BUS-VOLTAGE = ZERO] ⟹ ![SWITCH = CHARGER-2 ∧ CHARGER-2 = FULL-ON]

[BUS-VOLTAGE = LOW] ⟹ ![SWITCH = CHARGER-2 ∧ CHARGER-2 = FULL-ON]

[BUS-VOLTAGE = LOW] ⟹ ![SWITCH = CHARGER-2 ∧ CHARGER-2 = OFF]

[BUS-VOLTAGE = NOMINAL] ⟹ ![SWITCH = CHARGER-2 ∧ CHARGER-2 = OFF]

[BUS-VOLTAGE = ZERO] ⟹ ![SWITCH = STUCK-CHARGER-1 ∧ CHARGER-1 = TRICKLE]

[BUS-VOLTAGE = NOMINAL] ⟹ ![SWITCH = STUCK-CHARGER-1 ∧ CHARGER-1 = TRICKLE]

[BUS-VOLTAGE = ZERO] ⟹ ![SWITCH = STUCK-CHARGER-1 ∧ CHARGER-1 = FULL-ON]

[BUS-VOLTAGE = LOW] ⟹ ![SWITCH = STUCK-CHARGER-1 ∧ CHARGER-1 = FULL-ON]

[BUS-VOLTAGE = LOW] ⟹ ![SWITCH = STUCK-CHARGER-1 ∧ CHARGER-1 = OFF]

[BUS-VOLTAGE = NOMINAL] ⟹ ![SWITCH = STUCK-CHARGER-1 ∧ CHARGER-1 = OFF]

[BUS-VOLTAGE = ZERO] ⟹ ![SWITCH = STUCK-CHARGER-2 ∧ CHARGER-2 = TRICKLE]

[BUS-VOLTAGE = NOMINAL] ⟹ ![SWITCH = STUCK-CHARGER-2 ∧ CHARGER-2 = TRICKLE]

[BUS-VOLTAGE = ZERO] ⟹ ![SWITCH = STUCK-CHARGER-2 ∧ CHARGER-2 = FULL-ON]

[BUS-VOLTAGE = LOW] ⟹ ![SWITCH = STUCK-CHARGER-2 ∧ CHARGER-2 = FULL-ON]

[BUS-VOLTAGE = LOW] ⟹ ![SWITCH = STUCK-CHARGER-2 ∧ CHARGER-2 = OFF]

[BUS-VOLTAGE = NOMINAL] ⟹ ![SWITCH = STUCK-CHARGER-2 ∧ CHARGER-2 = OFF]

B.2 Transitions

B.2.1 Charger Switch

```
(TRANSITION SWITCH
        FROM-VALUE CHARGER-1
        TO-VALUE CHARGER-2
        GUARD (AND (CHARGER-1 = BROKEN))
        PROBABILITY 0.9899)
(TRANSITION SWITCH
     FROM-VALUE CHARGER-2
     TO-VALUE CHARGER-2
     GUARD NIL
     PROBABILITY 0.9899)
(TRANSITION SWITCH
     FROM-VALUE CHARGER-1
     TO-VALUE CHARGER-1
     GUARD (NOT (CHARGER-1 = BROKEN))
     PROBABILITY 0.9899)
(TRANSITION SWITCH
     FROM-VALUE CHARGER-1
     TO-VALUE STUCK-CHARGER-1
     GUARD NIL
     PROBABILITY 0.01)
(TRANSITION SWITCH
     FROM-VALUE CHARGER-1
     TO-VALUE STUCK-CHARGER-2
     GUARD NIL
     PROBABILITY 0.01)
(TRANSITION SWITCH
     FROM-VALUE CHARGER-2
     TO-VALUE STUCK-CHARGER-1
     GUARD NIL
     PROBABILITY 0.01)
(TRANSITION SWITCH
     FROM-VALUE CHARGER-2
     TO-VALUE STUCK-CHARGER-2
     GUARD NIL
     PROBABILITY 0.01)
(TRANSITION SWITCH
     FROM-VALUE STUCK-CHARGER-1
     TO-VALUE STUCK-CHARGER-1
     GUARD NIL
     PROBABILITY 0.99)
(TRANSITION SWITCH
     FROM-VALUE STUCK-CHARGER-2
     TO-VALUE STUCK-CHARGER-2
     GUARD NIL
     PROBABILITY 0.99)
(TRANSITION SWITCH
```

```
     FROM-VALUE CHARGER-1
     TO-VALUE UNKNOWN
     GUARD NIL
     PROBABILITY 0.0001)
(TRANSITION SWITCH
     FROM-VALUE CHARGER-2
     TO-VALUE UNKNOWN
     GUARD NIL
     PROBABILITY 0.0001)
(TRANSITION SWITCH
     FROM-VALUE STUCK-CHARGER-1
     TO-VALUE UNKNOWN
     GUARD NIL
     PROBABILITY 0.0001)
(TRANSITION SWITCH
     FROM-VALUE STUCK-CHARGER-2
     TO-VALUE UNKNOWN
     GUARD NIL
     PROBABILITY 0.0001)
(TRANSITION SWITCH
     FROM-VALUE UNKNOWN
     TO-VALUE UNKNOWN
     GUARD NIL
     PROBABILITY 1)
```

B.2.2 Charger-1

```
(TRANSITION CHARGER-1
     FROM-VALUE FULL-ON
     TO-VALUE FULL-ON
     GUARD (NOT (BATTERY-TEMPERATURE = HIGH))
     PROBABILITY 0.8899)
(TRANSITION CHARGER-1
     FROM-VALUE FULL-ON
     TO-VALUE TRICKLE
     GUARD (BATTERY-TEMPERATURE = HIGH)
     PROBABILITY 0.8899)
(TRANSITION CHARGER-1
     FROM-VALUE FULL-ON
     TO-VALUE OFF
     GUARD NIL
     PROBABILITY 0.1)
(TRANSITION CHARGER-1
     FROM-VALUE FULL-ON
     TO-VALUE BROKEN
     GUARD NIL
     PROBABILITY 0.01)
(TRANSITION CHARGER-1
     FROM-VALUE FULL-ON
     TO-VALUE UNKNOWN
     GUARD NIL
     PROBABILITY 0.0001)

(TRANSITION CHARGER-1
     FROM-VALUE TRICKLE
     TO-VALUE TRICKLE
     GUARD (BATTERY-TEMPERATURE = HIGH)
     PROBABILITY 0.8899)
(TRANSITION CHARGER-1
```

```
                    FROM-VALUE TRICKLE
                    TO-VALUE FULL-ON
                    GUARD (NOT (BATTERY-TEMPERATURE = HIGH))
                    PROBABILITY 0.8899)
(TRANSITION CHARGER-1
                    FROM-VALUE TRICKLE
                    TO-VALUE OFF
                    GUARD NIL
                    PROBABILITY 0.1)
(TRANSITION CHARGER-1
                    FROM-VALUE TRICKLE
                    TO-VALUE BROKEN
                    GUARD NIL
                    PROBABILITY 0.01)
(TRANSITION CHARGER-1
                    FROM-VALUE TRICKLE
                    TO-VALUE UNKNOWN
                    GUARD NIL
                    PROBABILITY 0.0001)

(TRANSITION CHARGER-1
                    FROM-VALUE OFF
                    TO-VALUE OFF
                    GUARD NIL
                    PROBABILITY 0.1)
(TRANSITION CHARGER-1
                    FROM-VALUE OFF
                    TO-VALUE TRICKLE
                    GUARD NIL
                    PROBABILITY 0.8899)
(TRANSITION CHARGER-1
                    FROM-VALUE OFF
                    TO-VALUE BROKEN
                    GUARD NIL
                    PROBABILITY 0.01)
(TRANSITION CHARGER-1
                    FROM-VALUE OFF
                    TO-VALUE UNKNOWN
                    GUARD NIL
                    PROBABILITY 0.0001)
(TRANSITION CHARGER-1
                    FROM-VALUE BROKEN
                    TO-VALUE BROKEN
                    GUARD NIL
                    PROBABILITY 0.99)
(TRANSITION CHARGER-1
                    FROM-VALUE BROKEN
                    TO-VALUE UNKNOWN
                    GUARD NIL
                    PROBABILITY 0.0001)

(TRANSITION CHARGER-1
                    FROM-VALUE UNKNOWN
                    TO-VALUE UNKNOWN
                    GUARD NIL
                    PROBABILITY 1)
```

Achieving Real-time Mode Estimation through Offline Compilation   285

B.2.3 Charger-2

```
(TRANSITION CHARGER-2
      FROM-VALUE FULL-ON
      TO-VALUE FULL-ON
      GUARD (NOT (BATTERY-TEMPERATURE = HIGH))
      PROBABILITY 0.8899)
(TRANSITION CHARGER-2
      FROM-VALUE FULL-ON
      TO-VALUE TRICKLE
      GUARD (BATTERY-TEMPERATURE = HIGH)
      PROBABILITY 0.8899)
(TRANSITION CHARGER-2
      FROM-VALUE FULL-ON
      TO-VALUE OFF
      GUARD NIL
      PROBABILITY 0.1)
(TRANSITION CHARGER-2
      FROM-VALUE FULL-ON
      TO-VALUE BROKEN
      GUARD NIL
      PROBABILITY 0.01)
(TRANSITION CHARGER-2
      FROM-VALUE FULL-ON
      TO-VALUE UNKNOWN
      GUARD NIL
      PROBABILITY 0.0001)

(TRANSITION CHARGER-2
      FROM-VALUE TRICKLE
      TO-VALUE TRICKLE
      GUARD (BATTERY-TEMPERATURE = HIGH)
      PROBABILITY 0.8899)
(TRANSITION CHARGER-2
      FROM-VALUE TRICKLE
      TO-VALUE FULL-ON
      GUARD (NOT (BATTERY-TEMPERATURE = HIGH))
      PROBABILITY 0.8899)
(TRANSITION CHARGER-2
      FROM-VALUE TRICKLE
      TO-VALUE OFF
      GUARD NIL
      PROBABILITY 0.1)
(TRANSITION CHARGER-2
      FROM-VALUE TRICKLE
      TO-VALUE BROKEN
      GUARD NIL
      PROBABILITY 0.01)
(TRANSITION CHARGER-2
      FROM-VALUE TRICKLE
      TO-VALUE UNKNOWN
      GUARD NIL
      PROBABILITY 0.0001)

(TRANSITION CHARGER-2
      FROM-VALUE OFF
      TO-VALUE OFF
      GUARD NIL
```

```
              PROBABILITY 0.1)
(TRANSITION CHARGER-2
     FROM-VALUE OFF
     TO-VALUE TRICKLE
     GUARD NIL
     PROBABILITY 0.8899)
(TRANSITION CHARGER-2
     FROM-VALUE OFF
     TO-VALUE BROKEN
     GUARD NIL
     PROBABILITY 0.01)
(TRANSITION CHARGER-2
     FROM-VALUE OFF
     TO-VALUE UNKNOWN
     GUARD NIL
     PROBABILITY 0.0001)
(TRANSITION CHARGER-2
     FROM-VALUE BROKEN
     TO-VALUE BROKEN
     GUARD NIL
     PROBABILITY 0.99)
(TRANSITION CHARGER-2
     FROM-VALUE BROKEN
     TO-VALUE UNKNOWN
     GUARD NIL
     PROBABILITY 0.0001)

(TRANSITION CHARGER-2
     FROM-VALUE UNKNOWN
     TO-VALUE UNKNOWN
     GUARD NIL
     PROBABILITY 1)
```

B.2.4 Battery

```
(TRANSITION BATTERY
     FROM-VALUE FULL
     TO-VALUE FULL
     GUARD (AND (NOT (CHARGER-1 = TRICKLE))
                    (NOT (CHARGER-1 = OFF)) )
     PROBABILITY 0.99)
(TRANSITION BATTERY
     FROM-VALUE FULL
     TO-VALUE FULL
     GUARD (AND (NOT (CHARGER-2  = TRICKLE))
                    (NOT (CHARGER-2  = OFF)) )
     PROBABILITY 0.99)
(TRANSITION BATTERY
     FROM-VALUE FULL
     TO-VALUE CHARGING
     GUARD (AND (CHARGER-1 = TRICKLE))
     PROBABILITY 0.99)
(TRANSITION BATTERY
     FROM-VALUE FULL
     TO-VALUE CHARGING
     GUARD (AND (CHARGER-2  = TRICKLE))
     PROBABILITY 0.99)
(TRANSITION BATTERY
     FROM-VALUE FULL
```

Achieving Real-time Mode Estimation through Offline Compilation   287

```
                    TO-VALUE DISCHARGING
                    GUARD (AND (CHARGER-1 = OFF))
                    PROBABILITY 0.99)
(TRANSITION BATTERY
        FROM-VALUE FULL
        TO-VALUE DISCHARGING
        GUARD (AND (CHARGER-2  = OFF))
        PROBABILITY 0.99)
(TRANSITION BATTERY
        FROM-VALUE FULL
        TO-VALUE DEAD
        GUARD NIL
        PROBABILITY 0.001)
(TRANSITION BATTERY
        FROM-VALUE FULL
        TO-VALUE UNKNOWN
        GUARD NIL
        PROBABILITY 0.0001)


(TRANSITION BATTERY
        FROM-VALUE CHARGING
        TO-VALUE CHARGING
        GUARD (AND (NOT (CHARGER-1 = FULL-ON))
                   (NOT (CHARGER-1 = OFF)) )
        PROBABILITY 0.99)
(TRANSITION BATTERY
        FROM-VALUE CHARGING
        TO-VALUE CHARGING
        GUARD (AND (NOT (CHARGER-2  = FULL-ON))
                   (NOT (CHARGER-2  = OFF)) )
        PROBABILITY 0.99)
(TRANSITION BATTERY
        FROM-VALUE CHARGING
        TO-VALUE FULL
        GUARD (AND (CHARGER-1 = FULL-ON))
        PROBABILITY 0.99)
(TRANSITION BATTERY
        FROM-VALUE CHARGING
        TO-VALUE FULL
        GUARD (AND (CHARGER-2  = FULL-ON))
        PROBABILITY 0.99)
(TRANSITION BATTERY
        FROM-VALUE CHARGING
        TO-VALUE DISCHARGING
        GUARD (AND (CHARGER-1 = OFF))
        PROBABILITY 0.99)
(TRANSITION BATTERY
        FROM-VALUE CHARGING
        TO-VALUE DISCHARGING
        GUARD (AND (CHARGER-2  = OFF))
        PROBABILITY 0.99)
(TRANSITION BATTERY
        FROM-VALUE CHARGING
        TO-VALUE DEAD
        GUARD NIL
        PROBABILITY 0.001)
(TRANSITION BATTERY
        FROM-VALUE CHARGING
```

```
      TO-VALUE UNKNOWN
      GUARD NIL
      PROBABILITY 10.0e-7)

(TRANSITION BATTERY
      FROM-VALUE DISCHARGING
      TO-VALUE DISCHARGING
      GUARD (AND (NOT (CHARGER-1 = TRICKLE)) )
      PROBABILITY 0.99)
(TRANSITION BATTERY
      FROM-VALUE DISCHARGING
      TO-VALUE DISCHARGING
      GUARD (AND (NOT (CHARGER-2  = TRICKLE)) )
      PROBABILITY 0.99)
 (TRANSITION BATTERY
      FROM-VALUE DISCHARGING
      TO-VALUE CHARGING
      GUARD (AND (CHARGER-1 = TRICKLE))
      PROBABILITY 0.99)
(TRANSITION BATTERY
      FROM-VALUE DISCHARGING
      TO-VALUE CHARGING
      GUARD (AND (CHARGER-2  = TRICKLE))
      PROBABILITY 0.99)
(TRANSITION BATTERY
      FROM-VALUE DISCHARGING
      TO-VALUE DEAD
      GUARD NIL
      PROBABILITY 0.001)
(TRANSITION BATTERY
      FROM-VALUE DISCHARGING
      TO-VALUE UNKNOWN
      GUARD NIL
      PROBABILITY 10.0e-7)

(TRANSITION BATTERY
      FROM-VALUE DEAD
      TO-VALUE DEAD
      GUARD NIL
      PROBABILITY 0.99)
(TRANSITION BATTERY
      FROM-VALUE DEAD
      TO-VALUE UNKNOWN
      GUARD NIL
      PROBABILITY 10.0e-7)

(TRANSITION BATTERY
      FROM-VALUE UNKNOWN
      TO-VALUE UNKNOWN
      GUARD NIL
      PROBABILITY 1)
```

Achieving Real-time Mode Estimation through Offline Compilation   289

This page intentionally left blank.

Appendix C.   Online-ME Detailed Example

This appendix demonstrates the steps of the Online Mode Estimation algorithms through an example. It goes through in gross detail each step and calculation using the NEAR Power Storage System. The architecture of the system is shown below.
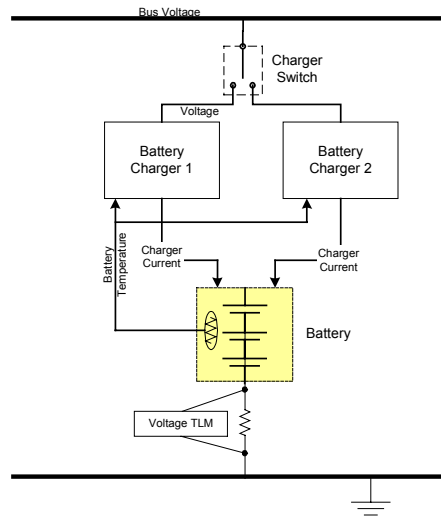
Figure C-1 - NEAR Power Storage System

C.1 *Observations and Initial Mode Estimate*

The initial state for this example is:

   (*switch = charger-1*), (*charger-1 = Full-On*), (*charger-2 = Off*), (*battery = charging*)

The observations for this example are as follows:

   (*bus-voltage = nominal*), (*battery-voltage = nominal*), (*battery-temperature = nominal*)

C.2 Dissents and Transitions

Using the dissents and transitions from Appendix B, and the observations and initial state above, the following dissents and transitions are triggered for this example.

C.2.1 Enabled Dissents

4. [ ] $\Rightarrow$![SWITCH = STUCK-CHARGER-2 $\wedge$ CHARGER-1 = FULL-ON]

5. [ ] $\Rightarrow$![SWITCH = STUCK-CHARGER-2 $\wedge$ CHARGER-1 = TRICKLE]

6. [ ] $\Rightarrow$![SWITCH = STUCK-CHARGER-1 $\wedge$ CHARGER-2 = FULL-ON]

7. [ ] $\Rightarrow$![SWITCH = STUCK-CHARGER-1 $\wedge$ CHARGER-2 = TRICKLE]

8. [ ] $\Rightarrow$![SWITCH = CHARGER-2 $\wedge$ CHARGER-1 = FULL-ON]

9. [ ] $\Rightarrow$![SWITCH = CHARGER-2 $\wedge$ CHARGER-1 = TRICKLE]

10. [ ] $\Rightarrow$![SWITCH = CHARGER-1 $\wedge$ CHARGER-2 = FULL-ON]

11. [ ] $\Rightarrow$![SWITCH = CHARGER-1 $\wedge$ CHARGER-2 = TRICKLE]

12. [BATTERY-TEMPERATURE = NOMINAL] $\Rightarrow$![BATTERY = FULL]

13. [BATTERY-TEMPERATURE = NOMINAL] $\Rightarrow$![BATTERY = DISCHARGING]

14. [BATTERY-VOLTAGE = NOMINAL] $\Rightarrow$![BATTERY = DISCHARGING]

15. [BATTERY-TEMPERATURE = NOMINAL] $\Rightarrow$![BATTERY = DEAD]

16. [BATTERY-VOLTAGE = NOMINAL] $\Rightarrow$![BATTERY = DEAD]

17. [BUS-VOLTAGE = NOMINAL] $\Rightarrow$![SWITCH = CHARGER-1 $\wedge$ CHARGER-1 = TRICKLE]

18. [BUS-VOLTAGE = NOMINAL] $\Rightarrow$![SWITCH = CHARGER-1 $\wedge$ CHARGER-1 = OFF]

19. [BUS-VOLTAGE = NOMINAL] $\Rightarrow$![SWITCH = CHARGER-2 $\wedge$ CHARGER-2 = TRICKLE]

20. [BUS-VOLTAGE = NOMINAL] $\Rightarrow$![SWITCH = CHARGER-2 $\wedge$ CHARGER-2 = OFF]

21. [BUS-VOLTAGE = NOMINAL] $\Rightarrow$![SWITCH =STUCK-CHARGER-1 $\wedge$ CHARGER-1 = TRICKLE]

22. [BUS-VOLTAGE = NOMINAL] $\Rightarrow$![SWITCH = STUCK-CHARGER-1 $\wedge$ CHARGER-1 = OFF]

23. [BUS-VOLTAGE = NOMINAL] $\Rightarrow$![SWITCH =STUCK-CHARGER-2 $\wedge$ CHARGER-2 = TRICKLE]

24. [BUS-VOLTAGE = NOMINAL] $\Rightarrow$![SWITCH = STUCK-CHARGER-2 $\wedge$ CHARGER-2 = OFF]

C.2.2 **Enabled Transitions**

```
(TRANSITION SWITCH
     FROM-VALUE CHARGER-1
     TO-VALUE CHARGER-1
     GUARD (NOT (CHARGER-1 = BROKEN))
     PROBABILITY 0.9899)
(TRANSITION SWITCH
     FROM-VALUE CHARGER-1
     TO-VALUE STUCK-CHARGER-1
     GUARD NIL
     PROBABILITY 0.01)
(TRANSITION SWITCH
     FROM-VALUE CHARGER-1
     TO-VALUE STUCK-CHARGER-2
     GUARD NIL
    PROBABILITY 0.01)
(TRANSITION SWITCH
     FROM-VALUE CHARGER-1
     TO-VALUE UNKNOWN
     GUARD NIL
     PROBABILITY 10E-6)

(TRANSITION CHARGER-1
     FROM-VALUE FULL-ON
     TO-VALUE FULL-ON
     GUARD (NOT (BATTERY-TEMP = HIGH))
     PROBABILITY 0.89)
(TRANSITION CHARGER-1
     FROM-VALUE FULL-ON
     TO-VALUE OFF
     GUARD NIL
     PROBABILITY 0.1)
(TRANSITION CHARGER-1
     FROM-VALUE FULL-ON
     TO-VALUE BROKEN
     GUARD NIL
     PROBABILITY 0.01)
(TRANSITION CHARGER-1
     FROM-VALUE FULL-ON
     TO-VALUE UNKNOWN
     GUARD NIL
     PROBABILITY 10.0e-7)

(TRANSITION CHARGER-2
     FROM-VALUE OFF
     TO-VALUE OFF
     GUARD NIL
     PROBABILITY 0.1)
(TRANSITION CHARGER-2
     FROM-VALUE OFF
     TO-VALUE TRICKLE
     GUARD NIL
     PROBABILITY 0.89)
(TRANSITION CHARGER-2
     FROM-VALUE OFF
     TO-VALUE BROKEN
     GUARD NIL
```

```
                     PROBABILITY 0.01)
(TRANSITION CHARGER-2
     FROM-VALUE OFF
     TO-VALUE UNKNOWN
     GUARD NIL
     PROBABILITY 10.0e-7)

(TRANSITION BATTERY
     FROM-VALUE CHARGING
     TO-VALUE FULL
     GUARD (AND (CHARGER-1 = FULL-ON))
     PROBABILITY 0.499)
(TRANSITION BATTERY
     FROM-VALUE CHARGING
     TO-VALUE CHARGING
     GUARD (AND (NOT (CHARGER-1 = OFF)) )
     PROBABILITY 0.499)
(TRANSITION BATTERY
     FROM-VALUE CHARGING
     TO-VALUE DEAD
     GUARD NIL
     PROBABILITY 0.001)
(TRANSITION BATTERY
     FROM-VALUE CHARGING
     TO-VALUE UNKNOWN
     GUARD NIL
     PROBABILITY 0.0001)
```

C.3 *Constituent Diagnoses*

4. [SWITCH=CHARGER-1 ∨ SWITCH=CHARGER-2 ∨ CHARGER-1=TRICKLE ∨

   CHARGER-1=OFF ∨

   SWITCH=STUCK-CHARGER-1 ∨ CHARGER-1=BROKEN ∨ SWITCH=UNKNOWN

   ∨ CHARGER-1=UNKNOWN]

5. [SWITCH=CHARGER-1 ∨ SWITCH=CHARGER-2 ∨ CHARGER-1=FULL-ON ∨

   CHARGER-1=OFF ∨

   SWITCH=STUCK-CHARGER-1 ∨ CHARGER-1=BROKEN ∨ SWITCH=UNKNOWN

   ∨ CHARGER-1=UNKNOWN]

6. [SWITCH=CHARGER-1 ∨ SWITCH=CHARGER-2 ∨ CHARGER-2=TRICKLE ∨

   CHARGER-2=OFF ∨ SWITCH=STUCK-CHARGER-2 ∨ CHARGER-2=BROKEN ∨

   SWITCH=UNKNOWN ∨ CHARGER-2=UNKNOWN]

7. [SWITCH=CHARGER-1 ∨ SWITCH=CHARGER-2 ∨ CHARGER-2=FULL-ON ∨

   CHARGER-2=OFF ∨ SWITCH=STUCK-CHARGER-2 ∨ CHARGER-2=BROKEN ∨

   SWITCH=UNKNOWN ∨ CHARGER-2=UNKNOWN]

8. [SWITCH=CHARGER-1 ∨ CHARGER-1=TRICKLE ∨ CHARGER-1=OFF ∨
   SWITCH=STUCK-CHARGER-1 ∨ SWITCH = STUCK-CHARGER-2 ∨
   CHARGER-1=BROKEN ∨ SWITCH=UNKNOWN ∨ CHARGER-1=UNKNOWN]

9. [SWITCH=CHARGER-1 ∨ CHARGER-1=FULL-ON ∨ CHARGER-1=OFF ∨
   SWITCH=STUCK-CHARGER-1 ∨ SWITCH = STUCK-CHARGER-2 ∨
   CHARGER-1=BROKEN ∨ SWITCH=UNKNOWN ∨ CHARGER-1=UNKNOWN]

10. [SWITCH=CHARGER-2 ∨ CHARGER-2=TRICKLE ∨ CHARGER-2=OFF ∨
    SWITCH=STUCK-CHARGER-1 ∨ SWITCH = STUCK-CHARGER-2 ∨
    CHARGER-2=BROKEN ∨ SWITCH=UNKNOWN ∨ CHARGER-2=UNKNOWN]

11. [SWITCH=CHARGER-2 ∨ CHARGER-2=FULL-ON ∨ CHARGER-2=OFF ∨
    SWITCH=STUCK-CHARGER-1 ∨ SWITCH = STUCK-CHARGER-2 ∨
    CHARGER-2=BROKEN ∨ SWITCH=UNKNOWN ∨ CHARGER-2=UNKNOWN]

12. [BATTERY = CHARGING ∨ BATTERY = DISCHARGING ∨ BATTERY =
    DEAD ∨ BATTERY = UNKNOWN]

13. [BATTERY = CHARGING ∨ BATTERY = FULL ∨ BATTERY = DEAD ∨
    BATTERY = UNKNOWN]

14. [BATTERY = CHARGING ∨ BATTERY = FULL ∨ BATTERY =
    DISCHARGING ∨ BATTERY = UNKNOWN]

15. [SWITCH=CHARGER-2 ∨ CHARGER-1=FULL-ON ∨ CHARGER-1=OFF ∨
    SWITCH=STUCK-CHARGER-1 ∨ SWITCH=STUCK-CHARGER-2 ∨ CHARGER-
    1=BROKEN ∨ SWITCH=UNKNOWN ∨ CHARGER-1=UNKNOWN]

16. [SWITCH=CHARGER-2 ∨ CHARGER-1=FULL-ON ∨ CHARGER-1=TRICKLE
    ∨ SWITCH=STUCK-CHARGER-1 ∨ SWITCH=STUCK-CHARGER-2 ∨
    CHARGER-1=BROKEN ∨ SWITCH=UNKNOWN ∨ CHARGER-1=UNKNOWN]

17. [SWITCH=CHARGER-1 ∨ CHARGER-2=FULL-ON ∨ CHARGER-2=OFF ∨
    SWITCH=STUCK-CHARGER-1 ∨ SWITCH=STUCK-CHARGER-2 ∨ CHARGER-
    2=BROKEN ∨ SWITCH=UNKNOWN ∨ CHARGER-2=UNKNOWN]

18. [SWITCH=CHARGER-1 ∨ CHARGER-2=FULL-ON ∨ CHARGER-2=TRICKLE
    ∨ SWITCH=STUCK-CHARGER-1 ∨ SWITCH=STUCK-CHARGER-2 ∨
    CHARGER-2=BROKEN ∨ SWITCH=UNKNOWN ∨ CHARGER-2=UNKNOWN]

19. [SWITCH=CHARGER-1 ∨ SWITCH=CHARGER-2 ∨ CHARGER-1=FULL-ON ∨
    CHARGER-1=OFF ∨ SWITCH=STUCK-CHARGER-2 ∨ CHARGER-1=BROKEN ∨
    SWITCH=UNKNOWN ∨ CHARGER-1=UNKNOWN]

```
20.[ SWITCH=CHARGER-1 ∨ SWITCH=CHARGER-2 ∨ CHARGER-1=FULL-ON ∨
    CHARGER-1=TRICKLE  ∨  SWITCH=STUCK-CHARGER-2  ∨  CHARGER-
    1=BROKEN ∨ SWITCH=UNKNOWN ∨ CHARGER-1=UNKNOWN ]
21.[ SWITCH=CHARGER-1 ∨ SWITCH=CHARGER-2 ∨ CHARGER-2=FULL-ON ∨
    CHARGER-2=OFF ∨ SWITCH=STUCK-CHARGER-1 ∨ CHARGER-2=BROKEN ∨
    SWITCH=UNKNOWN ∨ CHARGER-2=UNKNOWN ]
22.[ SWITCH=CHARGER-1 ∨ SWITCH=CHARGER-2 ∨ CHARGER-2=FULL-ON ∨
    CHARGER-2=TRICKLE  ∨  SWITCH=STUCK-CHARGER-1  ∨  CHARGER-
    2=BROKEN ∨ SWITCH=UNKNOWN ∨ CHARGER-2=UNKNOWN ]
```

Notice that the number of partial diagnoses does not equal the number of dissents specified in Section C.2.1. In transforming the dissents related to the battery voltage and current, the partial diagnoses resulting from dissents 10 and 11, and 12 and 13 are the same.

C.4 Reachable Current Modes

The space of possible current modes is generated using the compiled transitions that are enabled and the initial state specified in Section C.1. The following figure shows the initial state and the space of possible modes.
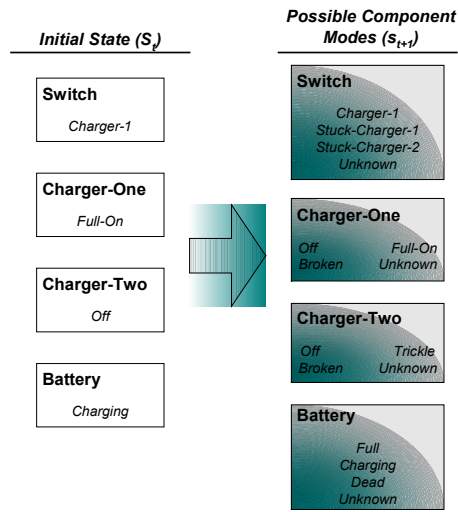
Figure C-2 - Space of Possible Component Modes

The space of possible component modes is shown above, and each mode also has an associated probability. The probabilities are as follows:

'*switch*' = { (*Charger-1*, p = 0.9899), (*Stuck-Charger-1*, p = 0.01), (*Stuck-Charger-2*, p = 0.01), (*Unknown*, p = 0.0001) }

'*charger-1*' = { (*Full-On*, p = 0.8899), (*Off*, p = 0.1), (*Broken*, p = 0.01), (*Unknown*, p = 0.0001) }

'*charger-2*' = { (*Off*, p = 0.1), (*Trickle*, p = 0.8899), (*Broken*, p = 0.01), (*Unknown*, p = 0.0001) }

'*battery*' = { (*Full*, p = 0.499), (*Charging*, p = 0.499), (*Dead*, p = 0.001), (*Unknown*, p = 0.0001) }

C.5 *Dynamic Mode Estimate Generation*

This portion of the Online Mode Estimation algorithm uses the partial diagnoses and the space of possible component modes to determine a diagnosis. Using the modified A* search described previously, the tree expansion is as follows. Since there is only one source state, the initial state, the process is simplified to only using this state to generate consistent current states. This section expands the partial diagnoses and walks through the expansion step by step.
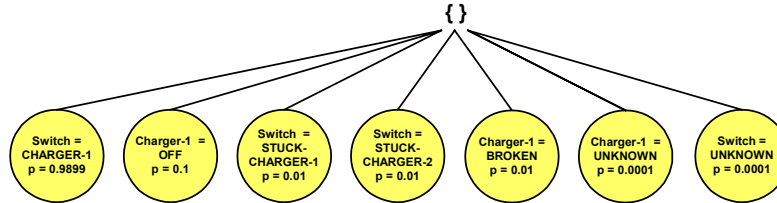
Figure C-3 - Expansion of Constituent diagnoses 1

The expansion of the first constituent diagnoses is shown above, and of these nodes, the search chooses the most likely node, in this case the component mode assignment '*switch = charger-1*'. The next step of the algorithm then determines which partial diagnoses this assignment satisfies.

In the case of '*switch = charger-1*', this assignment satisfies partial diagnoses 1 through 6 and 14 through 19. The next constituent diagnoses that is expanded then is 7, giving the following search tree:
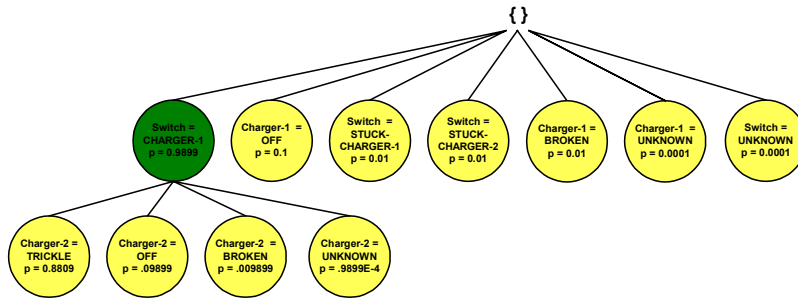
Figure C-4 - Expansion of Constituent diagnoses 7

Upon expanding the next available constituent diagnoses, 7, only the component mode assignments to '*charger-2*' are expanded because an assignment to the '*switch*' has already been chosen. Again, the search chooses to follow the most likely node of the search tree, in this case being to follow the '*switch = charger-1*' and '*charger-2 = trickle*' path of component mode assignments. However, upon following this path, the search determines that it is a dead end because there is no way to satisfy constituent diagnoses 8. As a result, this path of the search tree is cut off and is not considered any further. The search then finds the next most likely node in the tree, and this is the node '*charger-1 = off*'. This node satisfies partial diagnoses 1, 2, 5, 6, 12 and 16. The next constituent diagnoses that is expanded is then constituent diagnoses 3. The resultant expansion is represented in Figure C-4. The previous expansion under the '*switch*' is not shown so as to simplify the figure. These nodes are still considered in the search.
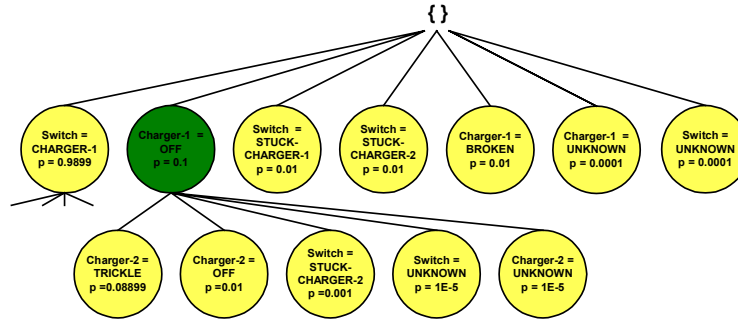
Figure C-5 - Expansion under '*Charger-1 = OFF*' Node of Constituent diagnoses 3

Using the expansion shown here, and the expansion of Figure C-4, the most likely path is under '*switch = charger-1*' and '*charger-2 = off*'. The constituent diagnoses that are satisfied by this path are 1 through 6, 7, 8, and 14 through 19. The next constituent diagnoses that is expanded under this search path is then constituent diagnoses 9, involving the '*battery*'. The expansion of this constituent diagnoses is shown in Figure C-6. Again, the expansion shown in Figure C-5 is not shown here for clarity.
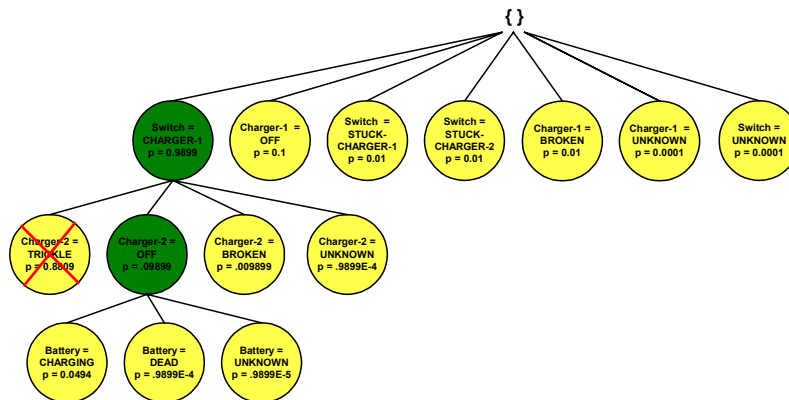
Figure C-6 - Expansion of Constituent diagnoses 9

The expansion shown above only shows the component mode assignments for the battery for '*charging*', '*dead*' and '*unknown*' because the '*discharging*' mode assignment is not in the allowable assignments for the

battery.   From the expansions of Figure C-6 and Figure C-5 the search
follows the most likely path of the tree. This next most likely path that the
search finds is then '*charger-1 = off*' and '*charger-2 = trickle*' with p =
0.08899.   The partial diagnoses satisfied by this set of assignments are 1
through 3, 5 through 7, 12, 15, 16 and 19.   The next expansion is then
performed using constituent diagnoses 4.
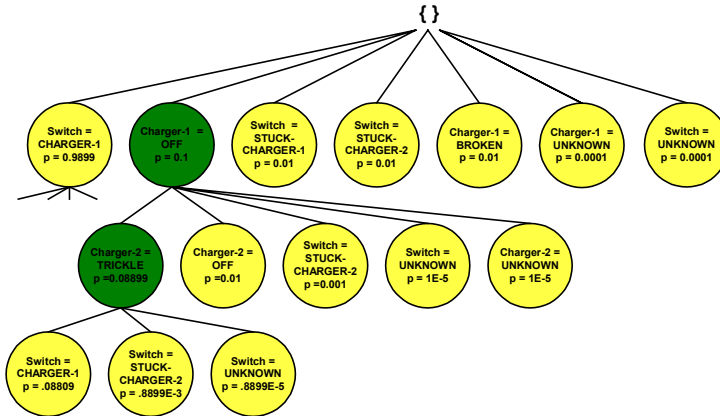
Figure C-7 - Expansion of the set of Constituent Diagnoses #4

Upon performing this expansion, each of the paths are checked for a dead
end.   Looking at the path '*charger-1 = off*', '*charger-2 = trickle*', and
'*switch = charger-1*', the remaining partial diagnoses to be satisfied are 8, 9
through 11 and 13, shown below.

```
8.  [  SWITCH=CHARGER-2  ∨  CHARGER-2=FULL-ON  ∨  CHARGER-2=OFF  ∨
       SWITCH=STUCK-CHARGER-1   ∨   SWITCH   =   STUCK-CHARGER-2   ∨
       CHARGER-2=BROKEN ∨ SWITCH=UNKNOWN ∨ CHARGER-2=UNKNOWN ]
9.  [  BATTERY  =  CHARGING  ∨  BATTERY  =  DISCHARGING  ∨  BATTERY  =
       DEAD ∨ BATTERY = UNKNOWN ]

10. [  BATTERY  =  CHARGING  ∨  BATTERY  =  FULL  ∨  BATTERY  =  DEAD  ∨
       BATTERY = UNKNOWN ]
11. [  BATTERY   =   CHARGING   ∨   BATTERY   =   FULL   ∨   BATTERY   =
       DISCHARGING ∨ BATTERY = UNKNOWN ]
```

300   Achieving Real-time Mode Estimation through Offline Compilation

```
13. [ SWITCH=CHARGER-2 ∨ CHARGER-1=FULL-ON ∨ CHARGER-1=TRICKLE
       ∨    SWITCH=STUCK-CHARGER-1    ∨    SWITCH=STUCK-CHARGER-2    ∨
       CHARGER-1=BROKEN ∨ SWITCH=UNKNOWN ∨ CHARGER-1=UNKNOWN ]
```

It is impossible for this branch of the search tree under '*charger-1 = off*' and '*charger-2 = trickle*' to satisfy all partial diagnoses. This branch is marked as a dead end by the search. The next node that the search then finds to expand is under the path '*switch = charger-1*', '*charger-2 = off*' and '*battery = charging*'. The remaining partial diagnoses to be satisfied under this path are 12 and 13. The expansion of constituent diagnoses 12 is shown in the Figure C-8.
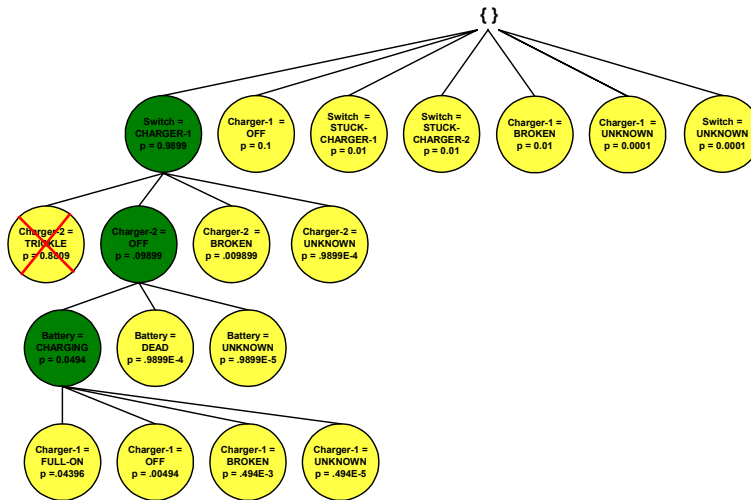
Figure C-8 - Expansion of Constituent diagnoses 12 under the Green Path

Following the path noted in 'green' on the search tree, and choosing the most likely node of the expansion, the search determines that all partial diagnoses have been satisfied by the assignment '*charger-1 = full-on*'. The search does continue however to generate consistent mode estimates until the halting conditions are met.

Appendix D.   CME Supporting Algorithms

D.1 Dissent and Transition Triggers

Recall that the Dissent and Transition Trigger algorithms are based on the property that the dissents and transitions involve antecedents that are known at the time of execution. In the case of dissents, there are observation assignments. The transitions involve command and component mode variables. This is exploited to simplify the triggering of dissents and transitions.

The basic idea of triggering is to determine if the assignments in the antecedents of the dissent or transition all appear in the current set of observations, and control variables, and in the previous mode estimates. If they do, then the dissent or transition is triggered, and referred to as enabled. A counter discipline is employed to determine when a dissent or transition is enabled. The triggered dissents are then placed in the list of Enabled Dissents, and the triggered transitions are placed in the Enabled Transitions. The complication alluded to in Chapter 6 is determining these lists with the fewest computations, and using truth-values to decrement and increment the counters. This section details the algorithms that perform these computations, beginning with the dissent and transition triggers, followed by the supporting algorithm that computes the truth-values of the different variables.

The inputs of the Dissent and Transition Trigger algorithms are shown below.
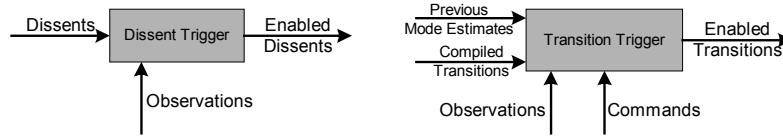
Figure D-1 - Inputs and Outputs of the Dissent and Transition Triggers

The Dissent Trigger only requires the observation information to determine when a particular dissent is enabled. The observations in this list are ones whose truth values have changed from time 't' to '$t+1$'. This list is denoted as $\Pi_o^{Changed}$. To simplify the implementation, the observations in this list have the added capability of knowing which dissents mention them. Storing this information enables the Dissent Trigger to only iterate through the list of changed observations instead of the full list of dissents. This is a standard indexing device which saves many computations over a brute force approach of iterating through the dissents and determining if the observations mentioned in the dissent are in the changed list of observations, $\Pi_o^{Changed}$. The steps for incrementing and decrementing are formalized in the Dissent Trigger algorithm below.

---

function Dissent-Trigger($\Pi_o^{Changed}$, *Dissents*)
    returns the enabled dissents, DS_{EN}
    for each $(x_{io} = v_{ij})$ in $\Pi_o^{Changed}$
        if *truth-current = true* & *truth-previous = false*
            then for each dissent, d_i, in dissents of $(x_{io} = v_{ij})$
                decrement the *OBS-counter* in d_i
                if *OBS-counter(d_i)* equals *zero*
                    then place d_i in DS_{EN}
            end

        if *truth-current = false* & *truth-previous = true*
            then for each dissent, d_i, in dissents of $(x_{io} = v_{ij})$
                increment the *OBS-counter* in d_i
            end
    end
    return DS_{EN}

The truth-values used by the Dissent Trigger are stored in two locations. The '*truth-previous*' represents if the observation assignment was true in the previous time step 't'. The '*truth-current*' represents if the observation assignment is true in the current time step, 't+1'.

The Transition Trigger uses the same list of changed observations as the Dissent Trigger, $\Pi_o^{Changed}$. The Transition Trigger uses a set of control variables reduced from the full set. This reduced set of assignments, represented by $\Pi_c^{Changed}$, are the control variables that have changed value from time 't' to 't+1'. The remaining inputs, the previous mode estimates, $B^{(t)}$, and the compiled transitions, $T_{Compiled}$, are unchanged external to the Transition Trigger algorithm.

The transition trigger is enabled by an algorithm that creates a single list of previous component modes, $\Pi_m^{Previous}$. This list is culled from all of the previous mode estimates by the algorithm 'Compress-Mode-Estimates'. This list of component modes allows the Transition Trigger algorithm to perform the same computation as it does for the observation and control variables. The algorithm iterates through these three lists, incrementing and decrementing the counters associated with each variable type and places the appropriate transitions in the list of enabled transitions, T_{EN}. The algorithm is detailed below.

---

function Transition-Trigger($\Pi_c^{Changed}$, $B^{(t)}$, $T_{Compiled}$)
    returns a list of *enabled transitions, T_{EN}*
    $\Pi_m^{Previous} \leftarrow$ Compress-Mode-Estimates($B^{(t)}$)
    for each $(x_{im} = v_{ij})$ in $\Pi_m^{Previous}$
        if *truth-previous = true*
            then for each T_i in transitions of $(x_{im} = v_{ij})$
                if $(x_{im} = v_{ij})$ is a source of transition T_i

then decrement the *SOURCE-counter* for T_i
if $(x_{im} = v_{ij})$ is in the guard of transition T_i
then decrement the *MODE-counter* for T_i
if *OBS-counter = 0* & *CMD-counter = 0* &
SOURCE-counter = 0 & *MODE-counter = 0*
then place T_i in T_{EN}
end

for each $(x_{ic} = v_{ij})$ in $\Pi_c^{Changed}$
if *truth-current = true* & *truth-previous = false*
then for each T_i in transitions of $(x_{ic} = v_{ij})$
decrement the *CMD-counter* for T_i
if *OBS-counter = 0* & *CMD-counter = 0* &
SOURCE-counter = 0 & *MODE-counter = 0*
then place T_i in T_{EN}
if *truth-current = false* & *truth-previous = true*
then for each T_i in z_i^{CMD} transitions
increment the *CMD-counter* for T_i
end
return T_{EN}

Figure D-3 - Transition Trigger Algorithm

The transition trigger algorithm is broken into two major steps. The first calls the 'Compress-Mode-Estimates' algorithm that returns the list of previous component modes, $\Pi_m^{Previous}$. The next portion of this first step is using these previous component modes to decrement the count of the transition's '*Source*' and '*Mode*' counters. The second step updates the '*CMD*' counters using the list of changed control variables, $\Pi_c^{Changed}$. At each step the counts for each variable type are checked to determine if the particular transition is enabled. If the transition is enabled, it is added to the list, T_{EN}.

This algorithm requires fewer computations than iterating through the list of transitions and determining if a variable in either the source or guard is in the current list of observations, commands or previous component modes.

D.1.1 Triggering Supporting Algorithms

The Dissent and Transition Trigger algorithms relied on two algorithms to enable their computations. The updating of truth values and the creation of reduced lists of observation and control variable assignments is the task of the 'Update-Truth' algorithm. The other is the 'Compress-Mode-Estimates' algorithm that takes the belief state, $B^{(t)}$, and produces a set of component modes culled from the belief state.

The 'Update-Truth' algorithm uses the full set of observations and control assignments, Π_o and Π_c, and the current lists of each, $\Pi_o^{Current}$ and $\Pi_c^{Current}$, to determine the changed list of observation and control assignments. The algorithm first moves the '*truth-current*' value of each assignment to the '*truth-previous*' field. The algorithm then iterates through the full list to determine if an assignment is in $\Pi_o^{Current}$ or $\Pi_c^{Current}$. If an assignment is in the current list, then the *truth-current* is updated to true. After updating each assignment's '*truth-current*' field, the two truth-values are compared, and if they are different, then the assignment is placed in the appropriate list of changed observations or control variables. Figure D-4 details the algorithm.

---

function Update-Truth(Π_o, Π_c, $\Pi_o^{Current}$, $\Pi_c^{Current}$)
    returns list of changed observations, $\Pi_o^{Changed}$, and commands, $\Pi_c^{Changed}$
    for each $(x_{io} = v_{ij})$ in Π_o
        truth-previous $\leftarrow$ *truth-current* for $(x_{io} = v_{ij})$
        if $(x_{io} = v_{ij}) \in \Pi_o^{Current}$
            then *truth-current* $\leftarrow$ *true* for $(x_{io} = v_{ij})$
            else *truth-current* $\leftarrow$ *false* for $(x_{io} = v_{ij})$
        if *truth-previous* != *truth-current*
            then $\Pi_o^{Changed} \leftarrow (x_{io} = v_{ij}) \cup \Pi_o^{Changed}$
    end

    for each $(x_{ic} = v_{ij})$ in Π_c
        truth-previous $\leftarrow$ *truth-current* for $(x_{ic} = v_{ij})$

if $(x_{ic} = v_{ij}) \in \Pi_c^{\text{Current}}$

      then *truth-current* $\leftarrow$ *true* for $(x_{ic} = v_{ij})$

      else *truth-current* $\leftarrow$ *false* for $(x_{ic} = v_{ij})$

if *truth-previous* **!** = *truth-current*

      then $\Pi_c^{\text{Changed}} \leftarrow (x_{ic} = v_{ij}) \cup \Pi_c^{\text{Changed}}$

end

return Π_o^{Changed} and Π_c^{Changed}

Figure D-4 – Update-Truth Algorithm Supporting Compiled Conflict Recognition

The final supporting algorithm of the Dissent and Transition Triggers is determining the list of previous modes. The set of previous modes is generated from all mode estimates in the previous belief state, 'B$^{(t)}$'. The following figure shows the desired calculation.

$$\left\{ \left(x_{1m} = v_{1l_1} \right), ..., \left(x_{im} = v_{il_j} \right), ..., \left(x_{nm} = v_{nl_n} \right) \right\}$$
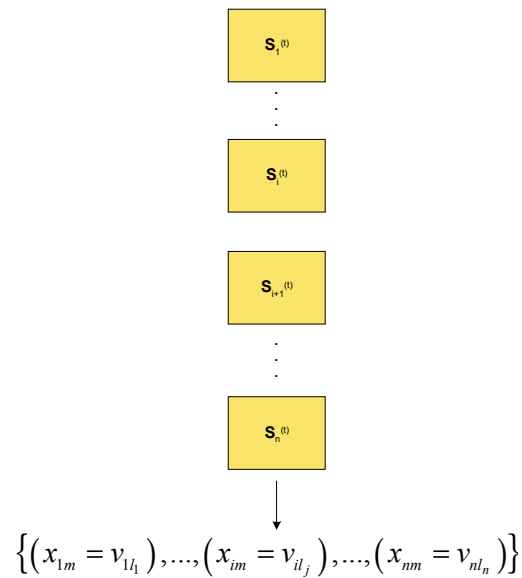
Figure D-5 - Compression of Previous Belief State

The compression of the belief state consists of a set of every component mode assignment that is mentioned in the individual mode estimates. When compressed, the list must represent the belief state, keeping knowledge of

the mode estimate probabilities. Note that a component mode should appear at most once in the list, but may be mentioned in multiple mode estimates. The 'Compress-Mode-Estimates' algorithm is shown below.

---

function Compress-Mode-Estimates($B^{(t)}$)
    returns a set of previous modes, $\Pi_m^{Previous}$
    for each $S_i^{(t)}$ in $B^{(t)}$
        for each $(x_{im} = v_{ij})$ in $S_i^{(t)}$
            mode estimate $\leftarrow \langle S_i^{(t)}, P(S_i^{(t)}) \rangle$ for $(x_{im} = v_{ij})$
            if $(x_{im} = v_{ij}) \notin \Pi_m^{Previous}$
                $\Pi_m^{Previous} \leftarrow (x_{im} = v_{ij}) \cup \Pi_m^{Previous}$
                truth-previous $\leftarrow$ *true* for $(x_{im} = v_{ij})$
                truth-current $\leftarrow$ *false* for $(x_{im} = v_{ij})$
        end
    end
    return $\Pi_m^{Previous}$

Figure D-6 - Compress States Algorithm

The algorithm iterates through each mode estimate in the previous belief state, and for each assignment places a reference to the mode estimate within a field in the assignment. Should an assignment be mentioned in more than one previous mode estimate, this field simply becomes a list. Also the 'truth-current' value is cleared since this is to be determined by the Dynamic Mode Estimate Generation.

D.2 Dynamic Mode Estimate Generation

D.2.1 Generate

The 'Insert-In-Order' algorithm will place a 'node' in the list of 'Nodes' in order of decreasing cost. The minor complexity is that if a node on the queue has a cost of 1, then this supercedes any other node in the queue. This is to force the Generate algorithm to choose each previous mode estimate at least once. With this in mind, the algorithm is as follows.

function Insert-In-Order(*new node*, *Nodes*)
    returns updated *Nodes* list
    for each *node* in *Nodes*
        if cost(*node*) = 1
            then move to *next node*
            if cost(*new node*) = cost(*node*)
                then put *new node* after *node*
                if cost(*new node*) > cost(*node*)
                    then put *node* before *node*
    end
    return *Nodes*

Figure D-7 - Insert-In-Order Algorithm Supporting the Generate Algorithm

This algorithm as designed puts the 'new node' after any node that has a cost of 1. This algorithm will also place 'new node' after one on the list if they both have the same cost, giving a tie to the node already in the queue. Finally, if the cost of the 'new node' is greater than the cost of the node in the list, the 'new node' is inserted before the one on the list.

This page intentionally left blank.

Appendix E. Results and Additional Experiments

E.1 Digital Shunt Nominal Operation

```
D:\MIT_Autonomy\Code\OnlineME\Debug\OnlineME.exe

>>>>>  Current Belief State >>>>>
        State list - ordered from most likely to least.
State information for: state (0) with probability: 9.887971e-001

        ( SA = operational [1 = 1]: 9.90e-001 : 9.90e-001 : 0.00e+000)
        ( DS-R = one-closed [3 = 2]: 9.70e-001 : 9.70e-001 : 0.00e+000)
        ( CH-P = trickle [6 = 2]: 8.90e-001 : 8.90e-001 : 0.00e+000)
        ( BTRY = full [8 = 1]: 9.90e-001 : 9.90e-001 : 0.00e+000)
        ( AS = none-closed [4 = 1]: 9.70e-001 : 9.70e-001 : 0.00e+000)
        ( CH-R = off [7 = 3]: 1.00e-001 : 1.00e-001 : 2.19e+000)
        ( SWCH = charger-p [5 = 1]: 9.90e-001 : 9.90e-001 : 0.00e+000)
        ( DS-P = one-closed [2 = 2]: 9.70e-001 : 9.70e-001 : 0.00e+000)

State information for: state (0) with probability: 1.019378e-002

        ( SA = operational [1 = 1]: 9.90e-001 : 9.90e-001 : 0.00e+000)
        ( DS-R = one-closed [3 = 2]: 9.70e-001 : 9.70e-001 : 0.00e+000)
        ( CH-P = trickle [6 = 2]: 8.90e-001 : 8.90e-001 : 0.00e+000)
        ( BTRY = full [8 = 1]: 9.90e-001 : 9.90e-001 : 0.00e+000)
        ( AS = none-closed [4 = 1]: 9.70e-001 : 9.70e-001 : 0.00e+000)
        ( CH-R = off [7 = 3]: 1.00e-001 : 1.00e-001 : 2.19e+000)
        ( SWCH = charger-p [5 = 1]: 9.90e-001 : 9.90e-001 : 0.00e+000)
        ( DS-P = stuck-closed [2 = 4]: 1.00e-002 : 1.00e-002 : 4.57e+000)

State information for: state (0) with probability: 9.987850e-004

        ( SA = broken [1 = 2]: 1.00e-003 : 1.00e-003 : 6.90e+000)
        ( DS-R = one-closed [3 = 2]: 9.70e-001 : 9.70e-001 : 0.00e+000)
        ( CH-P = trickle [6 = 2]: 8.90e-001 : 8.90e-001 : 0.00e+000)
        ( BTRY = full [8 = 1]: 9.90e-001 : 9.90e-001 : 0.00e+000)
        ( AS = none-closed [4 = 1]: 9.70e-001 : 9.70e-001 : 0.00e+000)
        ( CH-R = off [7 = 3]: 1.00e-001 : 1.00e-001 : 2.19e+000)
        ( SWCH = charger-p [5 = 1]: 9.90e-001 : 9.90e-001 : 0.00e+000)
        ( DS-P = one-closed [2 = 2]: 9.70e-001 : 9.70e-001 : 0.00e+000)

State information for: state (0) with probability: 1.029675e-005

        ( SA = broken [1 = 2]: 1.00e-003 : 1.00e-003 : 6.90e+000)
        ( DS-R = one-closed [3 = 2]: 9.70e-001 : 9.70e-001 : 0.00e+000)
        ( CH-P = trickle [6 = 2]: 8.90e-001 : 8.90e-001 : 0.00e+000)
        ( BTRY = full [8 = 1]: 9.90e-001 : 9.90e-001 : 0.00e+000)
        ( AS = none-closed [4 = 1]: 9.70e-001 : 9.70e-001 : 0.00e+000)
        ( CH-R = off [7 = 3]: 1.00e-001 : 1.00e-001 : 2.19e+000)
        ( SWCH = charger-p [5 = 1]: 9.90e-001 : 9.90e-001 : 0.00e+000)
        ( DS-P = stuck-closed [2 = 4]: 1.00e-002 : 1.00e-002 : 4.57e+000)

State information for: state (0) with probability: 1.081268e-008

        ( SA = operational [1 = 1]: 9.90e-001 : 9.90e-001 : 0.00e+000)
        ( DS-R = one-closed [3 = 2]: 9.70e-001 : 9.70e-001 : 0.00e+000)
        ( CH-P = trickle [6 = 2]: 8.90e-001 : 8.90e-001 : 0.00e+000)
        ( BTRY = full [8 = 1]: 9.90e-001 : 9.90e-001 : 0.00e+000)
        ( AS = none-closed [4 = 1]: 9.70e-001 : 9.70e-001 : 0.00e+000)
        ( CH-R = off [7 = 3]: 1.00e-001 : 1.00e-001 : 2.19e+000)
        ( SWCH = unknown [5 = 5]: 1.00e-006 : 1.00e-006 : 1.38e+001)
        ( DS-P = stuck-closed [2 = 4]: 1.00e-002 : 1.00e-002 : 4.57e+000)

Press any key to continue
```

E.2 Analog Shunt Nominal Operation

The nominal test of the analog shunts follows the same pattern as the digital shunts. In this case, the system is assumed operating normally with all components functioning. The NEAR spacecraft determines that too much power is being produced, so it gives the command, AS = close for the analog shunt to dissipate power. Under normal operation, this would result in the output current *Ishunt_PA = nominal*.

To begin the test, the system is assumed in the same modes as above for the digital shunts. The commands given to the system are:

{ DS-P-CMD = no-command, DS-R-CMD = no-command, AS-CMD = close }
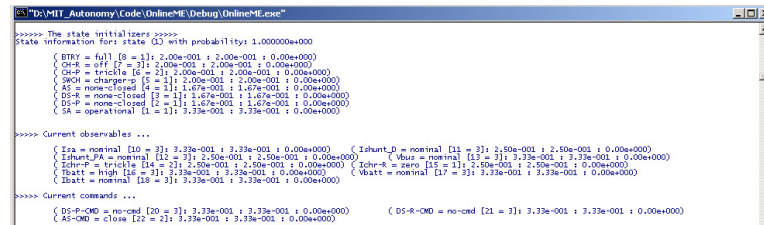
The observations input to the simulation are then:

{ Isa = nominal, Ishunt_D = nominal, Ishunt_PA = nominal, Ichr = trickle, Vbus = nominal, Vbatt = nominal, Tbatt = high, Ibatt = nominal }

The resultant mode estimate should include the changed component mode, *one-closed* for the analog shunts. The desired output is then:

{ SA = operational, DS-P = none-closed, DS-R = none-closed, AS = one-closed, S = CH-P, CH-P = trickle, CH-R = off, B = full }

The following is the output of the CME engine.

The inputs above produce the following most likely mode estimate. This is the same mode estimate expected for the scenario. The output here only shows the most likely mode estimate.

E.3 Nominal Battery Operation

E.4 Failed Analog Shunt

E.5 Solar Array Degradation

The next rule considered in the NEAR power system rules is one that indicates a low voltage on the bus caused by solar array degradation. The NEAR power system was designed to output a constant voltage at 24 V, and rule #26 addresses the situation when this voltage level drops below 23 V. Over time, the solar array productivity decreases due to many factors, such as thermal cycling, micrometeorite impacts, and the duration of exposure to the sun's radiation. If the solar array degrades enough, its power output is not at the expected level. This limits the operations the spacecraft can perform and is essential information to schedule tasks so that the available power is not exceeded.

The failure scenario described here is demonstrated assuming that all components are operating normally initially, which is given by the mode estimate:

{ SA = operational, DS-P = none-closed, DS-R = none-closed, AS-P = none-closed, S = CH-P, CH-P = trickle, CH-R = off, B = full }

The following are the observations and commands for this scenario:

{ Isa = low, Ishunt_D = nominal, Ishunt_PA = nominal, Ichr = trickle, Vbus = low, Tbatt = high, Vbatt = nominal, Ibatt = nominal }

{ DS-P-CMD = no-command, DS-R-CMD = no-command, AS-CMD = no-command }

The symptom of a low bus voltage and the low solar array output current is the indication that the solar array has broken in some way. One of these failures is due to solar array degradation. The desired output from the CME engine should contain the mode estimate:

{ SA = broken, DS-P = none-closed, DS-R = none-closed, AS-P = none-closed, S = CH-P, CH-P = trickle, CH-R = off, B = full }

The following is the output from the CME engine.

The observations, commands and previous mode estimate above result in the most likely mode estimate shown below.

E.6 Failed Charger


```
"D:\MIT_Autonomy\Code\OnlineME\Debug\OnlineME.exe"

>>>>>  Current Belief State >>>>>

        State list - ordered from most likely to least.
State information for: state (0) with probability: 9.596561e-001

        ( SA = operational [1 = 1]: 9.90e-001 : 9.90e-001 : 0.00e+000)
        ( DS-R = none-closed [3 = 1]: 9.70e-001 : 9.70e-001 : 0.00e+000)
        ( BTRY = full [8 = 1]: 9.90e-001 : 9.90e-001 : 0.00e+000)
        ( AS = none-closed [4 = 1]: 9.70e-001 : 9.70e-001 : 0.00e+000)
        ( CH-P = broken [6 = 4]: 1.00e-002 : 1.00e-002 : 4.49e+000)
        ( CH-R = trickle [7 = 2]: 8.90e-001 : 8.90e-001 : 0.00e+000)
        ( SWCH = charger-r [5 = 2]: 9.90e-001 : 9.90e-001 : 0.00e+000)
        ( DS-P = none-closed [2 = 1]: 9.70e-001 : 9.70e-001 : 0.00e+000)

State information for: state (0) with probability: 9.893362e-003

        ( SA = operational [1 = 1]: 9.90e-001 : 9.90e-001 : 0.00e+000)
        ( DS-P = none-closed [2 = 1]: 9.70e-001 : 9.70e-001 : 0.00e+000)
        ( BTRY = full [8 = 1]: 9.90e-001 : 9.90e-001 : 0.00e+000)
        ( AS = none-closed [4 = 1]: 9.70e-001 : 9.70e-001 : 0.00e+000)
        ( CH-P = broken [6 = 4]: 1.00e-002 : 1.00e-002 : 4.49e+000)
        ( CH-R = trickle [7 = 2]: 8.90e-001 : 8.90e-001 : 0.00e+000)
        ( SWCH = charger-r [5 = 2]: 9.90e-001 : 9.90e-001 : 0.00e+000)
        ( DS-R = stuck-open [3 = 5]: 1.00e-002 : 1.00e-002 : 4.57e+000)

State information for: state (0) with probability: 9.893362e-003

        ( SA = operational [1 = 1]: 9.90e-001 : 9.90e-001 : 0.00e+000)
        ( DS-R = none-closed [3 = 1]: 9.70e-001 : 9.70e-001 : 0.00e+000)
        ( BTRY = full [8 = 1]: 9.90e-001 : 9.90e-001 : 0.00e+000)
        ( AS = none-closed [4 = 1]: 9.70e-001 : 9.70e-001 : 0.00e+000)
        ( CH-P = broken [6 = 4]: 1.00e-002 : 1.00e-002 : 4.49e+000)
        ( CH-R = trickle [7 = 2]: 8.90e-001 : 8.90e-001 : 0.00e+000)
        ( SWCH = charger-r [5 = 2]: 9.90e-001 : 9.90e-001 : 0.00e+000)
        ( DS-P = stuck-closed [2 = 4]: 1.00e-002 : 1.00e-002 : 4.57e+000)

State information for: state (0) with probability: 9.893362e-003

        ( SA = operational [1 = 1]: 9.90e-001 : 9.90e-001 : 0.00e+000)
        ( DS-R = stuck-closed [3 = 4]: 1.00e-002 : 1.00e-002 : 4.57e+000)
        ( BTRY = full [8 = 1]: 9.90e-001 : 9.90e-001 : 0.00e+000)
        ( AS = none-closed [4 = 1]: 9.70e-001 : 9.70e-001 : 0.00e+000)
        ( CH-P = broken [6 = 4]: 1.00e-002 : 1.00e-002 : 4.49e+000)
        ( CH-R = trickle [7 = 2]: 8.90e-001 : 8.90e-001 : 0.00e+000)
        ( SWCH = charger-r [5 = 2]: 9.90e-001 : 9.90e-001 : 0.00e+000)
        ( DS-P = none-closed [2 = 1]: 9.70e-001 : 9.70e-001 : 0.00e+000)

State information for: state (0) with probability: 9.694475e-003

        ( SA = operational [1 = 1]: 9.90e-001 : 9.90e-001 : 0.00e+000)
        ( DS-R = none-closed [3 = 1]: 9.70e-001 : 9.70e-001 : 0.00e+000)
        ( BTRY = full [8 = 1]: 9.90e-001 : 9.90e-001 : 0.00e+000)
        ( AS = none-closed [4 = 1]: 9.70e-001 : 9.70e-001 : 0.00e+000)
        ( CH-P = broken [6 = 4]: 1.00e-002 : 1.00e-002 : 4.49e+000)
        ( CH-R = trickle [7 = 2]: 8.90e-001 : 8.90e-001 : 0.00e+000)
        ( SWCH = stuck-charger-r [5 = 4]: 1.00e-002 : 1.00e-002 : 4.60e+000)
        ( DS-P = none-closed [2 = 1]: 9.70e-001 : 9.70e-001 : 0.00e+000)
```

E.7 Failed Digital Shunts

Result of Second Observations

```
Select "D:\MIT_Autonomy\Code\OnlineME\Debug\OnlineME.exe"

>>>>>  Current Belief State >>>>>

        State list - ordered from most likely to least.
State information for: state (1) with probability: 9.567870e-001

        ( BTRY = full [8 = 1]: 9.90e-001 : 9.90e-001 : 0.00e+000)
        ( CH-R = off [7 = 3]: 1.00e-001 : 1.00e-001 : 2.19e+000)
        ( CH-P = trickle [6 = 2]: 8.90e-001 : 8.90e-001 : 0.00e+000)
        ( SWCH = charger-p [5 = 1]: 9.90e-001 : 9.90e-001 : 0.00e+000)
        ( AS = none-closed [4 = 1]: 9.70e-001 : 9.70e-001 : 0.00e+000)
        ( DS-R = stuck-open [3 = 5]: 1.00e-002 : 1.00e-002 : 4.57e+000)
        ( DS-P = one-closed [2 = 2]: 0.00e+000 : 9.70e-001 : 0.00e+000)
        ( SA = operational [1 = 1]: 9.90e-001 : 9.90e-001 : 0.00e+000)

State information for: state (2) with probability: 3.422300e-002

        ( BTRY = full [8 = 1]: 9.90e-001 : 9.90e-001 : 0.00e+000)
        ( CH-R = off [7 = 3]: 1.00e-001 : 1.00e-001 : 2.19e+000)
        ( CH-P = trickle [6 = 2]: 8.90e-001 : 8.90e-001 : 0.00e+000)
        ( SWCH = charger-p [5 = 1]: 9.90e-001 : 9.90e-001 : 0.00e+000)
        ( AS = none-closed [4 = 1]: 9.70e-001 : 9.70e-001 : 0.00e+000)
        ( DS-R = one-closed [3 = 2]: 9.70e-001 : 9.70e-001 : 0.00e+000)
        ( DS-P = one-closed [2 = 2]: 0.00e+000 : 9.70e-001 : 0.00e+000)
        ( SA = broken [1 = 2]: 1.00e-003 : 1.00e-003 : 6.90e+000)

State information for: state (3) with probability: 2.253200e-003

        ( BTRY = full [8 = 1]: 9.90e-001 : 9.90e-001 : 0.00e+000)
        ( CH-R = off [7 = 3]: 1.00e-001 : 1.00e-001 : 2.19e+000)
        ( CH-P = trickle [6 = 2]: 8.90e-001 : 8.90e-001 : 0.00e+000)
        ( SWCH = charger-p [5 = 1]: 9.90e-001 : 9.90e-001 : 0.00e+000)
        ( AS = none-closed [4 = 1]: 9.70e-001 : 9.70e-001 : 0.00e+000)
        ( DS-R = one-closed [3 = 2]: 9.70e-001 : 9.70e-001 : 0.00e+000)
        ( DS-P = one-closed [2 = 2]: 0.00e+000 : 9.70e-001 : 0.00e+000)
        ( SA = unknown [1 = 3]: 1.00e-006 : 1.00e-006 : 1.38e+001)

State information for: state (4) with probability: 3.253231e-004

        ( BTRY = full [8 = 1]: 9.90e-001 : 9.90e-001 : 0.00e+000)
        ( CH-R = off [7 = 3]: 1.00e-001 : 1.00e-001 : 2.19e+000)
        ( CH-P = trickle [6 = 2]: 8.90e-001 : 8.90e-001 : 0.00e+000)
        ( SWCH = charger-p [5 = 1]: 9.90e-001 : 9.90e-001 : 0.00e+000)
        ( AS = none-closed [4 = 1]: 9.70e-001 : 9.70e-001 : 0.00e+000)
        ( DS-R = one-closed [3 = 2]: 9.70e-001 : 9.70e-001 : 0.00e+000)
        ( DS-P = stuck-open [2 = 5]: 9.90e-001 : 1.00e-002 : 4.57e+000)
        ( SA = operational [1 = 1]: 9.90e-001 : 9.90e-001 : 0.00e+000)

State information for: state (5) with probability: 1.488640e-004

        ( BTRY = full [8 = 1]: 9.90e-001 : 9.90e-001 : 0.00e+000)
        ( CH-R = off [7 = 3]: 1.00e-001 : 1.00e-001 : 2.19e+000)
        ( CH-P = trickle [6 = 2]: 8.90e-001 : 8.90e-001 : 0.00e+000)
        ( SWCH = charger-p [5 = 1]: 9.90e-001 : 9.90e-001 : 0.00e+000)
        ( AS = none-closed [4 = 1]: 9.70e-001 : 9.70e-001 : 0.00e+000)
        ( DS-R = one-closed [3 = 2]: 9.70e-001 : 9.70e-001 : 0.00e+000)
        ( DS-P = unknown [2 = 6]: 1.00e-006 : 1.00e-006 : 1.38e+001)
        ( SA = operational [1 = 1]: 9.90e-001 : 9.90e-001 : 0.00e+000)
```

E.8 Failed Charger and Failed Analog Shunts

```
"D:\MIT_Autonomy\Code\OnlineME\Debug\OnlineME.exe"

>>>>>  Current Belief State >>>>>

          State list - ordered from most likely to least.
State information for: state (0) with probability: 9.604911e-001

        ( SA = operational [1 = 1]: 9.90e-001 : 9.90e-001 : 0.00e+000)
        ( DS-R = none-closed [3 = 1]: 9.70e-001 : 9.70e-001 : 0.00e+000)
        ( BTRY = full [8 = 1]: 9.90e-001 : 9.90e-001 : 0.00e+000)
        ( AS = stuck-open [4 = 5]: 1.00e-002 : 1.00e-002 : 4.57e+000)
        ( CH-P = broken [6 = 4]: 1.00e-002 : 1.00e-002 : 4.49e+000)
        ( CH-R = trickle [7 = 2]: 8.90e-001 : 8.90e-001 : 0.00e+000)
        ( SWCH = charger-r [5 = 2]: 9.90e-001 : 9.90e-001 : 0.00e+000)
        ( DS-P = none-closed [2 = 1]: 9.70e-001 : 9.70e-001 : 0.00e+000)

State information for: state (0) with probability: 9.901970e-003

        ( SA = operational [1 = 1]: 9.90e-001 : 9.90e-001 : 0.00e+000)
        ( DS-P = none-closed [2 = 1]: 9.70e-001 : 9.70e-001 : 0.00e+000)
        ( BTRY = full [8 = 1]: 9.90e-001 : 9.90e-001 : 0.00e+000)
        ( AS = stuck-open [4 = 5]: 1.00e-002 : 1.00e-002 : 4.57e+000)
        ( CH-P = broken [6 = 4]: 1.00e-002 : 1.00e-002 : 4.49e+000)
        ( CH-R = trickle [7 = 2]: 8.90e-001 : 8.90e-001 : 0.00e+000)
        ( SWCH = charger-r [5 = 2]: 9.90e-001 : 9.90e-001 : 0.00e+000)
        ( DS-R = stuck-open [3 = 5]: 1.00e-002 : 1.00e-002 : 4.57e+000)

State information for: state (0) with probability: 9.901970e-003

        ( SA = operational [1 = 1]: 9.90e-001 : 9.90e-001 : 0.00e+000)
        ( DS-R = none-closed [3 = 1]: 9.70e-001 : 9.70e-001 : 0.00e+000)
        ( BTRY = full [8 = 1]: 9.90e-001 : 9.90e-001 : 0.00e+000)
        ( AS = stuck-open [4 = 5]: 1.00e-002 : 1.00e-002 : 4.57e+000)
        ( CH-P = broken [6 = 4]: 1.00e-002 : 1.00e-002 : 4.49e+000)
        ( CH-R = trickle [7 = 2]: 8.90e-001 : 8.90e-001 : 0.00e+000)
        ( SWCH = charger-r [5 = 2]: 9.90e-001 : 9.90e-001 : 0.00e+000)
        ( DS-P = stuck-closed [2 = 4]: 1.00e-002 : 1.00e-002 : 4.57e+000)

State information for: state (0) with probability: 9.901970e-003

        ( SA = operational [1 = 1]: 9.90e-001 : 9.90e-001 : 0.00e+000)
        ( DS-R = stuck-closed [3 = 4]: 1.00e-002 : 1.00e-002 : 4.57e+000)
        ( BTRY = full [8 = 1]: 9.90e-001 : 9.90e-001 : 0.00e+000)
        ( AS = stuck-open [4 = 5]: 1.00e-002 : 1.00e-002 : 4.57e+000)
        ( CH-P = broken [6 = 4]: 1.00e-002 : 1.00e-002 : 4.49e+000)
        ( CH-R = trickle [7 = 2]: 8.90e-001 : 8.90e-001 : 0.00e+000)
        ( SWCH = charger-r [5 = 2]: 9.90e-001 : 9.90e-001 : 0.00e+000)
        ( DS-P = none-closed [2 = 1]: 9.70e-001 : 9.70e-001 : 0.00e+000)

State information for: state (0) with probability: 9.702911e-003

        ( SA = operational [1 = 1]: 9.90e-001 : 9.90e-001 : 0.00e+000)
        ( DS-R = none-closed [3 = 1]: 9.70e-001 : 9.70e-001 : 0.00e+000)
        ( BTRY = full [8 = 1]: 9.90e-001 : 9.90e-001 : 0.00e+000)
        ( AS = stuck-open [4 = 5]: 1.00e-002 : 1.00e-002 : 4.57e+000)
        ( CH-P = broken [6 = 4]: 1.00e-002 : 1.00e-002 : 4.49e+000)
        ( CH-R = trickle [7 = 2]: 8.90e-001 : 8.90e-001 : 0.00e+000)
        ( SWCH = stuck-charger-r [5 = 4]: 1.00e-002 : 1.00e-002 : 4.60e+000)
        ( DS-P = none-closed [2 = 1]: 9.70e-001 : 9.70e-001 : 0.00e+000)
```