A Decomposed Symbolic Approach to Reactive Planning

Seung H. Chung, Brian C. Williams

June 2003 SSL # 7-03

A DECOMPOSED SYMBOLIC APPROACH TO REACTIVE PLANNING

by

Seung H. Chung

B.A.Sc., Aeronautical and Astronautical Engineering
University of Washington, 1999

SUBMITTED TO THE DEPARTMENT OF AERONAUTICS AND ASTRONAUTICS
IN PARTIAL FULFILLMENT OF THE REQUIREMENT FOR THE DEGREE OF

MASTER OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2003

Signature of Author .....
Department of Aeronautics and Astronautics
May 23, 2003

Certified by .....
Brian C. Williams
Associate Professor of Aeronautics and Astronautics
Thesis Supervisor

Accepted by .....
Edward M. Greitzer
H.N. Slater Professor of Aeronautics and Astronautics
Chair, Department Committee on Graduate Students

A Decomposed Symbolic Approach to Reactive Planning

by

Seung H. Chung

Submitted to the Department of Aeronautics and Astronautics
on May 23, 2003, in partial fulfillment of the
requirements for the degree of
Master of Science at the
Massachusetts Institute of Technology

Abstract

Autonomous systems that operate in uncertain dynamic environments must respond to unanticipated events and goals by reconfiguring themselves in real-time. Reactive planning achieves reconfiguration in real-time by constructing a *goal-directed plan* (GDP) for all possible events and goals offline and then executing the GDP online, while monitoring the outcome of each execution step. A GDP, however, is susceptible to an explosion in space that is proportional to the square of the system's state space.

This thesis presents a new reactive plan encoding called a *decomposed goal-directed plan* (DGDP), which addresses the state space explosion problem by unifying two complementary approaches in the literature: *transition-based decomposition* and *symbolic encoding*. The DGDP encoding first uses transition-based decomposition to reduce the overall complexity of the planning problem by dividing the problem into a set of subproblems that may be solved independently, and then combined serially. This decomposition is based on the structure of the system's *transition dependency graph* (TDG), which captures the transition dependencies among the system components. Next, a GDP for each subproblem is generated using a compact symbolic encoding in terms of Ordered Binary Decision Diagrams (OBDD). Finally, these GDPs are combined into a full plan, the DGDP, based on the transition-based decomposition. This thesis makes two additional contributions to state-of-the-art reactive planning. First, it generalizes the "divide-and-conquer" approach introduced by the Burton reactive planner to systems with interdependent components, where a system with interdependent components is characterized by cycles within its TDG. Second, it generalizes OBDD plan encodings from universal plans, which are conditioned on all possible initial states, to goal-directed plans that are also conditioned on all possible goal states. The new decomposed symbolic reactive planner is empirically validated on representative spacecraft subsystem models.

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

Acknowledgments

This thesis is dedicated to my parents, Shin-Kwan and Young-Soon Chung. They are the source of all knowledge I deem most invaluable. They taught me of faith and love. They will remain the greatest teachers of my life, and I will eternally be grateful to them. I also thank my sisters, Hee-Won Chung and Hee-Sun Chung, for their love and friendship.

I especially thank my advisor, Professor Brian C. Williams, for his inspirational breadth of knowledge and guidance through which this thesis was made possible.

I thank Michel Ingham for his support and encouragement, not to mention his leadership as the senior graduate student of the Model-based Embedded and Robotic Systems Group. I also thank Paul Elliott and Robert Ragno for the discussions on C++ coding styles, and the rest of the Model-based Embedded and Robotic Systems Group members who worked together as a team in the development of the model-based executive.

A special thanks goes to Margaret Yoon, Greg Sullivan, and Mark Hilstad for their time commitment in proof-reading this thesis. I thank David Watson and Mike Pekala at the Johns Hopkins University Applied Physics Laboratory for being supportive of the model-based technology research and development. I thank all MIT students in the Space Systems Laboratory who have provided great moral support.

Most importantly, I thank God for His grace and blessings.

This research was supported in part by NASA's Cross Enterprise Technology Development program under contract NAG2-1466, DARPA's MOBIES program under contract F33615-00C-1702, and NASA Graduate Student Research Program Fellowship.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

NASA's MErcury Surface, Space ENvironment, GEochemistry, and Ranging (MESSENGER) mission will launch in March 2004 to explore Mercury for the first time in nearly 30 years. One of the most critical stages of the mission is Mercury orbit insertion (MOI). A fault during this stage could cause the MESSENGER spacecraft to either crash into Mercury or miss Mercury altogether, resulting in mission failure.

Traditionally, a ground operator controls a spacecraft by uploading the necessary sequence of commands. However, this type of open-loop commanding lacks robustness. Especially, if an anomaly occurs during the execution of the command sequence, the spacecraft could not only fail to achieve the intended objective, but the execution of the remaining commands in the sequence, given an off-nominal state, could potentially have disastrous effects.

If a spacecraft and its operational environment always behaved exactly as expected, open-loop commanding would suffice. Due to possible occurrences of anomalies, however, closed-loop control is necessary. Typically, control is provided with ground operators in the loop. While ground operators are very capable, their ability to react to anomalies is limited by the communication-time delay. For example, MESSENGER and Earth are so distant during MOI (approximately 200 million kilometers), that round-trip communication is delayed by approximately 21 minutes. If an unanticipated fault arises within 21 minutes prior to MOI, the spacecraft will be helpless.

Missions, like MESSENGER [1], use onboard rule-based systems for fault recovery. Rule-based system reactions are not limited by communication delay, but their recovery capability is limited to only the set of faults predicted by the engineers. Also, as spacecraft

become more complex to accommodate ambitious mission requirements, hand coding robust recovery rules becomes more arduous and prone to error. Thus, a new type of onboard reactive system is necessary to autonomously respond to anomalies.

1.1 Model-based Executive

Remote Agent, which flew on board Deep Space One (DS1) as one of the New Millennium Program technologies, provided the first demonstration of fully autonomous and adaptive operation of a spacecraft [27, 26]. Embracing the essential features of Remote Agent, the concepts of *model-based programming and execution* were developed to address the problems in traditional software development practice that can lead to software that is unreliable and lacks modularity and portability [30, 29]. Model-based programming was developed as an instance of the "executable specification" paradigm, in which a specification is automatically translated into system interactions by the model-based executive. This is in contrast to the traditional approach of hand translating a software implementation into a specification used for verification, where the hand coding opens up a considerable potential for human introduced errors. Model-based programming is distinguished from other executable specification languages [20, 4] in that it is *state and fault aware*. Specifications are expressed at a high level in terms of hidden state evolutions, rather than through detailed command and observation sequences. This allows the error prone process of reasoning through system interactions to be delegated to the model-based executive. Specifications are fault aware, in that they include models of the physical plant's nominal and faulty behaviors. This allows the executive to act appropriately during failure in order to achieve the specified state evolutions.

A model-based executive consists of two components, a *control sequencer* and a *deductive controller*, as shown in Figure 1-1 [29]. The control sequencer is responsible for generating a sequence of configuration goals using the control program and the plant state estimates. Each configuration goal specifies an abstract state of the plant to be achieved.

The deductive controller is comprised of two model-based modules, Mode Estimation

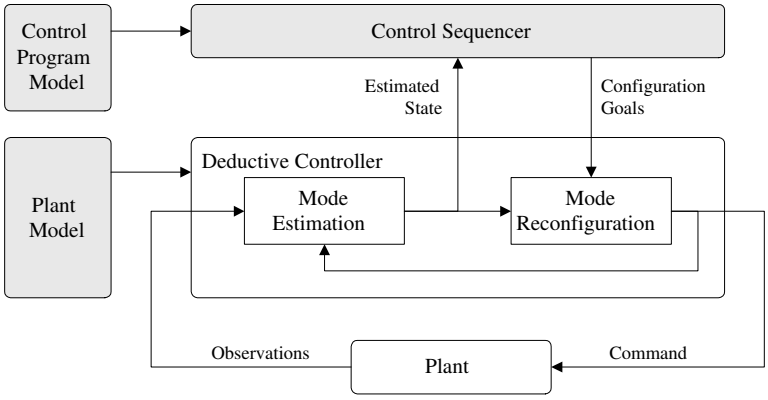

Figure 1-1: Model-based executive architecture.

(ME) and Mode Reconfiguration (MR) (see Figure 1-1). The two software modules along with the physical plant (e.g., spacecraft hardware) form a closed-loop control system. The ME module estimates the most likely state of the plant that is consistent with the model, the observations from the plant sensors, and the knowledge of the executed commands. Given the state estimates from ME, the MR module generates the commands necessary to achieve the goals specified by the control sequencer. Fault protection is inherent to this closed-loop architecture. In the event of a fault, ME diagnoses the faulty state, and MR immediately attempts to command the plant out of the faulty state and into the goal state that corresponds to the configuration goals. If such a command exists, MR executes it, thus recovering from the fault.

The MR module is comprised of two submodules as depicted in Figure 1-2. The first submodule is Goal Interpretation (GI). GI takes the configuration goals and generates the goal state that satisfies the configuration goals. The second submodule is the Reactive Planner (RP). RP determines and executes the command that will progress the plant from the estimated state to the goal state. The focus of this thesis is the RP submodule.

1.2 Motivation for Tractable Reactive Planning

While any general-purpose onboard planners could be put in place of the reactive planner, due to the PSPACE-complete nature of planning problems [9], real-time response cannot

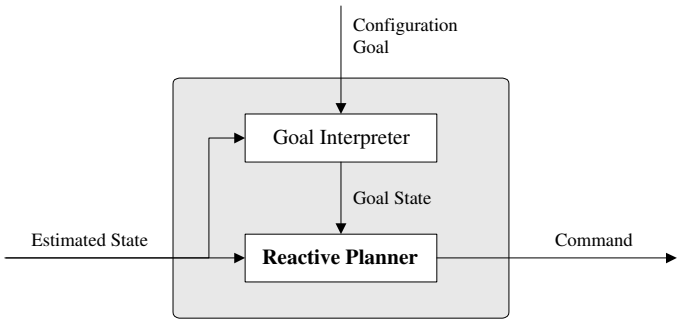

Figure 1-2: Mode Reconfiguration submodule of the deductive controller.

be guaranteed by a general-purpose planner. In time-critical situations, such as MOI, late response to a fault could be disastrous for the mission. A reactive planner compiles a plan offline for all possible situations, and then executes the plan online. Reactive planning is an approach that guarantees real-time response.

One of the early approaches to reactive planning is universal planning [28]. Given a goal state, a universal plan maps a set of all possible initial states to the actions that lead towards the goal state. With a universal plan, the correct sequence of actions can be decided at the execution-time, while observing the outcome of each action. Although universal plans provide the means to react to a nondeterministic environment in realtime, Ginsberg pointed out the intractability of universal planning due to the exponential state space explosion problem [18].

In addition, a universal plan cannot react to rapidly changing goals. Since a universal plan is valid only for a specific goal, a new universal plan must be generated for each new goal. As such, realtime response cannot be guaranteed when the goal changes. For example, if the primary propulsion system of MESSENGER fails just prior to MOI, the secondary propulsion system must be turned on, that is, a new goal state must be achieved.

Alternatively, Burton reactive planner produces a *goal-directed plan* (GDP) that can react to all possible initial states and goals as well. Burton is innovative for its use of a "divide-and-conquer" approach that divides a planning problem into smaller subproblems. This permits Burton to encode an extremely compact plan, where the plan's size

is linear in the number of system components. However, Burton's reactive planning capability is limited to problems in which no components are *interdependent*, that is, no two components may depend on one another for their transitions.

1.3 Thesis Contributions

This thesis presents a new reactive plan encoding called a *decomposed goal-directed plan* (DGDP), which addresses the state space explosion problem by unifying two complementary approaches in the literature: *transition-based decomposition* and *symbolic encoding*. The DGDP encoding first uses transition-based decomposition to reduce the overall complexity of the planning problem by dividing the problem into a set of subproblems that can be solved independently, and then combined serially. Next, a GDP for each subproblem is generated using a compact symbolic encoding in terms of Ordered Binary Decision Diagrams (OBDD). Finally, these GDPs are combined into a full plan, the DGDP, based on the transition-based decomposition.

This thesis makes two additional contributions to state-of-the-art reactive planning. First, it generalizes the "divide-and-conquer" approach introduced by the Burton reactive planner to systems with interdependent components. Second, it generalizes OBDD plan encodings from universal plans, which are conditioned on all possible initial states, to goal-directed plans that are also conditioned on all possible goal states.

1.3.1 Handling State Explosion through Decomposition

The method of divide-and-conquer is a well known, effective approach to solving problems. The *transition-based decomposition* approach [31] leveraged in this thesis is an approach that effectively divides a planning problem into subproblems. This decomposition is based on the structure of the transition dependency graph (TDG), which captures the transition dependencies among the system components. Although this method is very different in its specifics from that of the structural decomposition methods used for constraint satisfaction (CSP), database theory, and Bayesian Network problems, the contribution of transition-based decomposition to planning is analogous to that of structural

decomposition methods.

CSPs are known to be NP-complete in general; however, Freuder has shown that a CSP with a tree-structured constraint graph is solvable in linear time [17]. Similarly, Williams and Nayak have shown that if a planning problem has an acyclic TDG (i.e., the dependency among all components with respect to the transition conditions is unidirectional), then the problem can be solved within a state space that grows linearly in the number of system components [31]. For CSPs that do not have a tree-structured constraint graph, Dechter and Pearl have shown that the constraint graphs of those problems can be transformed into tree-structured graphs using a tree decomposition technique [16]. For planning problems with cyclic TDG (i.e., some components are interdependent), this thesis introduces a transition-based decomposition that transforms a cyclic TDG into an acyclic TDG. Thus, through the transition-based decomposition technique, even planning problems with cyclic TDG can be solved within a state space that grows approximately linearly in the number of system components.

1.3.2 Handling State Explosion through Symbolic Representation

The transition-based decomposition method reformulates a problem into a set of subproblems, then the global solution is constructed from serial solutions to the subproblems. The transition-based decomposition method addresses the state space explosion problem at the global level, similar to Burton, but then uses symbolic encoding methods to address the explosion problem difficulty at the subproblem level.

The logic synthesis and model checking communities have made very effective use of a symbolic representation called Ordered Binary Decision Diagrams (OBDD) [6] to construct compact state space encodings. An OBDD-based model checking technique has proven particularly successful in dealing with the state space explosion problem [8]. Recognizing the similarities between model checking and planning, Cimatti et al. introduced a new universal planning technique that takes advantage of the OBDDs [11]. Since then, several OBDD-based universal planning algorithms have been developed for operating

within nondeterministic domains [13, 12, 21]. Likewise, the new reactive planning approach adopts the OBDD encoding technique to compile the *goal-directed plans* (GDP) that can react to the nondeterministic environment as well as rapidly changing goals. In essence, a GDP maps all possible situations and goals to an action that is guaranteed to progress the system toward the goal state. The GDPs of the subproblems are combined into a *decomposed goal-directed plan* (DGDP) for the entire planning problem.

1.4 Thesis Outline

The remaining chapters discuss a new approach to reactive planning that unifies the aforementioned decomposition and symbolic representation methods. Figure 1-3 outlines the planning process that computes DGDPs:

First: A behavioral model of a system is represented as *concurrent automata*, $\mathcal{CA}$. In addition, for compactness these concurrent automata are encoded in OBDDs.

Second: The TDG of the concurrent automata is computed. The TDG describes how the transitions of an automaton depend on other concurrent automata.

Third: A cyclic TDG is transformed into an acyclic TDG by decomposing the system into smaller subsets. Then, the concurrent automata in each subset are composed into a single automaton.

Fourth: The composed automata are serialized in topological order.

Fifth: A DGDP is produced by computing a GDP for each composed automaton in topological order.

To set the context of this technical development, a spacecraft telecommunication system example is first introduced in Chapter 2. This example is used to help guide the reader throughout the aforementioned planning process. Chapter 3 maps to the first planning process. This chapter defines a concurrent automata formally and presents an

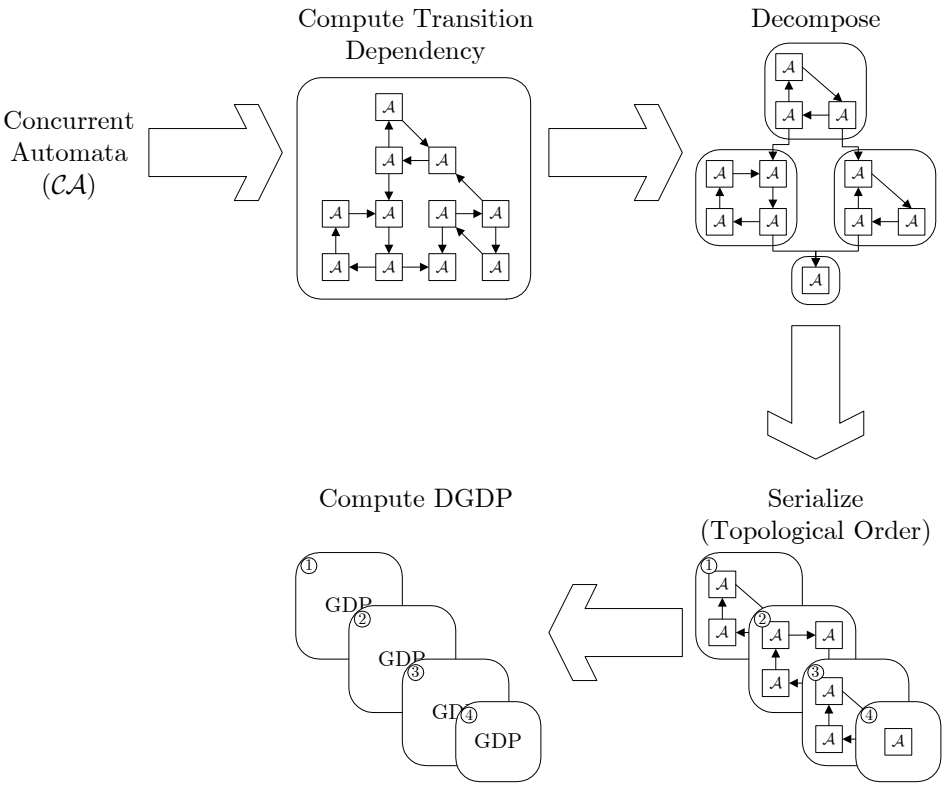

Figure 1-3: Decomposed goal-directed planning process.

OBDD representation of the concurrent automata. Before going into the details of decomposition, Chapter 4 introduces the method for computing and executing a GDP using a symbolic encoding. The algorithms outlined in this chapter also provides the foundation for computing DGDPs. Chapter 5 then discusses the decomposition, serialization, and DGDP computation. This chapter's discussions correspond to steps 3 through 5 of the planning process mentioned above. Chapter 6 discusses how a DGDP is executed, followed by Chapter 7, which provides a discussion of implementation and experimental results. Finally, Chapter 8 provides concluding statements and future work.

Chapter 2

Spacecraft Telecommunication System

NASA's MErcury Surface, Space ENvironment, GEochemistry, and Ranging (MESSENGER) mission will launch in March 2004 to explore Mercury for the first time in nearly 30 years. The objective of the mission is to further our understanding of Mercury's geological and atmospheric characteristics, ultimately advancing our knowledge of the terrestrial planets and their evolution. The telecommunication system is one of the spacecraft's critical subsystems required to achieve MESSENGER's objective. Without the telecommunication system, the commands necessary to carry out the science activities cannot be uploaded to MESSENGER and science data cannot be downloaded to Earth; without the science data, the mission will be a failure.

Maintaining an operational telecommunication system is crucial for the mission. If a reparable fault occurs in some component of the telecommunication system, it must be restored to an operational state immediately. An offline telecommunication system would result in the loss of science opportunities, as no commands can be uploaded to initiate any science activities. In the worst case, ground operators will be incapable of uploading the commands that initiate time-critical maneuvers, such as orbit insertion around Mercury, resulting in complete mission failure. Furthermore, since the ground operators cannot upload the commands necessary to repair the faulty telecommunication system, this repair operation must be autonomous.

As such, the telecommunication system is an appropriate example for illustrating an autonomous repair capability based on reactive planning. Furthermore, the complexity

and redundancy that the telecommunication system possesses presents an ideal scenario for an autonomous reconfiguration demonstration. This chapter introduces MESSENGER's telecommunication system design and provides a functional description of its components. Finally, a simplified, yet representative telecommunication system is introduced. The remaining chapters will use the simplified telecommunication system to describe the decomposed symbolic approach to reactive planning.

2.1 MESSENGER Telecommunication System

Due to its criticality to the mission, MESSENGER's telecommunication system was designed to be fully redundant with no credible single-point failures. As illustrated in Figure 2-1, the telecommunication system consists of two X-band Small Deep Space Transponders (DST), a hybrid coupler, two solid-state power amplifiers (SSPA), two radio-frequency (RF) switch assemblies, and eight antennas. A computer and a 1553 bus controller, a simplified integrated electronics module (IEM), were added to complete the data flow. In this section, each of these components is described in more detail.

Integrated Electronics Module

The IEM consists of three components: a computer, a 1553 bus controller, and a 1553 bus. The IEM's computer generates all data to be transmitted to Earth, and all data received from Earth is routed to the computer. The computer also commands all controllable devices (e.g., commanding the transponder to be on or off). The 1553 bus controller is responsible for directing the flow of data and commands between all devices connected on the 1553 bus. For example, the bus controller directs data to be downloaded to the appropriate DST, and routes the data received in the latest uplink by a DST back to the computer. Though the MESSENGER design includes a redundant IEM, only one is shown for the clarity of the diagram.

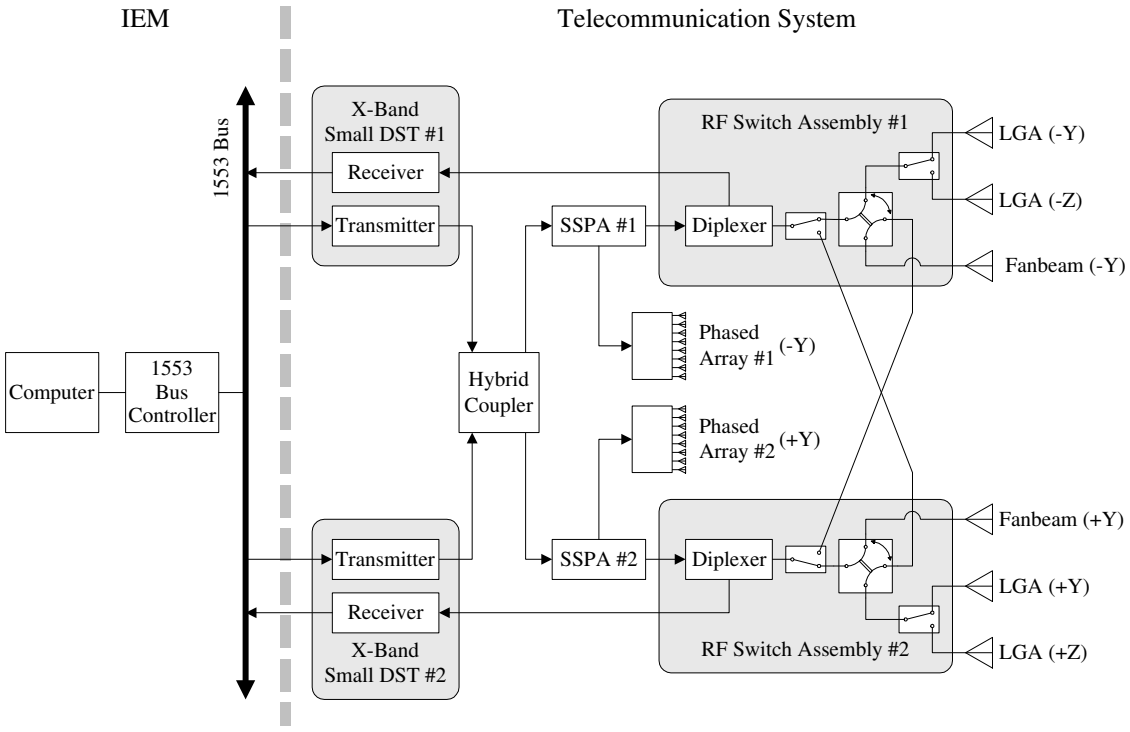

Figure 2-1: On the right side of the dotted line is the schematic of the MESSENGER telecommunication system [1]. To the left of the dotted line is a simplified schematic of MESSENGER's integrated electronics module (IEM).

Deep Space Transponder

Two DSTs are available within MESSENGER for redundancy. Each DST consists of a transmitter and a receiver. The transmitter converts the downlink data to an X-band signal that is appropriate for transmission via available antennas. The receiver converts the received X-band uplink signal into data that is recognizable by the computer.

Hybrid Coupler

A hybrid coupler is a passive device that distributes the signal from each transmitter to two SSPAs. The design of the hybrid coupler is very simple; as such, its failure rate is low enough not to be considered a credible point of failure, and no redundancy is necessary.

Solid-State Power Amplifier

Two SSPAs are available for redundancy; each one is associated with one RF switch assembly. Each SSPA is capable of amplifying the signal strength to the level required for transmission via one of the available antennas.

RF Switch Assembly

The amplified signal from each SSPA is sent to the corresponding RF switch assembly. An RF switch assembly is comprised of a diplexer and a series of switches. A diplexer allows the antennas to be used for both transmitting and receiving simultaneously. A series of switches in the two RF switch assemblies allow either of the diplexers to be connected with any of the available lowgain or fanbeam antennas, enabling multiple diplexer/antenna combinations to be used for transmitting and receiving data.

Antennas

The MESSENGER spacecraft uses three types of antennas. Two phased array antennas are used exclusively for downlinking data. A phased array antenna provides high bandwidth transmission at 40 bits per second (bps), with the ability to direct the signal in various directions. Unlike high bandwidth parabolic antennas, no mechanical parts are necessary for deployment or pointing. Lowgain and fanbeam antennas can be used

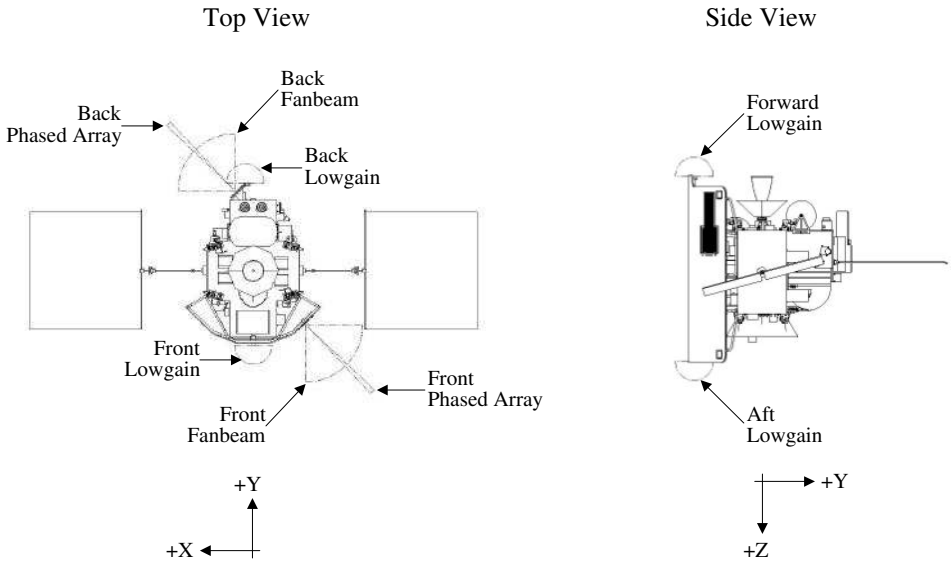

Figure 2-2: MESSENGER antenna locations with respect to the body axis [1].

for both uplink and downlink. The main difference is that a lowgain antenna (LGA) provides omnidirectional (i.e., approximately hemispherical coverage) transmission and reception at a bandwidth of 7.8 bps. A fanbeam antenna provides higher bandwidth at 10 bps, but the transmission and reception direction is limited compared to the lowgain antenna.

For additional redundancy, multiple antennas are positioned in various locations on the MESSENGER spacecraft (see Figure 2-2). The primary reason for the multiple antennas is that none of them can transmit or receive in all directions due to either limited beamwidth or transmission/reception obstruction by the spacecraft itself. Due to the close proximity of Mercury to the Sun, the MESSENGER spacecraft must cope with high heat and radiation. A heat shield in front of spacecraft protects the instruments onboard, and the MESSENGER spacecraft must always point its heat shield toward the Sun. The multiple location of antennas ensures that MESSENGER always has at least one antenna pointed toward the Earth, while the heat shield is maintained pointed toward the Sun. Furthermore, if one antenna fails, one of the others antenna can be used for transmission and reception.

2.2 Transmitter and Amplifier Interdependency

One important operational safety requirement on the telecommunication system is that the amplifier must be off before the transmitter can be turned on. The process of switching on the transmitter may generate a transient signal spike. If the amplifier is on and the power of the transient signal spike is beyond the acceptable range for the input to the amplifier, it could be damaged. Moreover, even if the transient signal does not damage the amplifier, the amplified transient signal spike may damage the devices downstream, such as the diplexer or antennas. Furthermore, for an additional safety, the transmitter is required to be on before the amplifier is turned on.

As described, the amplifier and transmitter components are *interdependent*, that is, the amplifier imposes an operational constraint on the transmitter, and the transmitter imposes an operational constraint on the amplifier. In many engineered systems, the behavioral specifications only impose a unidirectional operational constraints. Occasionally, however, the safety requirements impose bidirectional operational constraints on components. These interdependent components present a challenge for reactive planning. The upcoming chapters will demonstrate the use of transition-based decomposition to address this issue.

2.3 Simplified Telecommunication System

In the remaining chapters, a simplified model of MESSENGER's telecommunication system will be used to present the decomposed symbolic approach to reactive planning. As illustrated in Figure 2-3, the simplified system includes two transmitter/amplifier/LGA subsystems connected to the computer via the bus controller. The 1553 bus, receivers, hybrid coupler, diplexers, and switches are excluded from the simplified model. To simplify the model further, the computer is assumed to behave nominally at all times, thus the computer behavior is not modelled. Figure 2-3 depicts the direction of signal flow among the components. The computer sends data to be transmitted through the bus controller. When the data is received, the bus controller routes it to the transmitters.

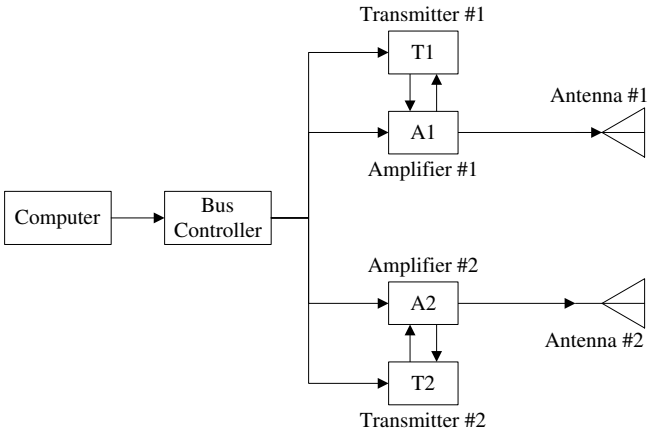

Figure 2-3: Simplified spacecraft telecommunication system.

The transmitters receive the data and generate the corresponding X-band signal. The signal is then amplified by the amplifiers and is finally transmitted through the antennas. The computer is also responsible for controlling the devices. For example, it may command the transmitter and amplifier to turn on or off. Again, these commands are sent to the appropriate devices via the bus controller. In the following sections, the behavior models for each component of the simplified telecommunication system are described in more detail.

2.3.1 Bus Controller Model

The computer can switch the Bus Controller on or off. Figure 2-4 illustrates the behavior of the component graphically. In the diagram, a state of a component is represented by a circle. The Bus Controller B has two operational modes: *on* (labelled $B = on$) and *off* (labelled $B = off$). For system safety concerns, a component model must include an unknown failure state that captures all unanticipated behaviors to ensure completeness of the model [24]. For most of the components in the simplified telecommunication system, however, these unknown failure modes have been omitted for the sake of keeping this illustrative example simple. However, in practice, the unknown failure state must never be left out.

A transition between states is represented by directed arcs, with the necessary transi-

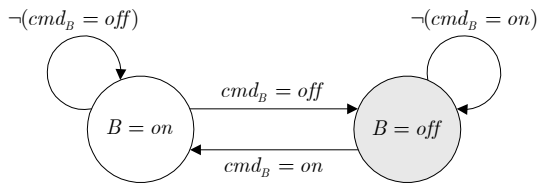

Figure 2-4: Simplified model of the Bus Controller.

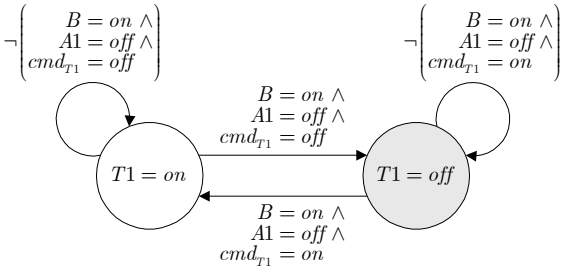

Figure 2-5: Simplified model of Transmitter #1.

tion condition(s) labelled on the arc. As illustrated in Figure 2-4, the computer can turn on the Bus Controller from the *off* state by simply issuing the command $cmd_B = on$. If the command is not issued (i.e., the condition $\neg(cmd_B = on)$ is true), the Bus Controller stays *off*. Here, the $\neg$ symbol represents the logical operator *not*. When the Bus Controller is in the *on* state, the computer may turn it off by commanding $cmd_B = off$. Otherwise, if the computer does not command the Bus Controller to turn off, $\neg(cmd_B = off)$, then the Bus Controller remains *on*.

2.3.2 Transmitter Model

As illustrated in Figure 2-5, Transmitter #1 also has two possible states, $T1 = on$ and $T1 = off$, similar to the Bus Controller. The conditions on the transitions, however, are more complex for the transmitter than for the Bus Controller. Since all commands from the computer are routed by the Bus Controller, the computer cannot command the transmitter if the Bus Controller is *off*. Thus, all transitions of the transmitter are conditioned on the state of the Bus Controller being *on*. Also, the interdependency between the transmitter and amplifier (as discussed in Section 2.2) adds extra complexity to the transition conditions.

For example, the computer can switch Transmitter #1 on from the *off* state only if the Bus Controller is currently *on*, the Amplifier #1 is *off*, and $cmd_{T1} = on$ is commanded. The simultaneous requirement of these three conditions is represented using the logical conjunction operator $\wedge$ (i.e., $B = on \wedge A1 = off \wedge cmd_{T1} = on$). If any of these conditions are not satisfied (i.e., $\neg (B = on \wedge A1 = off \wedge cmd_{T1} = on)$), the transmitter must remain in the *off* state.

Similarly, the computer can turn Transmitter #1 off from the *on* state only if the Bus Controller is currently *on*, the Amplifier #1 is *off*, and $cmd_{T1} = off$ is commanded (i.e., $B = on \wedge A1 = off \wedge cmd_{T1} = off$). Again, if the conjunction of the conditions is not satisfied, the transmitter must remain *on*.

The model for Transmitter #2 is exactly the same as Figure 2-5 except that $T1$, $A1$, and cmd_{T1} are replaced by $T2$, $A2$, and cmd_{T2}, respectively.

2.3.3 Amplifier Model

As illustrated in Figure 2-6, in addition to having *on* and *off* states, Amplifier #1 includes a repairable failure state ($A1 = resettable$). This state captures situations in which a failed amplifier can be repaired simply by turning it off and then back on. For example, when the state of the power amplifier becomes uncertain due to observed off-nominal behavior, spacecraft operators would restore the nominal behavior of the amplifier by resetting it. In the case of the Mars Polar Lander's SSPA, the engineers found that if the RF output power drops below the required level, cycling the power off and on solves the problem [10]. Inclusion of this state will help demonstrate the reactive planner's ability to repair in the upcoming chapters.

Due to the interdependency between Transmitter #1 and Amplifier #1, the transition from *off* to *on* is conditioned on the Bus Controller being *on*, Transmitter #1 also being *on*, and Amplifier #1 being commanded on ($B = on \wedge T1 = off \wedge cmd_{A1} = on$). Otherwise, the amplifier remains in the *off* state. Since turning the amplifier off does not have any harmful effects on any downstream components, turning the amplifier off from the *on* state only requires the Bus Controller to be *on* and the amplifier to be commanded off

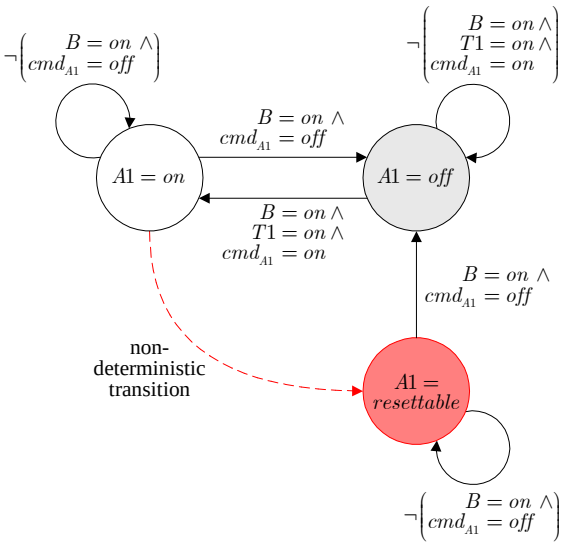

Figure 2-6: Simplified model of Amplifier #1.

$(B = on \wedge cmd_{A1} = off)$. For the same reason, the computer can turn the amplifier off from the *resettable* state if $B = on \wedge cmd_{A1} = off$. Note that from the *on* state, the amplifier can non-deterministically fail to the *resettable* state at any time without the need to satisfy any constraints. Such non-deterministic unconstrained transitions are typical of failures, which generally occur unexpectedly.

Again, the model of the Amplifier #2 is exactly the same as Figure 2-6 except that $T1$, $A1$, and cmd_{T1} are replaced by $T2$, $A2$, and cmd_{T2}, respectively.

2.3.4 Antenna Model

A lowgain antenna (LGA) is a passive device consisting of two possible states, *nominal* and $failed$ (see Figure 2-7). The *nominal* state is the operational state of the antenna. The $failed$ state represents the unknown faulty state of the antenna. Although the antennas themselves serve no purpose in demonstrating commanding and repair, they will be used to demonstrate the reactive planner's capability to reconfigure the system quickly back to an operational state when an irreparable failure occurs. For example, assume that the Transmitter, Amplifier, and Antenna #1 are being used for downlink. If Antenna #1 fails, the reactive planner must generate commands to turn on the Transmitter and

Figure 2-7: Simplified model Antenna #1.

Amplifier #2 to maintain downlink capability.

Chapter 3

Symbolic Representation of Concurrent Automata

A central idea in the model-based programming paradigm is the notion of an *executable specification* [29]. In an executable specification, the system behavioral description is used directly for reactive planning. Thus, the conceptual description of the system behavior must be written in, or automatically mapped to, some form of model on which deductive algorithms can operate. Furthermore, the computational model must be capable of representing complex behaviors of a system while facilitating computationally tractable reactive planning.

Within the model-based execution framework, the behavior of the system being controlled is modelled as a factored partially observable Markov decision process (POMDP) that is compactly encoded as probabilistic *concurrent constraint automata* (CCA) [30]. Concurrency is used to model the behavior of a set of components that operate synchronously. Constraints are used to represent co-temporal interactions and intercommunication between components. Probabilistic transitions are used to model the stochastic behavior of components, such as failure.

While compact, this representation is also expressive enough to facilitate both mode estimation and reconfiguration. For the purpose of reactive planning, however, only a subset of the full CCA model is necessary, corresponding to state and control variables and transition functions. The constraints on dependent variables are eliminated by substituting them with entailed constraints on state and control variables. The essential elements of the CCA model are extracted using knowledge compilation methods [31]

and encoded as *concurrent automata* ($\mathcal{CA}$) for reactive planning. In essence, $\mathcal{CA}$ represent a nondeterministic transition system with finite state and concurrently operating components. Compiling a CCA into a $\mathcal{CA}$ eliminates the need for constraint-based reasoning. Also, the elimination of the dependent variables reduces the size of the state space. For example, the Deep Space One (DS1) CCA model developed for the Remote Agent included approximately 3000 propositional variables; with the dependent variables eliminated, only about 100 variables remained.

Regardless of the compactness of this representation, the exponential state space explosion problem pointed out by Ginsberg [18] still remains for reactive planning. This problem is addressed by leveraging the transition-based decomposition and the compact state space encoding capability of Ordered Binary Decision Diagrams (OBDD), i.e., a symbolic encoding. OBDD-based model checking [8] and OBDD-based universal planning [11, 13, 12, 21] have proven particularly successful in dealing with the state explosion problem. This problem can be mitigated for reactive planning by encoding $\mathcal{CA}$ as OBDDs, similar to how a typical automaton is encoded in an OBDD [7]. Once $\mathcal{CA}$ is represented using OBDDs, the planning algorithms can be defined in terms of OBDD operators.

In this chapter, a formal description of the $\mathcal{CA}$ computational model is introduced. Then, after a brief introduction to OBDDs, the OBDD representation of $\mathcal{CA}$ is discussed. Examples based on the simplified telecommunication system discussed in Chapter 2 are interleaved throughout this chapter.

3.1 Computational Model: Concurrent Automata

$\mathcal{CA}$ denote a set of concurrently operating automata. Though the $\mathcal{CA}$ model has not yet been formally introduced, the models of the simplified telecommunication system depicted in Section 2.3 (see Figures 2-4, 2-5, 2-6, and 2-7) are, in fact, graphical representations of the system's $\mathcal{CA}$. In this section, the automaton for a single component is first formally defined, then $\mathcal{CA}$ is defined as a set of such automata. These definitions are similar to the definition of a CCA [30, 31, 29].

3.1.1 Automaton Definition

Each automaton has an associated *state variable* s_i with domain $\mathcal{D}(s_i)$. Given the current state assignment $(s_i = v)$, an automaton transitions its state in the next time step, according to a transition function τ_i. A transition function may be conditioned on a constraint involving the state of other automata and/or values of control variables. A transition is *enabled* if its constraint is entailed. In general, the domain of all control variables includes the default *noCmd* (i.e., no command) value. Formally, the automaton is defined as follows:

Definition 3.1. The *automaton* $\mathcal{A}_i$ for component i is a 4-tuple $\langle \Pi_i, \Sigma_i, \tau_i, \sigma_i^{(0)} \rangle$, where:

1. $\Pi_i = \Pi_i^s \cup \Pi_i^c$ is a finite set of variables for the component, where each variable $x \in \Pi_i$ ranges over a finite domain $\mathcal{D}(x)$. Π_i is partitioned into a set of control variables Π_i^c and a set of state variables Π_i^s that includes the component's state variable s_i and possibly other $s_{j \neq i}$.

2. Σ_i is a finite set of full assignments over Π_i. A state of the automaton, denoted σ_i, is an assignment to the component state variable s_i (i.e., $\sigma_i \equiv (s_i = v)$), where $s_i \in \Pi_i^s$ and $v \in \mathcal{D}(s_i)$. The state space of the component is the set $\Sigma_i^{s_i} \subset \Sigma_i$, the projection of Σ_i to variable s_i. $\Sigma_i^c \subset \Sigma_i$ is the projection of Σ_i to Π_i^c.

3. $\tau_i : \Sigma_i^{s_i} \times \mathbb{C}(\Pi_i) \rightarrow 2^{\Sigma_i^{s_i}}$ is a transition function. $\mathbb{C}(\Pi_i)$ denotes the set of all finite domain constraints over Π_i. A constraint is defined using *propositional state logic*, in which a proposition is an assignment to a variable $(x = v)$, or one of the constants $true$ or $false$. Propositions are composed into a formula using the standard first-order logic operators: AND $(\wedge)$, OR $(\vee)$, and NOT $(\neg)$. Given a state assignment $\sigma_i^{(t)} \in \Sigma_i^{s_i}$ at time t and a constraint $c_i^{(t)} \in \mathbb{C}(\Pi_i)$ entailed at time t, $\tau_i(\sigma_i^{(t)}, c_i^{(t)})$ specifies a set of states to which the automaton can transition at time $t+1$. The transition function captures both nominal and fault behaviors, represented by $\tau_i^n \subseteq \tau_i$ and $\tau_i^f \subseteq \tau_i$, respectively. In the absence of fault behavior,

the nominal transition function is always deterministic (i.e., $\tau_i^n : \Sigma_i^{s_i} \times \mathbb{C}(\Pi_i) \rightarrow \Sigma_i^{s_i}$). The fault transitions introduce nondeterminism into the system.

4. $\sigma_i^{(0)} \in \Sigma_i^{s_i}$ is the initial state of the automaton.

For example, consider Amplifier #1 illustrated once again in Figure 3-1. The automaton is represented as 4-tuple $\langle \Pi_{A1}, \Sigma_{A1}, \tau_{A1}, \sigma_{A1}^{(0)} \rangle$ where:

1. $\Pi_{A1} = \{B, A1, T1, cmd_{A1}\}$ is the set of variables, of which the state variable of Amplifier #1's component is $A1$. The variables are partitioned into a set of state variables $\Pi_{A1}^s = \{B, A1, T1\}$ and a set of control variables $\Pi_{A1}^c = \{cmd_{A1}\}$. The domain of $A1$ is $\mathcal{D}(A1) = \{off, on, resettable\}$, and the domain of the remaining variables are $\mathcal{D}(B) = \mathcal{D}(T1) = \{off, on\}$ and $\mathcal{D}(cmd_{A1}) = \{off, on, noCmd\}$.

2. While set $\Sigma_{A1} = \mathcal{D}(B) \times \mathcal{D}(A1) \times \mathcal{D}(T1) \times \mathcal{D}(cmd_{A1})$ is too large to list, having 24 elements, the set of Amplifier #1 states (i.e., projection of Σ_{A1} to s_{A1}) is $\Sigma_{A1}^{s_{A1}} = \{A1 = off, A1 = on, A1 = resettable\}$, and the projection of Σ_{A1} to Π_{A1}^c is $\Sigma_{A1}^c = \{cmd_{A1} = off, cmd_{A1} = on\}$.

3. The set of constraints $\mathbb{C}(\Pi_{A1})$ associated with the transition function τ_{A1} is defined in Equation 3.1.

$$\mathbb{C}(\Pi_{A1}) = \left\{ \begin{array}{c} \neg\left(B = on \wedge T1 = on \wedge cmd_{A1} = on\right), \\ B = on \wedge T1 = on \wedge cmd_{A1} = on, \\ \neg\left(B = on \wedge cmd_{A1} = off\right), \\ B = on \wedge cmd_{A1} = off, \\ \vdots \end{array} \right\} \tag{3.1}$$

The transition function $\tau_{A1}(\sigma_{A1}, c_{A1})$, where $\sigma_{A1} \in \Sigma_{A1}$ and $c_{A1} \in \mathbb{C}(\Pi_{A1})$ is defined in Table 3.1.

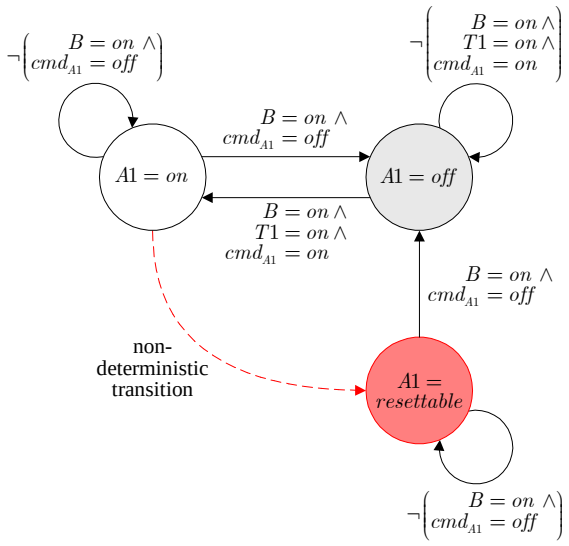

Figure 3-1: Automaton of Amplifier #1 from the simplified telecommunication system.

Table 3.1: Amplifier #1's Transition Function

$\sigma_{A1} \in \Sigma_{A1}$	$c_{A1} \in \mathbb{C}_{A1}$	$\tau_{A1}(\sigma_{A1}, c_{A1})$
$A1 = off$	$\neg(B = on \wedge T1 = on \wedge cmd_{A1} = on)$	$\{A1 = off\}$
$A1 = off$	$B = on \wedge T1 = on \wedge cmd_{A1} = on$	$\{A1 = on\}$
$A1 = on$	$\neg(B = on \wedge cmd_{A1} = off)$	$\{A1 = on, A1 = resettable^{\dagger}\}$
$A1 = on$	$B = on \wedge cmd_{A1} = off$	$\{A1 = off, A1 = resettable^{\dagger}\}$
$A1 = resettable$	$\neg(B = on \wedge cmd_{A1} = off)$	$\{A1 = resettable\}$
$A1 = resettable$	$B = on \wedge cmd_{A1} = off$	$\{A1 = off\}$

†Represents faulty behavior.

3.1.2 Concurrent Automata

$\mathcal{CA}$ is a set of concurrently operating automata. The simplified telecommunication system's $\mathcal{CA}$ consists of seven automata, one for each modelled component (i.e., Bus Controller, Transmitter #1 and #2, Amplifier #1 and #2, and Antenna #1 and #2). Within this formalism, all automata are assumed to operate synchronously, that is, at each time step every component performs a single state transition. In this section, $\mathcal{CA}$ and its *legal execution* are formally defined.

Definition 3.2. $\mathcal{CA}$ is a 3-tuple $\langle \mathcal{A}, \Pi, \Sigma \rangle$, where $\mathcal{A} = \{\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n\}$ is a finite set of automata associated with the n components in the system. $\Pi = \Pi^s \cup \Pi^c$ is the set of system variables, where each variable $x \in \Pi$ ranges over a finite domain $\mathcal{D}(x)$. Π is partitioned into a set of control variables $\Pi^c = \bigcup_{i=1}^{n} \Pi_i^c$ and a set of state variables $\Pi^s = \bigcup_{i=1}^{n} \Pi_i^s$. $\Sigma = \prod_{i=1}^{n} \Sigma_i$ is a finite set of full assignments over Π_i.

The state space of $\mathcal{CA}$, denoted Σ^s, is the Cartesian product of the individual automaton state spaces $\Sigma_i^{s_i}$, for all automata $\mathcal{A}_i \in \mathcal{A}$. The state of the system at time t, $\sigma^{(t)} \in \Sigma^s$, is $\bigcup_{i=1}^{n} \sigma_i^{(t)}$, where $\sigma_i^{(t)}$ is the state of automaton $\mathcal{A}_i$ at time t. Similarly, a control action $\mu^{(t)} \in \Sigma^c$ is an assignment to all control variables, Π^c.

Definition 3.3. A *legal execution* of a $\mathcal{CA}$ is a trajectory of states $[\sigma^{(0)}, \sigma^{(1)}, \ldots]$ and control actions $[\mu^{(0)}, \mu^{(1)}, \ldots]$ such that:

1. $\sigma^{(0)}$ is an initial state of the system.
2. $\sigma^{(t+1)} \in \prod_{i=1}^{n} \bigcup_j \tau_i(\sigma_i^{(t)}, c_j)$ is the state of the system in the next time step $t+1$, for all $c_j \in \mathbb{C}(\Pi_i)$ entailed by $\sigma^{(t)}$ and $\mu^{(t)}$.

The second part of the definition asserts that the next state, $\sigma^{(t+1)}$, is defined by the transition functions $\{\tau_i(\sigma_i^{(t)}, c_j) | i = 1, 2, \ldots, n\}$, where each transition is enabled by some $c_j \in \mathbb{C}(\Pi_i)$ that the system's state and control actions entail at time t. Due to the system's

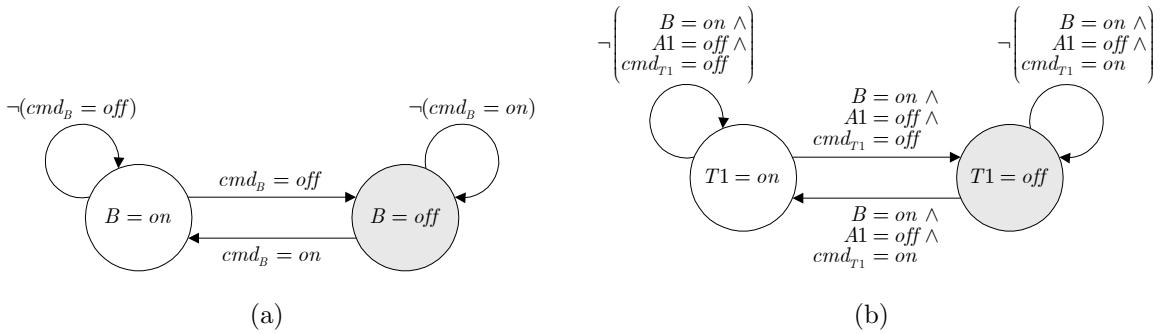

Figure 3-2: Concurrent automata of (a) Bus Controller and (b) Transmitter #1.

nondeterministic behavior, the enabled transitions may lead to a set of possible states. Thus, $\sigma^{(t+1)}$ is defined as one of those possible states, that is $\sigma^{(t+1)} \in \prod_{i=1}^{n} \bigcup_j \tau_i(\sigma_i^{(t)}, c_j)$.

For example, consider a subset of the simplified telecommunication system in which there are only three components: Bus Controller, Amplifier #1, and Transmitter #1. The graphical representations of the three concurrent automata are once again illustrated in Figures 3-2(a), 3-2(b), and 3-1, respectively. A state trajectory

$$\begin{bmatrix} \{B = \mathit{off}, T1 = \mathit{off}, A1 = \mathit{off}\}, \\ \{B = on, T1 = \mathit{off}, A1 = \mathit{off}\}, \\ \{B = on, T1 = on, A1 = \mathit{off}\}, \\ \{B = on, T1 = on, A1 = on\}, \\ \vdots \end{bmatrix} \tag{3.2}$$

and a sequence of control actions:

$$\begin{bmatrix} \{cmd_B = on, cmd_{T1} = noCmd, cmd_{A1} = noCmd\}, \\ \{cmd_B = noCmd, cmd_{T1} = on, cmd_{A1} = noCmd\}, \\ \{cmd_B = noCmd, cmd_{T1} = noCmd, cmd_{A1} = on\}, \\ \vdots \end{bmatrix} \tag{3.3}$$

represent a legal execution. In this scenario, all components are initially off, $\{B = \mathit{off}, T1 = \mathit{off}, A1 = \mathit{off}\}$, and one by one each component is turned on.

3.2 Symbolic Representation of $\mathcal{CA}$

The use of a symbolic encoding called Ordered Binary Decision Diagram (OBDD) for reactive planning is motivated by the exponential state space explosion problem. This problem is not exclusive to reactive planning, but it has been a major source of discouragement for research in this area. After all, responsiveness, the objective of reactive planning, is generally realized by trading off the required computational time for the memory space.

The state space explosion problem is obvious and can be observed even in a model as simple as the simplified telecommunication system. Of the seven components in the simplified telecommunication system, five components have two states and the remaining two components have three states. This means that the simplified telecommunication system has a total of $2^5 \times 3^2 = 288$ possible states, that is, the number of states in a system is exponential in the number of components. To alleviate this problem, OBDDs are used to encode all variable assignments, Σ, as well as transitions, τ, of a $\mathcal{CA}$. OBDDs provide two distinct benefits: (1) a compact state space encoding and (2) operators that allow a state space to be searched without the need to enumerate all of the states explicitly.

Encoding $\mathcal{CA}$ using OBDDs involves representing each finite domain variable in Π and the transition functions τ_i as OBDDs. In this section, OBDDs are briefly introduced for readers who are not familiar with this representation. Then, the method for encoding finite domain variables in OBDDs is presented, followed by the method for encoding the transitions. For a more detailed discussion of OBDDs, refer to Appendix A.

3.2.1 Ordered Binary Decision Diagram

An Ordered Binary Decision Diagram (OBDD) is a rooted, directed acyclic graph (DAG) representation of a Boolean function, where the set of Boolean variables are ordered sequentially. A Boolean function $f : \mathcal{B}^n \rightarrow \mathcal{B}$ is an expression formed with Boolean variables, and Boolean operators, including, but not limited to, negation $\neg$, conjunction $\wedge$, disjunction $\vee$, implication $\Rightarrow$, and equivalence $\Leftrightarrow$, where $\mathcal{B}$ is the Boolean domain

$\{true, false\}$. Equation 3.4 is an example of a Boolean function.

$$(x_1 \wedge y_1) \vee (x_2 \wedge y_2) \vee (x_3 \wedge y_3) \tag{3.4}$$

The corresponding OBDD with $x_1 \prec y_1 \prec x_2 \prec y_2 \prec x_3 \prec y_3$ ordering is shown in Figure 3-3(a). Each node of an OBDD represents a Boolean variable. The dotted and solid outgoing edges respectively represent $false$ and $true$ evaluations of the Boolean variable. The terminal nodes 1 and 0 represent the evaluation of the Boolean function (i.e., OBDD) where each path from the root to a terminal evaluates to 1 for $true$ or 0 for $false$. The OBDD in Figure 3-3(a) has a total of five different paths from the root to terminal 1:

$$\begin{aligned}
&\{x_1 = true, y_1 = true\} \\
&\{x_1 = true, y_1 = false, x_2 = true, y_2 = true\} \\
&\{x_1 = false, x_2 = true, y_2 = true\} \\
&\{x_1 = false, x_2 = true, y_2 = false, x_3 = ture, y_3 = ture\} \\
&\{x_1 = false, x_2 = false, x_3 = true, y_3 = true\}
\end{aligned} \tag{3.5}$$

The fact that $\{x_1 = true, y_1 = true\}$ terminates to 1 implies that as long as $x_1 = true$ and $y_1 = true$, Equation 3.4 evaluates to $true$ regardless of the values of x_2, x_3, y_2, and y_3.

The ordering of the variables is crucial to the compactness of an OBDD representation. For example, if the ordering of the same Boolean function in Equation 3.4 changes to $x_1 \prec x_2 \prec x_3 \prec y_1 \prec x_2 \prec y_3$, the size of the OBDD becomes considerably larger as shown in Figure 3-3. Unfortunately, determining an ordering that minimizes the size of the OBDD is a coNP-Complete problem [6]. However, ordering the "dependent" variables near each other is a good heuristic for reducing the size of an OBDD [14]. For example, $x_1 \prec y_1 \prec x_2 \prec y_2 \prec x_3 \prec y_3$ places the "dependent" variables near each other. That is, one possible solution of Equation 3.4 requires both x_1 and y_1 to be $true$, thus x_1 and y_1 are "dependent" on one another. Similarly, x_2 and y_2 are "dependent" on one another, and x_3 and y_3 are "dependent" on one another.

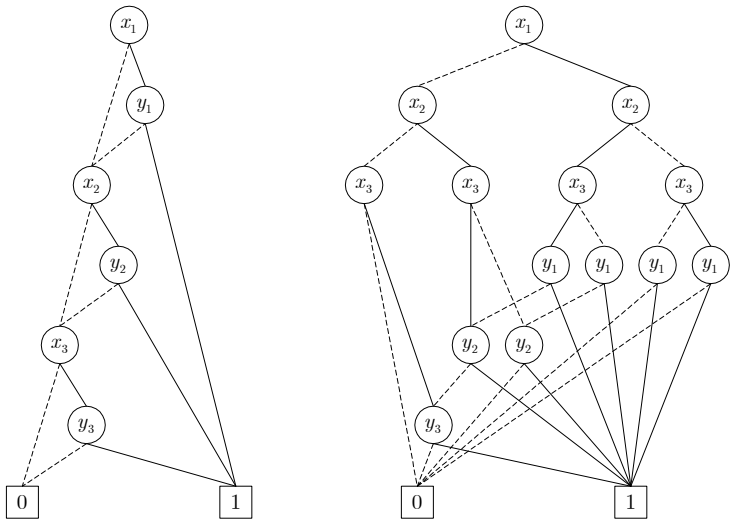

Figure 3-3: OBDD for $(x_1 \wedge y_1) \vee (x_2 \wedge y_2) \vee (x_3 \wedge y_3)$ with two different ordering sequences: (a) $x_1 \prec y_1 \prec x_2 \prec y_2 \prec x_3 \prec y_3$ and (b) $x_1 \prec x_2 \prec x_3 \prec y_1 \prec x_2 \prec y_3$.

Many graph-based algorithms have been developed that facilitate efficient "operation" on OBDDs [6, 7]. Algorithms exist for most commonly used logical operators and quantifiers, including $\neg$, $\wedge$, $\vee$, $\exists$, and $\forall$. These operators have also been extended for set operations, such as $\cup$, $\cap$, $\in$, $\subseteq$, and $-$ (i.e., set subtraction). For example, if X and Y are OBDD encodings of boolean functions representing sets $\mathcal{X}$ and $\mathcal{Y}$ respectively, $X \wedge Y$ is equivalent to $\mathcal{X} \cap \mathcal{Y}$. Similarly, $X \vee Y$ is equivalent to $\mathcal{X} \cup \mathcal{Y}$, $X \wedge \neg Y$ to $\mathcal{X} - \mathcal{Y}$, and $X \wedge \neg Y = false$ to $\mathcal{X} \subseteq \mathcal{Y}$. It should be noted that all of the aforementioned operators have linear or polynomial time complexity with respect to the number of Boolean variables used in the OBDDs.

3.2.2 Encoding Finite Domain Variables

Each finite domain variable in Π of a $\mathcal{CA}$ must be encoded as an OBDD. For variables with binary domains, this task is trivial. For example, consider Transmitter #1's state variable $T1$ where $\mathcal{D}(T1) = \{off, on\}$. $T1$ can be represented in an OBDD by simply using an OBDD Boolean variable whose $false$ and $true$ values correspond to *off* and *on*, respectively. In this case, variable $T1$ is overloaded to also denote an OBDD Boolean variable that represents $T1$.

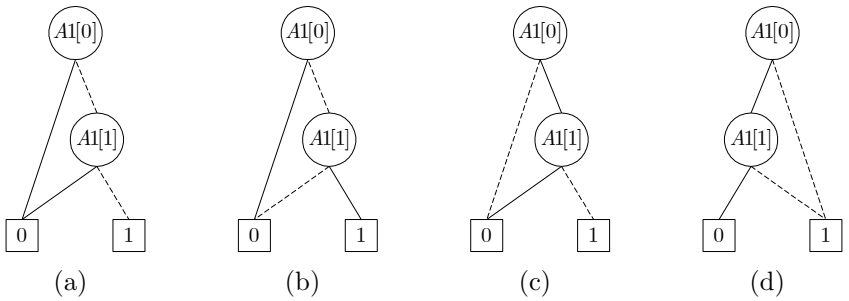

Figure 3-4: Two Boolean variables, $A1[0]$ and $A1[1]$, are used to represent Amplifier #1's states where: (a) $A1 = \langle 0,0 \rangle$ corresponds to $A1 = \mathit{off}$ (a) $A1 = \langle 0,1 \rangle$ corresponds to $A1 = \mathit{on}$, and (a) $A1 = \langle 1,0 \rangle$ corresponds to $A1 = \mathit{resettable}$. The set of feasible states $\{A1 = \langle 0,0 \rangle, A1 = \langle 0,1 \rangle, A1 = \langle 1,0 \rangle\}$ is represented by (d).

For variables with a domain larger than a binary domain, a vector of OBDD Boolean variables must be used. For example, consider state variable $A1$ of Amplifier #1, where $\mathcal{D}(A1) = \{\mathit{off}, \mathit{on}, \mathit{resettable}\}$. A 2-bit binary vector is necessary to represent $A1$ in OBDD. Again, overloading $A1$ to also denote a 2-bit binary vector, $A1 = \langle 0,0 \rangle$ corresponds to $A1 = \mathit{off}$, $A1 = \langle 0,1 \rangle$ corresponds to $A1 = \mathit{on}$, and $A1 = \langle 1,0 \rangle$ corresponds to $A1 = \mathit{resettable}$. With each bit, $A1[0]$ and $A1[1]$, represented as an OBDD Boolean variable, the OBDD encodings of Amplifier #1's states $\{A1 = \mathit{off}, A1 = \mathit{on}, A1 = \mathit{resettable}\}$ are shown in Figures 3-4(a), 3-4(b), and 3-4(c), respectively.

Note that $A1 = \langle 1,1 \rangle$ has no corresponding state. This knowledge is taken into account by representing the feasible state space as an OBDD. A set is encoded in an OBDD as a disjunction of its elements. In the case of Amplifier #1, the state space is encoded in an OBDD as $A1 = \langle 0,0 \rangle \vee A1 = \langle 0,1 \rangle \vee A1 = \langle 1,0 \rangle$. The OBDD of Amplifier #1's state space is illustrated in Figure 3-4(d).

In general, a finite domain variable x with $N = |\mathcal{D}(x)|$ values is represented by $n = \lceil \log_2 N \rceil$ OBDD Boolean variables, where each distinct n-bit binary vector corresponds to a value in $\mathcal{D}(x)$. While a set of variables are defined to represent the control variables in Π^c, two sets of variables must be defined for state variables in Π^s so that the state at time t can be distinguished from the state at time $t+1$. In the case of the state variable for Amplifier #1, the vector $A1$ denotes the state of the amplifier at current time t and $A1'$ denotes the state at time $t+1$.

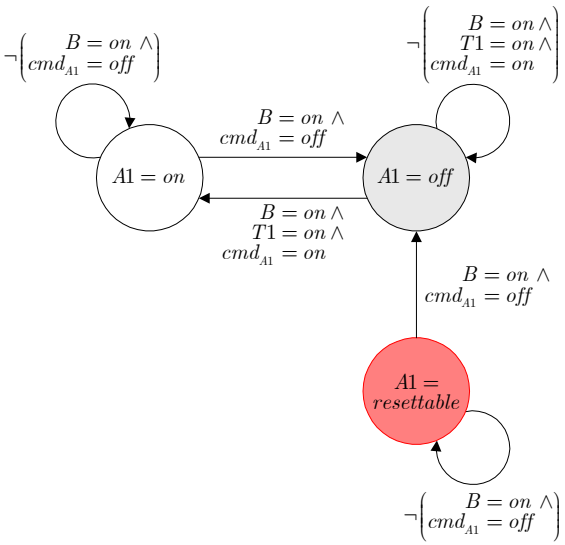

Figure 3-5: Amplifier #1 automaton. Nondeterministic fault transitions have been omitted for the purposes of reactive planning.

3.2.3 Encoding the Transition Function

For the purpose of reactive planning, all nondeterministic fault transitions are omitted from the model. In general, fault transitions are highly unlikely, and a plan that requires a highly improbable event to occur is futile. Moreover, purposely failing a system to achieve the desired goal state is unacceptable. In the case of Amplifier #1, the nondeterministic failure transition from *on* to *resettable* (i.e., the dashed arc in Figure 3-1) is omitted, resulting in the deterministic automaton shown in Figure 3-5. Thus, only the transition τ^n_{A1}, representing the nominal behavior of Amplifier #1, needs to be encoded in an OBDD.

The transition functions of all concurrent automata are stored as a set of OBDDs, $R = \{R_i | i = 1, 2, \ldots, n\}$, where each R_i is an OBDD encoding of the automaton $\mathcal{A}_i$'s transition function. A transition function of an automaton $\mathcal{A}_i$ is represented as a transition relation $R_i = \langle \Sigma_i^{s_i}, \mathbb{C}(\Pi_i), \Sigma_i^{s_i} \rangle$ using the characteristic function $R_i : \Sigma_i^{s_i} \times \mathbb{C}(\Pi_i) \times \Sigma_i^{s_i} \rightarrow \mathcal{B}$, where $\mathcal{B}$ is the Boolean domain. The relation is defined as $R_i(\sigma_i, c_i, \sigma_i') = (\sigma_i' \in \tau_i^n(\sigma_i, c_i))$, where σ_i' indicates the state at the next time step and $c_i \in \mathbb{C}(\Pi_i)$ denotes a constraint that is entailed in the current time step. For example, the transition relation R_{A1} of Amplifier #1 is specified using a standard encoding [31] as follows:

$$
\begin{aligned}
(A1 = \mathit{off} \wedge \neg (B = on \wedge T1 = on \wedge cmd_{A1} = on)) &\Rightarrow A1' = \mathit{off} \\
(A1 = \mathit{off} \wedge B = on \wedge T1 = on \wedge cmd_{A1} = on) &\Rightarrow A1' = on \\
(A1 = on \wedge \neg (B = on \wedge cmd_{A1} = \mathit{off})) &\Rightarrow A1' = on \\
(A1 = on \wedge B = on \wedge cmd_{A1} = \mathit{off}) &\Rightarrow A1' = \mathit{off} \\
(A1 = resettable \wedge \neg (B = on \wedge cmd_{A1} = \mathit{off})) &\Rightarrow A1' = resettable \\
(A1 = resettable \wedge (B = on \wedge cmd_{A1} = \mathit{off})) &\Rightarrow A1' = \mathit{off}
\end{aligned}
$$

In general, the transition relation is:

$$R_i = \Sigma_i^{s_i} \wedge \Sigma_i^c \wedge \Sigma_i^{s'_i} \wedge \bigwedge_{\sigma_i \in \Sigma_i^{s_i}} \bigwedge_{c_i \in \mathbb{C}(\Pi_i)} \sigma_i \wedge c_i \Rightarrow \sigma'_i \tag{3.6}$$

where $\sigma'_i \in \tau_i^n(\sigma_i, c_i)$. Each variable in the relation is represented with the appropriate OBDD Boolean variables, and $\tau_i^n(\sigma_i, c_i)$ is represented with the appropriate OBDD variables that denote the the next time step state. The conjunction of $\Sigma_i^{s_i}$, Σ_i^c, and $\Sigma_i^{s'_i}$ restricts the transition relation to only the feasible states and control assignments. All logical operators used in Equation 3.6, including those used in the constraints c_i, are computed using the corresponding OBDD operator.

Figure 3-6(a) illustrates the OBDD representation of the transition $(A1 = on \wedge B = on \wedge cmd_{A1} = \mathit{off}) \Rightarrow A1' = \mathit{off}$. Figure 3-6(b) shows the result of conjoining the OBDDs of the transitions into the transition relation R_{A1}[1]. In Figure 3-6(b), all paths that lead to $false$ have been omitted for simplicity. One of the benefits of using OBDDs to represent transition relations is their relative compactness. [12] shows that the size of an OBDD does not necessarily depend on the number of states, but rather on the structure of the information the OBDD encodes. As such, OBDD-based model checking and planning have been successful even for problems with large state space. However, the compactness of OBDDs is not guaranteed.

[1]In this example, we assume cmd_{T1} and cmd_{A1} domain is $\{\mathit{off}, on\}$. The *noCmd* has been removed only to simplify the example.

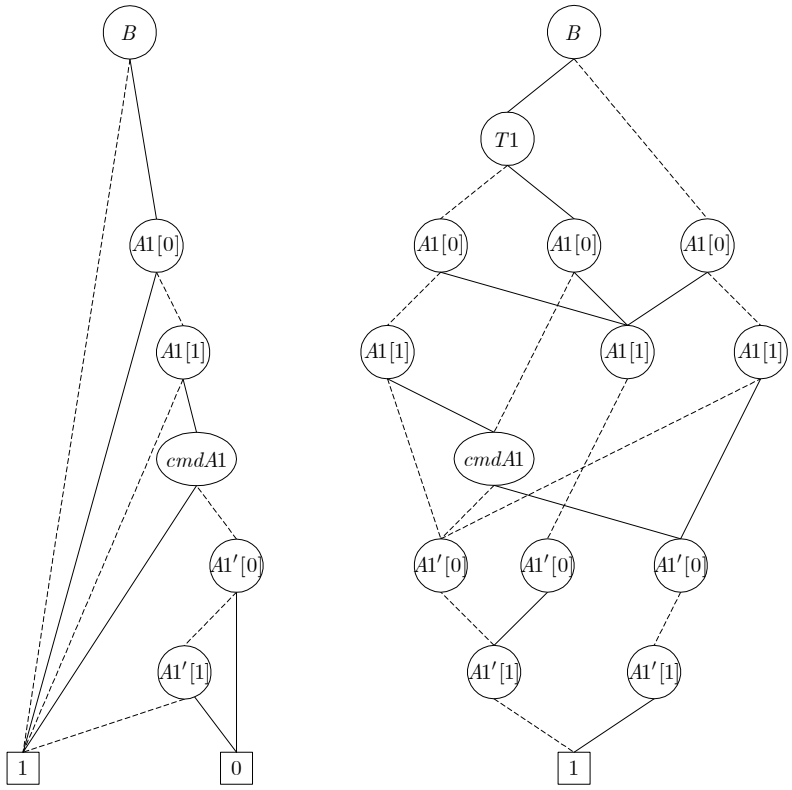

Figure 3-6: OBDD representation of (a) $(A1 = on \wedge B = on \wedge cmd_{A1} = \mathit{off}) \Rightarrow A1' = \mathit{off}$ and (b) Amplifier #1 transition relation R_{A1}. In (b), all paths to 0 terminal (i.e., $false$) have been omitted.

Chapter 4

Goal-directed Plans

Traditionally, a ground operator controls a spacecraft by uploading the necessary sequence of commands. However, this type of open-loop control lacks robustness to anomalies. Typically, a spacecraft manages anomalies using a rule-based fault protection system. However, as spacecraft become more complex in order to satisfy more ambitious mission requirements, hand coding robust recovery rules becomes more arduous and prone to error.

Use of a universal plan is an innovative approach to managing anomalies during execution [28]: a universal plan can provide robustness by specifying ways to achieve a goal from any spacecraft state, including fault states. However, a new universal plan must be computed on the fly for each new goal state received (i.e each reconfiguration request). Since computing a new plan is a PSAPCE-complete problem in general [9], this approach cannot guarantee hard real-time response to reconfiguration requests.

Goal-directed plans (GDP), a concept pioneered by Williams and Nayak [31], can be executed for both repair and reconfiguration. Much like a universal plan, a GDP accounts for possible anomalies during execution, i.e., performs repair. However, distinct from a universal plan, a GDP can be used for reconfiguration as well. In essence, a GDP compactly encodes a set of universal plans for all potential goal states. A GDP specifies the right action to take in all situations, conditioned on the specified goal state, thus the term "goal-directed".

Computing a GDP for concurrent automata can be challenging. Since a system is modelled as a set of synchronous and concurrently operating automata, the transition dependencies among automata must be taken into account, which complicates the planning

problem. One way to address this issue is to compose the set of concurrent automata into a single *composed automaton*, thus eliminating concurrency during planning.

In this chapter, a method for composing concurrent automata into a single composed automaton is introduced. Then, computing a GDP for the composed automaton is discussed. Finally, the OBDD encoding of a GDP is described, as an approach to mitigating the state space explosion problem.

4.1 Composing Concurrent Automata

In this section, the composition of concurrent automata into a single automaton is formally introduced, followed by a discussion of the implications of composing concurrent automata.

4.1.1 Composed Automaton

For n concurrent automata, the state space of the composed automaton $\Sigma_{\mathcal{CA}}^{s_{\mathcal{CA}}}$ is simply the cartesian product of each concurrent automaton's state space $\Sigma_i^{s_i}$, that is $\Sigma_{\mathcal{CA}}^{s_{\mathcal{CA}}} = \prod_{i=1}^{n} \Sigma_i^{s_i}$. Similarly, the initial state of the composed automaton $\sigma_{\mathcal{CA}}^{(0)}$ is the union of each concurrent automaton's initial state $\sigma_i^{(0)}$, that is $\sigma_{\mathcal{CA}}^{(0)} = \bigcup_{i=1}^{n} \sigma_i^{(0)}$. A transition $\tau_{\mathcal{CA}}(\sigma_{\mathcal{CA}}^{(t)}, c_{\mathcal{CA}})$ of the composed automaton represents a combination of each automaton's transition $\tau_i(\sigma_i^{(t)}, c_i)$, where the composed automaton state $\sigma_{\mathcal{CA}}^{(t)}$ represents each automaton's state $\sigma_i^{(t)}$ and the constraint $c_{\mathcal{CA}}$ is the logical conjunction of each c_i. Given a state $\sigma_{\mathcal{CA}}^{(t)}$ and a control action $\mu_{\mathcal{CA}}^{(t)}$ at time t, a transition $\tau_{\mathcal{CA}}(\sigma_{\mathcal{CA}}^{(t)}, c_{\mathcal{CA}})$ is enabled at time $(t+1)$ if and only if the constraint $c_{\mathcal{CA}}$ is entailed by $\sigma_{\mathcal{CA}}^{(t)}$ and $\mu_{\mathcal{CA}}^{(t)}$. Formally, a composed automaton $\mathcal{A}_{\mathcal{CA}}$ is defined as follows:

Definition 4.1. Given concurrent automata $\mathcal{CA} = \langle \mathcal{A}, \Pi, \Sigma \rangle$ where $\mathcal{A} = \{\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_m\}$, a *composed automaton* $\mathcal{A}_{\mathcal{CA}}$ is a 4-tuple $\langle \Pi_{\mathcal{CA}}, \Sigma_{\mathcal{CA}}, \tau_{\mathcal{CA}}, \sigma_{\mathcal{CA}}^{(0)} \rangle$, where:

1. $\Pi_{\mathcal{CA}} = \Pi$ is a finite set of variables for $\mathcal{CA}$. $\Pi_{\mathcal{CA}} = \Pi_{\mathcal{CA}}^{s} \cup \Pi_{\mathcal{CA}}^{c}$ is partitioned into a set of state variables $\Pi_{\mathcal{CA}}^{s}$ and a set of control variables $\Pi_{\mathcal{CA}}^{c}$.

2. $\Sigma_{\mathcal{CA}} = \Sigma$ is a finite set of full assignments over $\Pi_{\mathcal{CA}}$. $\Sigma_{\mathcal{CA}} = \Sigma^s_{\mathcal{CA}} \cup \Sigma^c_{\mathcal{CA}}$ is also partitioned into a set of assignments to state variables $\Sigma^s_{\mathcal{CA}}$ and a set of assignments to control variables $\Sigma^c_{\mathcal{CA}}$. For a composed automata, the state space $\Sigma^{s_{\mathcal{CA}}}_{\mathcal{CA}}$ is equivalent to $\Sigma^s_{\mathcal{CA}}$.

3. $\tau_{\mathcal{CA}}(\sigma_{\mathcal{CA}}, c_{\mathcal{CA}}) = \prod_{i=1}^{m} \tau_i(\sigma_i, c_i)$ is the transition function of the composed automaton, where $\tau_i(\sigma_i, c_i)$ is the transition of an automaton $\mathcal{A}_i \in \mathcal{A}$. $\sigma_{\mathcal{CA}} = \bigcup_{i=1}^{m} \sigma_i$ is the state of the composed automaton, where σ_i is a state of automaton $\mathcal{A}_i \in \mathcal{A}$. The transition conditions on the constraint $c_{\mathcal{CA}} = \bigwedge_{i=1}^{m} c_i$. Given a state $\sigma^{(t)}_{\mathcal{CA}}$ and a control action $\mu^{(t)}_{\mathcal{CA}}$ at time t, a transition $\tau_{\mathcal{CA}}(\sigma^{(t)}_{\mathcal{CA}}, c_{\mathcal{CA}})$ is enabled at time $(t+1)$ if and only if the constraint $c_{\mathcal{CA}} \in \mathbb{C}(\Pi_{\mathcal{CA}})$ is entailed by $\sigma^{(t)}_{\mathcal{CA}}$ and $\mu^{(t)}_{\mathcal{CA}}$, where $\mathbb{C}(\Pi_{\mathcal{CA}})$ is a set of all finite domain constraints over $\Pi_{\mathcal{CA}}$.

4. $\sigma^{(0)}_{\mathcal{CA}} = \bigcup_{i=1}^{m} \sigma^{(0)}_i$ is the initial state of the composed automaton, where $\sigma^{(0)}_i \in \Sigma^{s_i}_i$ is the initial state of automaton $\mathcal{A}_i \in \mathcal{A}$.

In reactive planning, only the transitions that represent the nominal behavior (i.e., nominal transitions) is considered. A nominal transition of the composed automaton is simply $\tau^n_{\mathcal{CA}}(\sigma_{\mathcal{CA}}, c_{\mathcal{CA}}) = \bigcup_{i=1}^{m} \tau^n_i(\sigma_i, c_i)$, where $\tau^n_i(\sigma_i, c_i)$ is a nominal transition of an automaton $\mathcal{A}_i \in \mathcal{A}$.

This composition operation is straight forward using the OBDD representation of concurrent automata. The state space of the composed automaton is simply a conjunction of each automaton's state space:

$$\Sigma^{s_{\mathcal{CA}}}_{\mathcal{CA}} = \bigwedge_{i=1}^{m} \Sigma^{s_i}_i \tag{4.1}$$

where $\Sigma^{s_i}_i$ is an OBDD-encoded state space of automaton $\mathcal{A}_i \in \mathcal{A}$. Similarly, a set of assignments to control variables is:

$$\Sigma^c_{\mathcal{CA}} = \bigwedge_{i=1}^{m} \Sigma^c_i \tag{4.2}$$

where Σ^c_i is an OBDD-encoded set of assignments to control variables of automaton $\mathcal{A}_i \in \mathcal{A}$. The transition relation for the composed automaton is a conjunction of the transition

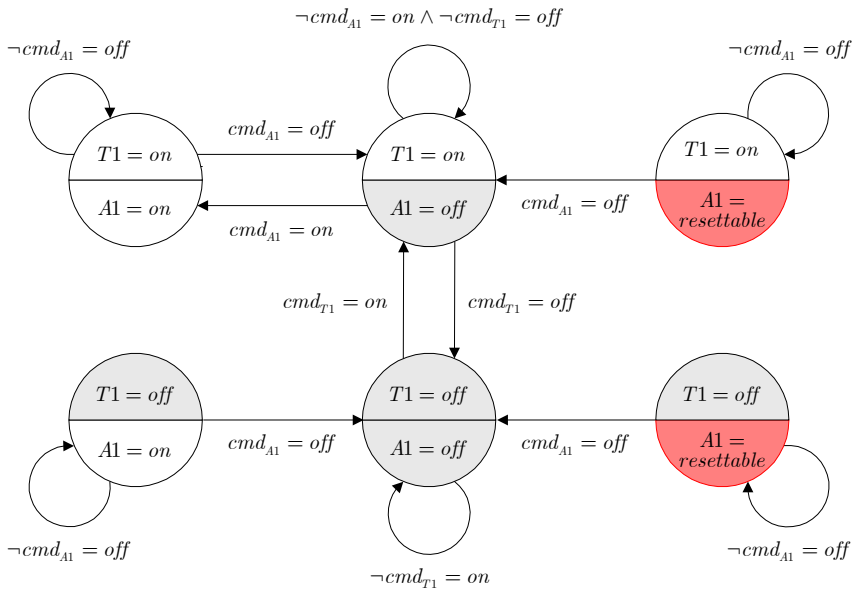

Figure 4-1: Composed automaton representing a system consisting of Transmitter #1 and Amplifier #1. The Bus Controller was removed from the system by assuming that it is always *on*.

relations for the concurrent automata. That is, given a set of transition relations $R = \{R_1, R_2, \ldots, R_m\}$ of the concurrent automata $\mathcal{CA}$, the transition relation $R_{\mathcal{CA}}$ of the composed automaton is:

$$R_{\mathcal{CA}} = \bigwedge_{i=1}^{m} R_i \tag{4.3}$$

4.1.2 Implementing Concurrency via Interleaving

For example, consider the system consisting of Transmitter #1 and Amplifier #1. Concurrent automata for these two components are illustrated in Figures 2-5 and 2-6 (pages 32, 34). Also, for simplicity, assume that the Bus Controller is always *on*. Thus, $B = on$ is removed from all transition conditions of Transmitter #1 and Amplifier #1. The composed automaton that represents this system is shown in Figure 4-1.

Notice that one transition seems missing, the transition from $\{T1 = on, A1 = off\}$ to $\{T1 = off, A1 = on\}$. According to the model shown in Figures 2-5 and 2-6, such transition may occur if and only if Transmitter #1 is commanded off ($cmd_{T1} = off$) and the Amplifier #1 is commanded on ($cmd_{A1} = on$) simultaneously.

This ability to execute multiple commands simultaneously requires a synchronous system. While a concurrent automata is assumed synchronous in that each automaton performs a single state transition at each time step, the concurrent automata are not assumed to transition simultaneously within a time step. Instead, the transitions of the concurrent automata are assumed interleaved. This model of concurrency via interleaving is assumed for two reasons. First, a spacecraft typically consists of a single main processor, which executes synchronous activities by interleaving them. Two, for asynchronous systems, as is the case with many spacecraft components, the interleaved model is more robust.

With concurrency modelled via interleaving, issuing the command $cmd_{T1} = \mathit{off}$ and $cmd_{A1} = on$ at exactly the same time cannot be guaranteed. In fact, taking such action could be hazardous: the amplifier could be damaged if $cmd_{A1} = on$ precedes $cmd_{T1} = \mathit{off}$ even by a fraction of a second.

To ensure safe and proper execution, the interleaved transitions must not interfere with one another nor compete for mutually exclusive needs, as described by the mutual exclusion rule of Graphplan [5]. The composition of the OBDD-encoded transition relations resolves all mutual exclusion rules except for one interference problem, that is, assuring that the effect of one transition does not remove the needed precondition of another transition. For example, turning Transmitter #1 off requires Amplifier #1 *off*, but commanding $cmd_{A1} = on$ will turn Amplifier #1 on, an interference.

To resolve this interference issue, the transition relation of each automaton must be modified. If a transition is conditioned on the state of another automaton, the transition is constrained so that the state condition must be true both before and after the transition occurs. For example, consider the amplifier's transition from *off* to *on*:

$$(A1 = \mathit{off} \wedge T1 = on \wedge cmd_{A1} = on) \Rightarrow A1' = on \tag{4.4}$$

where the variable $A1'$ indicates the amplifier state at the next time step. The transition relies on the transmitter being *on* ($T1 = on$). Thus, the transition is modified to

guarantee that the transmitter is on before and after:

$$(A1 = \mathit{off} \wedge T1 = on \wedge T1' = on \wedge cmd_{A1} = on) \Rightarrow A1' = on. \tag{4.5}$$

Similarly, the transmitter's transition

$$(T1 = on \wedge A1 = \mathit{off} \wedge cmd_{T1} = \mathit{off}) \Rightarrow T1' = \mathit{off} \tag{4.6}$$

is modified to

$$(T1 = on \wedge A1 = \mathit{off} \wedge A1' = \mathit{off} \wedge cmd_{T1} = \mathit{off}) \Rightarrow T1' = \mathit{off}. \tag{4.7}$$

When the transitions in Equations 4.5 and 4.7 are logically conjoined, the transitions they represent cannot occur simultaneously. Thus, the transition from $\{T1 = on, A1 = \mathit{off}\}$ to $\{T1 = \mathit{off}, A1 = on\}$ is eliminated from the composed automaton (see Figure 4-1).

4.1.3 Size of Composed Transition Relations

In general, when composing automata, the concern is the size of the OBDD representing the composed automaton. For example, while the composed automaton for the Transmitter #1/Amplifier #1 system has a total of six possible states, a total of 22 nodes are necessary to encode the composed automaton's transition relation in an OBDD. Recall that Figure 3-6(b) showed that for Amplifier #1 alone, which has a total of three possible states, the OBDD-encoded transition relation requires 14 nodes, excluding the 0 and 1 terminals.

Figure 4-2 shows systems of various state space size and their corresponding OBDD transition relation sizes. The number of states in the system was increased by adding additional components. See Section 7.2 (page 88) for more details. The system with 288 states represents the simplified telecommunication system introduced in Chapter 3. The simplified telecommunication system's composed transition relation encoded as an OBDD required a total of 68 OBDD nodes. The composed transition relation of the full MESSENGER telecommunication system with only two antennas had a state space size of 18432 and required a total of 108 OBDD nodes. As the trend in Figure 4-2 suggests,

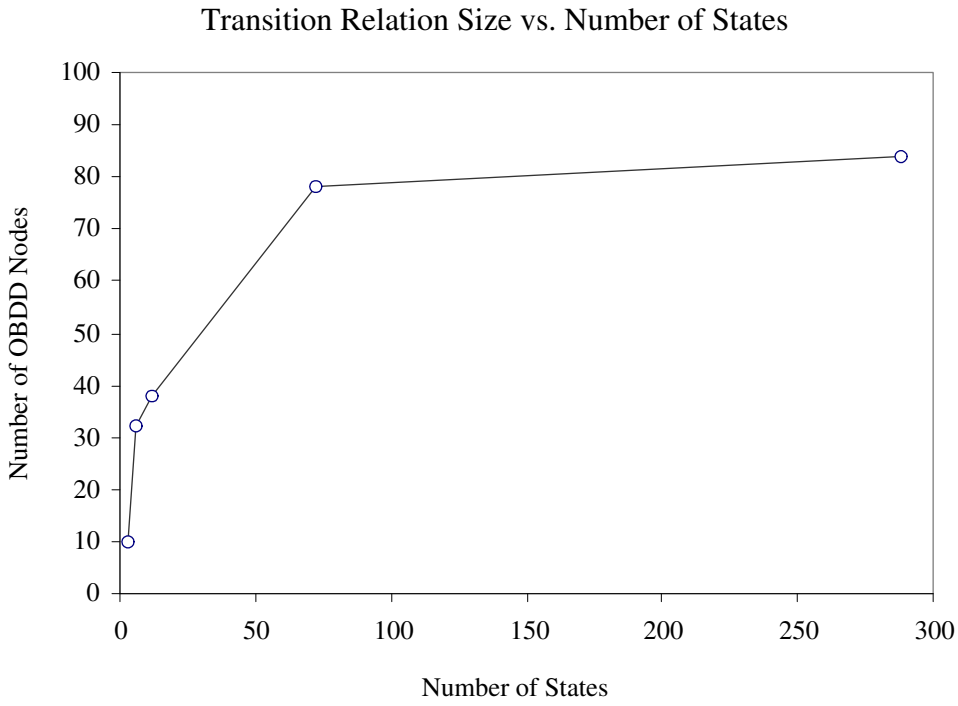

Figure 4-2: Size of OBDD encoded transition relations relative to the size of the state space.

the size of the OBDD-encoded transition relation does not explode along with the state space of the system. This trend is similar to the result shown in [12], in which the growth of a universal plan encoded in an OBDD is shown to be small with respect to the growth of the state space of the problem.

4.2 Goal-directed Plan

Recall that GDPs include plans that can react to all possible initial states and goals, and the central focus of this thesis is in compactly encoding the GDPs.

A GDP is comprised of a set of goal-directed rules, where a goal-directed rule is a 3-tuple $\langle \sigma, \mu, \sigma' \rangle$. A goal-directed rule can be interpreted as "if the current state is σ and the goal state is σ', execute μ", i.e., $\langle \sigma, \sigma' \rangle \Rightarrow \mu$.

Figure 4-3 is a tabular representation of the goal-directed plan for the Transmitter #1/Amplifier #1 system. Each entry in the table corresponds to a goal-directed rule.

Current State	Goal State			
	$T1 = on,\ A1 = on$	$T1 = on,\ A1 = \mathit{off}$	$T1 = \mathit{off},\ A1 = \mathit{off}$	$T1 = \mathit{off},\ A1 = on$
$T1 = on,\ A1 = on$	*idle*	$cmd_{A1} = \mathit{off}$ (1)	$cmd_{A1} = \mathit{off}$ (2)	*failure*
$T1 = on,\ A1 = \mathit{off}$	$cmd_{A1} = on$ (1)	*idle*	$cmd_{T1} = \mathit{off}$ (1)	*failure*
$T1 = on,\ A1 = resettable$	$cmd_{A1} = \mathit{off}$ (2)	$cmd_{A1} = \mathit{off}$ (1)	$cmd_{T1} = \mathit{off}$ (2)	*failure*
$T1 = \mathit{off},\ A1 = \mathit{off}$	$cmd_{T1} = on$ (2)	$cmd_{T1} = on$ (1)	*idle*	*failure*
$T1 = \mathit{off},\ A1 = on$	$cmd_{A1} = \mathit{off}$ (3)	$cmd_{A1} = \mathit{off}$ (2)	$cmd_{A1} = \mathit{off}$ (1)	*idle*
$T1 = \mathit{off},\ A1 = resettable$	$cmd_{A1} = \mathit{off}$ (3)	$cmd_{A1} = \mathit{off}$ (2)	$cmd_{A1} = \mathit{off}$ (1)	*failure*

Figure 4-3: Goal-directed plan for the Transmitter #1/Amplifier #1 system. The number next to each command represents the total number of steps necessary to achieve the goal. Similar to the goal state $\{T1 = \mathit{off}, A1 = on\}$, neither the goal state $\{T1 = on, A1 = resettable\}$ nor $\{T1 = \mathit{off}, A1 = resettable\}$ can be reached from any other states.

For example, when the system is off, i.e., $\{T1 = \mathit{off}, A1 = \mathit{off}\}$, the goal-directed rule for the goal state $\{T1 = on, A1 = on\}$ is:

$$\langle\{T1 = \mathit{off}, A1 = \mathit{off}\}, \{cmd_{T1} = on\}, \{T1 = on, A1 = on\}\rangle. \tag{4.8}$$

Upon execution of this rule, issuing command $cmd_{T1} = on$[1], the system's next will be $\{T1 = on, A1 = \mathit{off}\}$ under a nominal situation (i.e., no occurrence of a fault). Then, from the state $\{T1 = on, A1 = \mathit{off}\}$, the goal-directed rule

$$\langle\{T1 = on, A1 = \mathit{off}\}, \{cmd_{A1} = on\}, \{T1 = on, A1 = on\}\rangle \tag{4.9}$$

should be executed (i.e., issue the command $cmd_{A1} = on$). Under a nominal execution of the rule, the desired goal state is finally achieved. If the system fails to a repairable fault state (for example, $\{T1 = on, A1 = resettable\}$), the execution of the GDP will naturally repair the system back to the desired goal-state. This is similar to the behavior of a universal plan or a Markov decision process policy. Recall that the difference is that a GDP includes plans for all goal states. This enables a robust reactive execution with both repair and reconfiguration capabilities and fast online response.

Although no plan can guarantee to achieve a goal under nondeterministic failures, a plan must be guaranteed to achieve a goal under nominal execution:

[1]The control action $cmd_{T1} = on$ implies $cmd_{A1} = noCmd$, that is "no command". In general, no explicit assignment to a control variable implies the values $noCmd$.

Definition 4.2. Given a composed automaton $\mathcal{A} = \langle \Pi, \Sigma, \tau, \sigma^{(0)} \rangle$[2], a goal-directed rule $\langle \sigma, \mu, \sigma' \rangle$ is *valid* if and only if $\mu \in \Sigma^c$ is guaranteed to progress the automaton toward $\sigma' \in \Sigma^s$ under nominal execution.

As long as the goal-directed rules are valid and the system behaves nominally, the system is guaranteed to reach the goal in some finite sequence of GDP execution.

4.2.1 Generating the Goal-directed Plan

A GDP is generated by iteratively searching the state space in (1) parallel, (2) backward, and (3) breadth-first manner. (1) Using an OBDD encoding, states within the search space do not have to be explicitly enumerated. Goal-directed rules are generated for all goals and initial states simultaneously using a compact analytical encoding, thus a "parallel search". This is one of the key advantages in using the OBDD encoding. (2) The search method is also characterized as a "backward search", as the GDP is generated by searching for the states that can reach the goal, instead of searching for the goals that can be reached from the current state. (3) In the process of generating the goal-directed rules, the one-step rules, that is, rules that achieve the goal after a single transition, are generated first. In Figure 4-3, one-step rules are those with "(1)" next to the commands. For example, the goal-directed rule

$$\langle \{T1 = on, A1 = \mathit{off}\}, \{cmd_{A1} = on\}, \{T1 = on, A1 = on\} \rangle \tag{4.10}$$

is a one-step rule. Notice that one-step rules correspond directly to the transitions $\langle \sigma, \mu, \sigma' \rangle$ in the transition relation. Next, the two-step rules, labelled "(2)" in Figure 4-3, are generated. For example, the goal-directed rule

$$\langle \{T1 = \mathit{off}, A1 = \mathit{off}\}, \{cmd_{T1} = on\}, \{T1 = on, A1 = on\} \rangle \tag{4.11}$$

is a two-step rule. This process continues sequentially until the fixed-point is reached, thus "breadth-first". A fixed-point is reached if no new goal-directed rules exist. In

[2]The subscript $\mathcal{CA}$ that indicates composed automaton has been omitted for simplicity.

the Transmitter #1/Amplifier #1 system example, the fixed-point is reached after two iterations (i.e., after the three-step rules are generated).

The ComputeGDP algorithm for generating a GDP is shown as Algorithm 4.1. The ComputeGDP algorithm takes the transition relation R and the set of assignments to all control variables Σ^c of an automaton as its input. As discussed earlier, goal-directed rules are searched iteratively in breadth-first order. An OBDD called *oldPlan* is initially empty as reflected in line 1 of ComputeGDP. *oldPlan* stores the goal-directed rules found in the previous iteration. An OBDD *newPlan* stores all goal-directed rules found up to the current iteration. Again, the one-step rules correspond directly to the transition relation (line 2). In lines 3–5, it iteratively searches for two-step rules, three-step rules, etc. while adding them to the *newPlan*. The procedure exits once the fixed-point is reached (line 3), and returns the plan (line 6).

Algorithm 4.1 ComputeGDP(R, Σ^c)

1: $oldPlan \leftarrow \emptyset$
2: $newPlan \leftarrow R$
3: **while** $oldPlan \neq newPlan$ **do**
4: $\quad oldPlan \leftarrow newPlan$
5: $\quad newPlan \leftarrow oldPlan \cup$ GenerateNextStepRules$(R, \Sigma^c, oldPlan)$
6: **return** $newPlan$

In line 5, GenerateNextStepRules$(R, \Sigma^c, oldPlan)$ generates all n-step rules, given all rules of less than n steps. The argument R is the transition relation, Σ^c is the set of control actions, and *oldPlan* is a set of all rules of less that n steps. Assume that $\langle \sigma^i, \mu^j, \sigma^{k\prime} \rangle$ is an element of the transition relation R and that an $(n-1)$-step rule $\langle \sigma^k, \mu^l, \sigma^{m\prime} \rangle$ is an element of the old goal-directed plan *oldPlan*. Then, $\langle \sigma^i, \mu^j, \sigma^{m\prime} \rangle$, returned by GenerateNextStepRules(R, Σ^c, P), is one of the valid n-step rules. For example, given the 1-step rule

$$\langle \{T1 = on, A1 = \mathit{off}\}, \{cmd_{T1} = \mathit{off}\}, \{T1' = \mathit{off}, A1' = \mathit{off}\} \rangle \tag{4.12}$$

and the transition relation

$$\langle\{T1 = on, A1 = on\}, \{cmd_{A1} = \mathit{off}\}, \{T1' = on, A1' = \mathit{off}\}\rangle \tag{4.13}$$

GENERATENEXTSTEPRULES produces the 2-step rule

$$\langle\{T1 = on, A1 = on\}, \{cmd_{A1} = \mathit{off}\}, \{T1' = \mathit{off}, A1' = \mathit{off}\}\rangle \tag{4.14}$$

.

Formally, GENERATENEXTSTEPRULES(R, Σ^c, P) generates a set of n-step goal-directed rules $\langle\sigma, \mu, \sigma'\rangle$, where R is a transition relation and P is a goal-directed plan containing all m-step rules, for $m = 1, 2, \ldots, (n-1)$. Each rule $\langle\sigma^i, \mu^j, \sigma^{k'}\rangle$ is restricted such that $\sigma^{l'} \subseteq (R \wedge \sigma^i \wedge \mu^j)$, $\exists(\mu \in \Sigma^c).\langle\sigma^l, \mu, \sigma^{k'}\rangle \in P$, and $\neg\exists(\mu \in \Sigma^c).\langle\sigma^i, \mu, \sigma^{k'}\rangle \in P$.

$\sigma^{l'} \subseteq (R \wedge \sigma^i \wedge \mu^j)$ states that given input μ^j, $\sigma^{l'}$ must be reachable from state σ^i in a single transition. $\exists(\mu \in \Sigma^c).\langle\sigma^l, \mu, \sigma^{k'}\rangle \in P$ states that some goal-directed rule for current state σ^l and goal state σ^k must exist in the plan P. The restriction $\neg\exists(\mu \in \Sigma^c).\langle\sigma^i, \mu, \sigma^{k'}\rangle \in P$ says that $\langle\sigma^i, \mu^j, \sigma^{k'}\rangle$ cannot be a new goal-directed rule if a rule for the current state σ^i and the goal state $\sigma^{k'}$ already exists in the plan P. With this restriction, the resulting GDP is guaranteed to be optimal, where an optimal plan is defined as a plan with the shortest control sequence. For example, while

$$\langle\{T1 = on, A1 = \mathit{off}\}, \{cmd_{T1} = \mathit{off}\}, \{T1' = on, A1' = on\}\rangle \tag{4.15}$$

is a 3-step rule, it is not included in the GDP, since the optimal 1-step rule already exists in the plan:

$$\langle\{T1 = on, A1 = \mathit{off}\}, \{cmd_{A1} = on\}, \{T1' = on, A1' = on\}\rangle \tag{4.16}$$

In fact, the 3-step rule is not even a "valid" rule, since the rule commands the system further away from the desired goal-state.

The algorithm for GENERATENEXTSTEPRULES(R, Σ^c, P) is given as Algorithm 4.2. This algorithm leverages the OBDD representation to efficiently search the state space, without enumeration.

Algorithm 4.2 GENERATENEXTSTEPRULES(R, Σ^c, P)

1: $nextPlan \leftarrow R_{[\sigma^{temp}/\sigma']} \wedge \exists\Sigma^c.P_{[\sigma^{temp}/\sigma]}$
2: $optimalNextPlanWithNoCmd \leftarrow \exists\Sigma^c.nextPlan - \exists\Sigma^c.P$
3: **return** $(nextPlan \wedge optimalNextPlanWithNoCmd)$

Line 1 of GENERATENEXTSTEPRULES(R, Σ^c, P) computes all next step goal-directed rules including the non-optimal ones[3]. Line 2 determines which rules are the optimal rules. Finally, line 3 returns only those rules that are valid. Note that all next step rules are computed simultaneously, thus "parallel".

4.2.2 Goal-directed Plan Execution

As illustrated in the beginning of this section, executing a goal-directed plan is, in essence, a matter of a simple lookup for the appropriate goal-directed rule. Given the current state σ and the goal state σ', the goal-directed rule $\langle\sigma, \mu, \sigma'\rangle$ must be isolated from the GDP. Upon identification of the rule, the command μ is executed. The algorithm for executing a GDP is shown in Algorithm 4.3.

Algorithm 4.3 EXECUTEGDP(P, σ, σ')

1: **if** $\sigma = \sigma'$ **then**
2: **return** $(success)$
3: **else**
4: $rule = P \wedge \sigma \wedge \sigma'$
5: **if** $rule \neq false$ **then**
6: **return** $\exists\sigma, \sigma'.rule$
7: **else**
8: **return** $failure$

The EXECUTEGDP algorithm takes a GDP P, the current state σ, and the goal state σ'. First, if the current state σ is the same as the goal state σ' (line 1), then the algorithm returns *success* (line 2). Otherwise, the conjunction $P \wedge \sigma \wedge \sigma'$ identifies the correct goal-directed rule $\langle\sigma, \mu, \sigma'\rangle$ (line 4). If the rule for the specified current state and goal state exists (line 5), then the existential quantification of the goal-directed rule over the

[3] $[\sigma^{temp}/\sigma]$ symbolizes the replacement of variable σ with variable σ^{temp}.

current state σ and the goal state σ' extracts the control action and it is returned (line 6). Otherwise, no plan exists that can achieve the goal. Thus, $failure$ is returned by `ExecuteGDP` algorithm (line 8).

Chapter 5

Decomposed Goal-directed Planning

A goal-directed plan (GDP), as introduced in Chapter 4, is in essence a set of goal-directed rules that map a current state and a goal state to an action that guarantees progress of the system toward the goal state. As the transition dependencies among concurrent automata complicates the planning problem, the concurrent automata are composed into a single automaton. However, the number of states of a composed automaton grows exponentially with respect to the number of concurrent automata (i.e., the number of modelled components). Thus, the number of goal-directed rules in a GDP is also exponential in the number of concurrent automata. This exponential growth is mitigated somewhat by encoding a GDP compactly as an OBDD; however, no guarantees can be made about the compactness of the OBDD-encoded GDPs. In the worst case, OBDD-encoded GDPs can also grow exponentially with respect to the number of concurrent automata. Thus, generating a GDP for a composed automaton is intractable in general.

A more tractable approach to reactive planning is the divide-and-conquer approach. If a problem can be divided into a set of subproblems, and the size of the largest subproblem is bounded, then the difficulty of the full problem is bounded by the difficulty of the largest subproblem. Such a divide-and-conquer approach can be applied to reactive planning, in which concurrent automata $\mathcal{CA}$ are decomposed into a set of sub-automata that define the subproblems. The set of goal-directed plans for the individual sub-automata is called a decomposed goal-directed plan (DGDP). The DGDP can be executed to control the entire system. This "decomposed" approach eliminates the exponential explosion problem of reactive planning.

In this chapter, the method for decomposing concurrent automata is first introduced. Then, computing DGDP on the decomposed automata is discussed.

5.1 Decomposing the System

Decomposition of $\mathcal{CA}$ is based on *subgoal serializability*, where a set of subgoals are serializable if and only if the goal can be partitioned into a set of subgoals that can be solved sequentially to achieve the goal [23]. As will be discussed in the following sections, the absence of component "interdependence", described in Section 2.2 (page 30), can lead to a simple goal serialization method. If no interdependence exists among the components of a system, the decomposition of $\mathcal{CA}$ is trivial, since the set of individual automata in $\mathcal{CA}$ is itself the decomposed automata. This decomposition can also be generalized for systems with interdependencies.

In this section the notion of subgoal serialization is introduced through an example. Then, a method for determining the subgoal serialization is described using a graph called the *transition dependency graph* [31]. Finally, the decomposition of $\mathcal{CA}$ is discussed in the context of subgoal serialization.

5.1.1 Serializable Subgoals: Example

Consider Bus Controller and Switch #1A illustrated in Figures 5-1(a) and 5-1(b), respectively. As described earlier, one of the roles of a bus controller is to route the commands from the computer to the intended devices. Switch #1A is one of the three switches in the RF Switch Assembly #1 that routes the signal to either one of the two lowgain antennas (LGA) attached to the RF Switch Assembly #1 (see Figure 2-1, page 27). Switch #1A routes the signal to the LGA pointing in the -Y direction if it is in position 1, $pos1$, and it routes the signal to the LGA pointing in the -Z direction if it is in position 2, $pos2$. As shown in Figure 5-1(b), Switch #1A can be commanded from $pos1$ to $pos2$ and vice versa if and only if the bus controller is *on*.

Presume that the current state of the Bus Controller/Switch #1A system is $\{B = \mathit{off}, S1A = pos1\}$ and the goal state to achieve is $\{B = \mathit{off}, S1A = pos2\}$. In this case, it

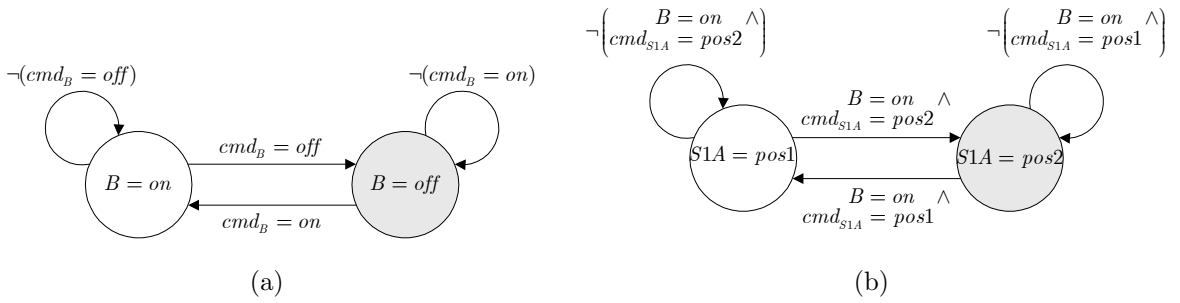

Figure 5-1: Concurrent automata of (a) Bus Controller and (b) Switch #1A.

is not necessary to compute a single plan that achieves both $B = \mathit{off}$ and $S1A = pos2$. Rather, the goal can be partitioned into two subgoals: $S1A = pos2$ and $B = \mathit{off}$. Once $S1A = pos2$ has been achieved, $B = \mathit{off}$ can be achieved without worrying about affecting the already achieved $S1A = pos2$ subgoal. That is, the $S1A = pos2$ subgoal is achieved by first turning the Bus Controller on. Once the Bus Controller is *on*, the switch position can be commanded to *pos*2. As the process of achieving the subgoal $S1A = pos2$ leaves the Bus Controller *on*, an essential side-effect, the Bus Controller must be commanded off to achieve the second subgoal ($B = \mathit{off}$). Achieving the second subgoal has no consequences on the first subgoal that has already been achieved. Hence, the subgoals are serializable. Note that "serialization" implies an ordering. That is, the subgoal must be achieved in the specified order. Attempting to achieve the subgoals in the reverse order (i.e., $B = \mathit{off}$ and then $S1A = pos2$) will not accomplish the intended goal $\{B = \mathit{off}, S1A = pos2\}$. In particular, after achieving the subgoal $B = \mathit{off}$, which requires no action, the bus is turned on during the process of achieving $S1A = pos1$. The resulting state $\{B = on, S1A = pos2\}$ is not the desired goal state $\{B = \mathit{off}, S1A = pos2\}$.

5.1.2 Subgoal Serialization

For the Bus Controller/Switch #1A system, the execution order of the subgoals is actually independent of the goal specified. In other words, regardless of the goal, as long as the subgoal associated with Switch #1A is achieved first, followed by the subgoal associated with the Bus Controller, the desired goal state can be attained. The subgoal ordering is independent of a goal due to the acyclicity of the system's *transition dependency graph*

(TDG). That is, no "interdependency" exists among the components. The TDG of $\mathcal{CA}$ is defined formally as follows:

Definition 5.1. The *transition dependency graph* $G = \langle V, E \rangle$ of concurrent automata $\mathcal{CA} = \langle \mathcal{A}, \Pi, \Sigma \rangle$ is a directed graph, where each vertex $v \in V$ corresponds to an automaton $\mathcal{A}_v \in \mathcal{A}$. Given a vertex $u \in V$ corresponding to the automaton $\mathcal{A}_u \in \mathcal{A}$ and a vertex $v \in V$ corresponding to the automaton $\mathcal{A}_v \in \mathcal{A}$, G contains a directed edge from vertex u to vertex v (i.e., $(u, v) \in E$) if and only if the state of automaton $\mathcal{A}_u$ is referred to by one of the transition conditions of $\mathcal{A}_v$.

An acyclic TDG defines the hierarchy of the concurrent automata. That is, the transitions of an automaton represented by vertex v are conditioned on the states of the automata that correspond to the ancestors of vertex v. Conversely, the transitions of an automaton represented by vertex v are not conditioned on the states of the automata that correspond to v's descendants. Thus, if each subgoal is associated with an automaton and each automaton with a vertex in the TDG, each successive achievement of subgoals is guaranteed not to interfere with the previously achieved subgoals, as long as the subgoals are achieved in the depth-first order of the TDG. This depth-first order is also equivalent to the inverse topological order of an acyclic TDG [15]. Williams and Nayak first recognized and exploited this relationship between the topological order of an acyclic TDG and the ordering that serializes the subgoals [31]. If the TDG is cyclic, however, the subgoal ordering will depend on the specified current state and goal state. Furthermore, identifying the correct ordering that serializes the subgoals requires solving the planning problem itself.

5.1.3 Decomposing Concurrent Automata

If the TDG of a $\mathcal{CA}$ is acyclic, the concurrent automata decomposition is trivial. In this case, computing a goal-directed plan of the composed automaton is unnecessary, as described in Chapter 4. Instead, a goal-directed plan for each individual automaton can

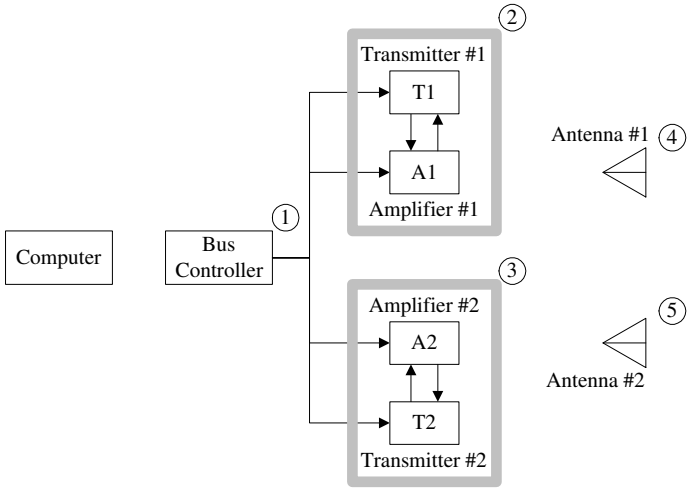

Figure 5-2: Transition dependency graph of a simplified telecommunication system. The computer has no numbering since it is not modelled.

be computed as described in the following section. In essence, the problem has been decomposed into a set of subproblems, where each subproblem is associated with an individual automaton.

Fortunately, this divide-and-conquer approach can also be applied to $\mathcal{CA}$ with a cyclic TDG, by transforming a cyclic TDG into an acyclic TDG. For example, the TDG of the simplified telecommunication system is cyclic (see in Figure 5-2) due to the inter-dependencies between each transmitter and its associated amplifier. However, if the transmitter and the amplifier are grouped into a single vertex in the TDG, the resulting graph is acyclic. As each vertex corresponds to an automaton, grouping a set of vertices corresponds to composing the automata associated with these vertices. It should be recognized that a set of cyclic vertices in a TDG directly corresponds to a *strongly connected component* (SCC) of the TDG (i.e., Transmitter #1 and Amplifier #1 together form one of the SCCs in the TDG in Figure 5-2) [15]. In addition, composing the automata within an SCC corresponds to transforming the TDG with SCCs into a *component graph* [15].

Transforming a set of concurrent automata contained within an SCC into a single automaton involves composing the concurrent automata as described in Section 4.1 (page 52). However, one small, yet important, difference exists between composing the $\mathcal{CA}$ of

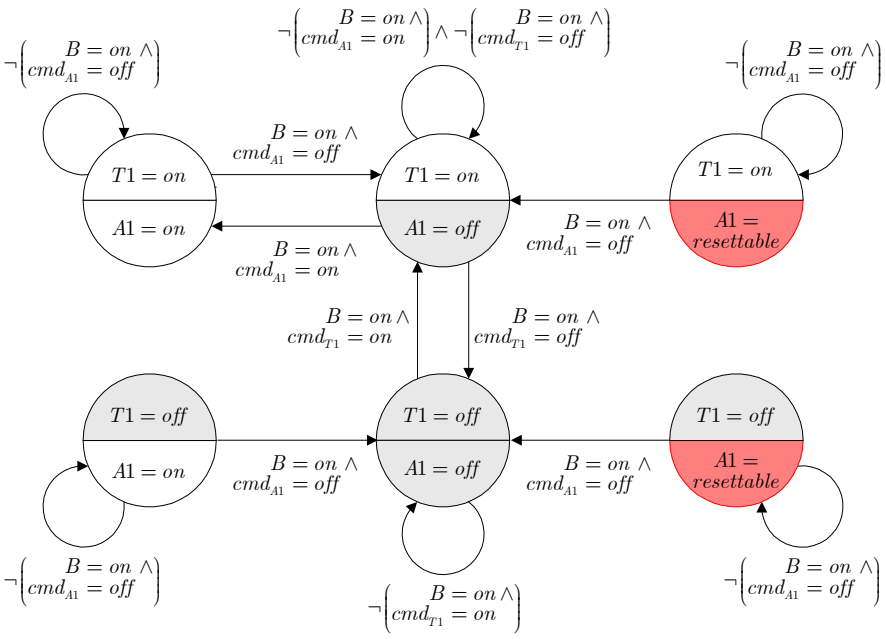

Figure 5-3: Composed automaton representing the two interdependent components, Transmitter #1 and Amplifier #1.

the entire system versus composing a subset of the concurrent automata.

To illustrate this difference, consider composing Transmitter #1 and Amplifier #1. The composition of two automata that represent the components is shown in Figure 5-3. Note that when the $\mathcal{CA}$ of the entire system is composed into a single automaton, the transitions of the composed automaton are only conditioned on the commands. However, when a subset of the $\mathcal{CA}$ is composed into an automaton, the transitions of the composed automaton may be conditioned on the state of other automata as well. For example, when Transmitter #1 and Amplifier #1 are composed, the transitions of the composed automaton are conditioned on the Bus Controller state.

Once $\mathcal{CA}$ are decomposed into a set of SCCs, the ordering on the decomposed automata is defined for the purpose of subgoal serialization. As discussed in [31], the subgoal serialization ordering is computed using a simple topological sorting algorithm [15] on the component graph of the TDG. The topological order, as labelled in Figure 5-2, corresponds to the inverse subgoal serialization ordering.

Current State	Goal State			
	$T1 = on,\ A1 = on$	$T1 = on,\ A1 = off$	$T1 = off,\ A1 = off$	$T1 = off,\ A1 = on$
$T1 = on,\ A1 = on$	*idle*	$B = on$ $cmd_{A1} = off$	$B = on$ $cmd_{A1} = off$	*failure*
$T1 = on,\ A1 = off$	$B = on$ $cmd_{A1} = on$	*idle*	$B = on$ $cmd_{T1} = off$	*failure*
$T1 = on,\ A1 = resettable$	$B = on$ $cmd_{A1} = off$	$B = on$ $cmd_{A1} = off$	$B = on$ $cmd_{T1} = off$	*failure*
$T1 = off,\ A1 = off$	$B = on$ $cmd_{T1} = on$	$B = on$ $cmd_{T1} = on$	*idle*	*failure*
$T1 = off,\ A1 = on$	$B = on$ $cmd_{A1} = off$	$B = on$ $cmd_{A1} = off$	$B = on$ $cmd_{A1} = off$	*idle*
$T1 = off,\ A1 = resettable$	$B = on$ $cmd_{A1} = off$	$B = on$ $cmd_{A1} = off$	$B = on$ $cmd_{A1} = off$	*failure*

Figure 5-4: Goal-directed plan for the Transmitter #1/Amplifier #1 SCC automaton. Similar to the goal state $\{T1 = off,\ A1 = on\}$, neither the goal state $\{T1 = on,\ A1 = resettable\}$ nor $\{T1 = off,\ A1 = resettable\}$ can be reached from any other states.

5.2 Decomposed Goal-directed Plan

Once the TDG is made acyclic by replacing each SCC with a single vertex, a GDP can be computed for each SCC successively, instead of generating a single GDP for the entire $\mathcal{CA}$. This set of GDPs for all SCCs in the system is called a *decomposed goal-directed plan* (DGDP). For example, the GDP of a composed automaton that represent the Transmitter #1/Amplifier #1 SCC automata is shown in Figure 5-4. This GDP, unlike the GDP of the composed automaton that represents the Transmitter #1/Amplifier "system", can include *intermediate subgoals* (i.e., $B = on$) within the control action μ.

For example, when the transmitter and the amplifier are off (i.e., $T1 = off,\ A1 = off$), the goal-directed rule for the goal state $\{T1 = on, A1 = on\}$ is:

$$\langle \{T1 = off, A1 = off\}, \{B = on, cmd_{T1} = on\}, \{T1 = on, A1 = on\} \rangle. \qquad (5.1)$$

Here, $B = on$ is an intermediate subgoal that must be achieved before the command $cmd_{T1} = on$ can be executed. The intermediate subgoal can be achieved simply by looking up the GDP of the Bus Controller. Upon success in achieving the intermediate goal, the control action $cmd_{T1} = on$ can be commanded. The achievement of the intermediate subgoal $B = on$ is an essential "side-effect" of achieving the subgoal $\{T1 = off,\ A1 = off\}$.

The following section introduces how a DGDP is computed, followed by an analysis of DGDP sizes.

5.2.1 Computing a DGDP

A DGDP is generated by computing the GDPs of each composed automaton associated with an SCC. While the ComputeGDP algorithm (see Section 4.1, page 60) can be used to compute the GDPs, a simple modification must be made to the way ComputeGDP is called. Recall that GDPs in a DGDP may include intermediate subgoals in the control action of a goal-directed rule. Intermediate subgoals are handled in ComputeGDP by simply treating them as commands. For example, in the case of the Transmitter #1/Amplifier #1 SC composed automaton, $T1$ and $A1$ define the state of the automaton. B, which defines the state of the Bus Controller, is treated as a control variable for the purposes of computing the GDP of the Transmitter #1/Amplifier #1 composed automaton.

Additionally, all intermediate subgoals of GDPs in a DGDP must be *reversibly reachable states* [31]. Reversibly reachable states are states that can be reached from the initial state and can also lead back to the initial state through some sequence of actions. In general, fault states of a component are not reversibly reachable, but the nominal states are reversibly reachable. In some cases, however, even some nominal states are not reversibly reachable. For example, one of the components commonly used on a spacecraft's propulsion system is the normally-open pyro-valve. This valve is initially open, but can be closed. Once closed, however, the valve is permanently closed, with no mechanism to reopen the valve. Thus, the valve's closed state is not reversibly reachable from the open state. These valves are used because they are highly reliable and virtually leak-free, but because of the irreversibility associated with the closed state, the valve should be closed if and only if it is truly necessary. Hence, if the closed state is explicitly requested as a subgoal, executing the GDP of the valve should close the valve. However, the valve should not be commanded closed as a side-effect of achieving other subgoals, as this side-effect may not be the original intent. If the valve is closed unintentionally, the permanently closed valve could potentially fail the mission. Thus, a GDP must be generated such that

it can achieve any of its subgoals, including the states that are not reversibly reachable, *but should not include any intermediate subgoals that are not reversibly reachable.*

An intermediate subgoal appears in a GDP of an automaton if one or more of its transitions are conditioned on the state of other automata with lower topological number, that is, composed automata associated with the ancestors in the component graph of a TDG. Thus, if a transition relation is restricted to a subset of "allowed" transitions, which are only conditioned on the reversibly reachable states of the ancestors, the GDP computed from this "allowed" transition relation is guaranteed to include only reversibly reachable intermediate subgoals.

`ComputeDGDP`$(n, R, S, C, S', s^{(0)})$ in Algorithm 5.1 generates the DGDP of a $\mathcal{CA}$. n is the number of SCC composed automata. R is a vector of transition relations. $R[i]$ corresponds to the OBDD-encoded transition relation of the i-th composed automaton, where the composed automata are ordered in topological order. S is a vector of the state spaces of the current time step, where $S[i]$ corresponds to an OBDD-encoded state space of the i-th composed automaton. Similarly, C is a vector of control actions set, where $C[i]$ is an OBDD-encoded set of all possible control actions for the i-th composed automaton. S' is a vector of the state spaces of the next time step, where $S'[i]$ corresponds to an OBDD-encoded state space of the i-th composed automaton. $s^{(0)}$ is a vector of initial states, where $s^{(0)}[i]$ is the initial state of the i-th automaton.

Algorithm 5.1 `ComputeDGDP`$(n, R, S, C, S', s^{(0)})$

1: $revReachAncs \leftarrow true$
2: **for** $i = 1$ to n **do**
3: $\quad allwdR \leftarrow R[i] \wedge revReachAncs$
4: $\quad DGDP[i] \leftarrow$ `ComputeGDP`$(allwdR, C[i])$
5: $\quad revReachAncs \leftarrow revReachAncs \cup$ `ComputeRRS`$(R[i], S[i], C[i], S'[i], s^{(0)}[i])$
6: **return** $DGDP$;

Lines 2–5 successively generates the GDPs of each SCC composed automaton. Each GDP is computed in the topological order and stored in the vector $DGDP$ (lines 2–6). In line 1, an OBDD-encoded set of reversibly reachable ancestor states, $revReachAncs$, is initialized to $true$. Note that an OBDD set equal to $true$ implies that the set includes

all possible elements of the set. Then, *revReachAncs*, computed in the i-th iteration, is used to compute *allwdR* of the $(i+1)$-th SCC composed automaton (line 3). *allwdR* is computed by restricting the transition relation $R[i]$ to only the reversibly reachable ancestor states (i.e., states of the composed automata with the topological number lower than i). Line 4 computes a GDP by calling `ComputeGDP` and stores the GDP in $DGDP$. Line 5 computes the reversibly reachable states of the i-th SCC and adds them to the set of reversibly reachable ancestor states, which is used to compute *allwdR* of the $(i+1)$-th SCC composed automaton in line 3. `ComputeDGDP` is based on the algorithm for computing *concurrent policies*[31].

`ComputeRRS`, shown in Algorithm 5.2, computes the reversibly reachable states by computing the intersection of the set of states reachable from the initial state and the set of states that can reach the initial state. This is similar to computing a SCC. `ComputeRRS` takes the transition relation R, the state space of the current time step S, a set of all possible control actions C, the state space of the next time step S', and the initial state $s^{(0)}$.

Algorithm 5.2 `ComputeRRS`$(R, S, C, S', s^{(0)})$

1: $RNoCmd \leftarrow \exists(\mu \in C).R$
2: $fReachOld \leftarrow \emptyset$
3: $fReachNew \leftarrow s^{(0)}$
4: **while** $fReachNew \neq fReachOld$ **do**
5: $\quad fReachOld \leftarrow fReachNew$
6: $\quad fReachNew \leftarrow fReachOld \cup (\exists(s \in S).(RNoCmd \wedge fReachOld))_{[S'/S]}$
7: $bReachOld \leftarrow \emptyset$
8: $bReachNew \leftarrow s^{(0)}$
9: **while** $bReachNew \neq bReachOld$ **do**
10: $\quad bReachOld \leftarrow fReachNew$
11: $\quad bReachNew \leftarrow bReachOld \cup \exists(s' \in S').(RNoCmd \wedge fReachOld_{[S/S']})$
12: **return** $fReachNew \cap bReachNew$

In computing the reversibly reachable states, it is not important to determine how the states are reversibly reachable, rather, only the fact that they are reversibly reachable is necessary. Thus, the control action information in the transition relation is irrelevant. As such, line 1 modifies the transition relation $\langle \sigma, \mu, \sigma' \rangle$ into $\langle \sigma, \sigma' \rangle$ and stores it in $RNoCmd$.

Line 2 initializes $fReachOld$ to an empty set. $fReachOld$ represents a set of all states that were found to be reachable in the previous iteration of the search. Line 3 initializes $fReachNew$ to the initial state. $fReachNew$ represents a set of all reachable states found in all iterations, up to the current iteration. Lines 4–6 iteratively searches for all states that are reachable from the current state, i.e., forward reachable states. While line 5 updates $fReachOld$ with $fReachNew$, line 6 adds a set of states that are reachable from $fReachOld$ into $fReachNew$. This process is continued until the fixed-point is reached, at which no more new forward reachable states are found (line 4).

Similarly, line 7 initializes $bReachOld$ to an empty set, where $bReachOld$ represents a set of all states that were found to be backward reachable (i.e., all states that can reach the initial state) in the previous iteration of the search. Line 8 initializes $bReachNew$ to the initial state. $bReachNew$ represents a set of all backward reachable states found in all iterations, up to the current iteration. Lines 9–11 iteratively searches for all states that can reach the current state. While line 10 updates $bReachOld$ with $bReachNew$, line 11 adds a set of states can reach $bReachOld$ into $bReachNew$. This process is continued until the fixed-point is reached, at which no more new backward reachable states are found (line 9).

Finally, the intersection of the forward and backward reachable states is returned as the reversibly reachable states (line 12).

5.2.2 DGDP Size Analysis

The advantage of this method is that the size of the DGDP is much smaller than a single GDP for the full $\mathcal{CA} = \langle \mathcal{A}, \Pi, \Sigma \rangle$. For a given automaton with x number of states, the size of the GDP for the automaton is quadratic in x, $O(x^2)$. Now, assume that the number of concurrent automata is $n = |\mathcal{A}|$, and the average number of states per automaton is $m = |\Sigma_i^{s_i}|$. If $\mathcal{CA}$ is composed into a single automaton, the number of states in the GDP grows exponentially in n, $O(m^n)$, and so does the size of the GDP, $O(m^{2n})$. If the maximum number of the automata in an SCC is w, however, the number of states in the corresponding DGDP is only $O(l \cdot m^w)$ and the size of the DGDP is only $O(l \cdot m^{2w})$,

where l is the total number of SCCs. Thus, even if the size of a $\mathcal{CA}$ grows, as long as m and w remains constant, the size of the corresponding DGDP grows only linearly in l. Constant m and w is a fair assumption, since components with large state space are seldom built and systems with many interdependent components are also rare.

Chapter 6

DGDP Execution

In Chapter 5, the method for constructing a DGDP was introduced. As discussed earlier, a DGDP is capable of reactively repairing and reconfiguring. A system with a DGDP is quick to respond to anomalies since all necessary reactive plans have been computed offline. During online execution, no replanning is necessary.

In this chapter, the simplified telecommunication system example is used to demonstrate the ability to repair and reconfigure using a DGDP. This example is followed by a description of the online execution algorithm `ExecuteDGDP`. This algorithm is identical to the concurrent policy execution algorithm developed by Williams and Nayak [31]. The only difference is that the execution algorithm presented in this chapter operates on OBDD-encoded DGDPs. Next, the time complexity of the execution algorithm is discussed based on the analysis work of Williams and Nayak [31]. Finally, the optimality of DGDPs is informally discussed.

6.1 DGDP Execution Example

Once again, consider the simplified telecommunication system (see Section 2.3, page 30). This system is decomposed into five sets of SCC composed automata: Bus Controller, Transmitter #1/Amplifier #1, Transmitter #2/Amplifier #2, Antenna #1, and Antenna #2. As a result, the DGDP of the simplified telecommunication system includes five GDPs, each GDP associated with one of the composed automata. The GDP of the Bus Controller is shown in Figure 6-1. The GDP of Transmitter #1/Amplifier #1 composed automaton is shown in Figure 6-2. The GDP of the Transmitter #2/Amplifier #2 composed automaton is not illustrated, since the GDP is exactly the same as the GDP

Current State	Goal State	
	$B = on$	$B = off$
$B = on$	$idle$	$cmd_B = off$
$B = off$	$cmd_B = on$	$idle$

Figure 6-1: Goal-directed plan for the Bus Controller SCC composed automaton.

Current State	Goal State			
	$T1 = on$, $A1 = on$	$T1 = on$, $A1 = off$	$T1 = off$, $A1 = off$	$T1 = off$, $A1 = on$
$T1 = on$, $A1 = on$	$idle$	$B = on$ $cmd_{A1} = off$	$B = on$ $cmd_{A1} = off$	$failure$
$T1 = on$, $A1 = off$	$B = on$ $cmd_{A1} = on$	$idle$	$B = on$ $cmd_{T1} = off$	$failure$
$T1 = on$, $A1 = resettable$	$B = on$ $cmd_{A1} = off$	$B = on$ $cmd_{A1} = off$	$B = on$ $cmd_{T1} = off$	$failure$
$T1 = off$, $A1 = off$	$B = on$ $cmd_{T1} = on$	$B = on$ $cmd_{T1} = on$	$idle$	$failure$
$T1 = off$, $A1 = on$	$B = on$ $cmd_{A1} = off$	$B = on$ $cmd_{A1} = off$	$B = on$ $cmd_{A1} = off$	$idle$
$T1 = off$, $A1 = resettable$	$B = on$ $cmd_{A1} = off$	$B = on$ $cmd_{A1} = off$	$B = on$ $cmd_{A1} = off$	$failure$

Figure 6-2: Goal-directed plan for the Transmitter #1/Amplifier #1 SCC automaton. Similar to the goal state $\{T1 = off,\ A1 = on\}$, neither goal state $\{T1 = on,\ A1 = resettable\}$ nor $\{T1 = off,\ A1 = resettable\}$ can be reached from any other states.

of the Transmitter #1/Amplifier #1 composed automaton, except for the replacement of the variables $T1$ and $A1$ with $T2$ and $A2$, respectively. Also, the GDPs of the two antennas have been omitted, as they are passive devices. The topological ordering of the decomposed system is (B, $T1/A1$, $T2/A2$, $Ant1$, $Ant2$) as shown in Figure 6-3.

As an example, assume that the current state and the desired goal state of the simplified telecommunication system is as follows:

- Current State: $\{B = off, T1 = off, A1 = off, T2 = off, A2 = off, Ant1 = nominal, Ant2 = nominal\}$

- Goal State: $\{B = on,\ T1 = on,\ A1 = on,\ T2 = off,\ A2 = off,\ Ant1 = nominal,\ Ant2 = nominal\}$

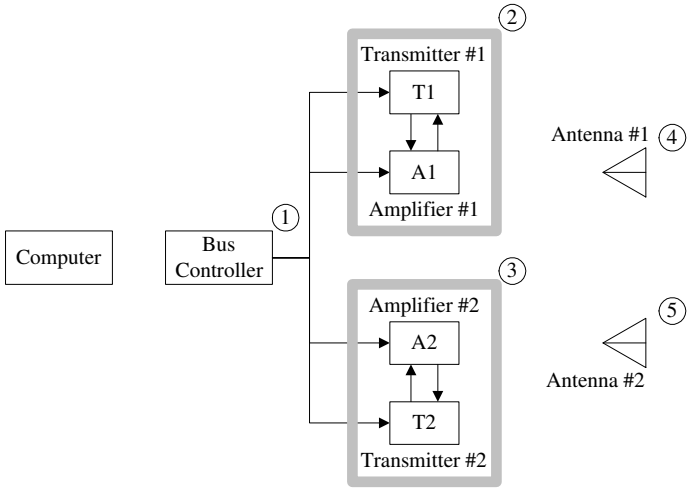

Figure 6-3: Transition dependency graph of a simplified telecommunication system. The numbers correspond to the topological order. The computer has no numbering since it is not modelled.

6.1.1 Nominal Execution

Before executing the DGDP, the goal must be partitioned into subgoals and then serialized. According to the decomposition shown in Figure 6-3, the goals associated with $T1$ and $A1$ must be combined into a single subgoal, and the goals associated with $T2$ and $A2$ must be combined into a subgoal. The list of subgoals in topological order is shown below:

- Serialized Subgoals: $(\{B = on\}, \{T1 = on, A1 = on\}, \{T2 = \mathit{off}, A2 = \mathit{off}\}, \{Ant1 = nominal\}, \{Ant2 = nominal\})$

First Command: Turn on the Bus Controller

For proper execution, the subgoals must be achieved in inverse topological order, which is the order for successful goal serialization. The first subgoal is $Ant2 = nominal$. Since the current state of Antenna #2 is *nominal*, nothing needs be done. Similarly, nothing is required for the second subgoal ($Ant1 = nominal$) nor the third ($T2 = \mathit{off}$, $A2 = \mathit{off}$). However, actions must be taken to achieve the forth subgoal ($T1 = on$, $A1 = on$). Looking up the GDP of $T1/A1$ in Figure 6-2. The following goal-directed rule applies

for the current state:

$$\langle\{T1 = \mathit{off}, A1 = \mathit{off}\}, \{B = on, cmd_{T1} = on\}, \{T1 = on, A1 = on\}\rangle \tag{6.1}$$

According to this rule the intermediate subgoal $B = on$ must be achieved before issuing $cmd_{T1} = on$. Thus, the GDP of the Bus Controller is used to determine how to achieve $B = on$ from the current state $B = \mathit{off}$. The corresponding goal-directed rule from Figure 6-1 is:

$$\langle\{B = \mathit{off}\}, \{cmd_B = on\}, \{B = on\}\rangle \tag{6.2}$$

This rule requires commanding $cmd_B = on$.

Second Command: Turn on Transmitter #1

After the execution of $cmd_B = on$, the system transitions nominally and the new state is:

- Current State: $\{B = on$, $T1 = \mathit{off}$, $A1 = \mathit{off}$, $T2 = \mathit{off}$, $A2 = \mathit{off}$, $Ant1 = nominal$, $Ant2 = nominal\}$

Given the new state, the next unachieved subgoal with the highest topological number is $\{T1 = on, A1 = on\}$. Again the same goal-directed rule is applied:

$$\langle\{T1 = \mathit{off}, A1 = \mathit{off}\}, \{B = on, cmd_{T1} = on\}, \{T1 = on, A1 = on\}\rangle \tag{6.3}$$

This time, since the intermediate subgoal $B = on$ has already been achieved, $cmd_{T1} = on$ can be commanded.

Third Command: Turn on Amplifier #1

Again, the system behaves nominally and the command $cmd_{T1} = on$ progresses the system to a state closer to the goal state:

- Current State: $\{B = on$, $T1 = on$, $A1 = \mathit{off}$, $T2 = \mathit{off}$, $A2 = \mathit{off}$, $Ant1 = nominal$, $Ant2 = nominal\}$

Once more from the $T1/A1$ GDP, the required next command is determined to be $cmd_{A1} = on$.

6.1.2 Repairing a Faulty State

This time, however, assume that Amplifier #1 fails to the *resettable* state instead of finally transitioning the system into the goal state:

- Current State: $\{B = on, T1 = on, A1 = resettable, T2 = off, A2 = off, Ant1 = nominal, Ant2 = nominal\}$

According to the GDP of $T1/A1$, the proper repair action is to command the amplifier off ($cmd_{A1} = off$). Once that repair action is successful, the same process above can be repeated to reach the goal state.

6.1.3 Reconfiguration

However, if Antenna #1 fails during the process of achieving the goal state, nothing can be done to achieve the goal state. To bring the telecommunication system online, the redundant antenna must be used. Thus, the new reconfiguration goal state is:

- Goal State: $\{B = on,\ T1 = off,\ A1 = off,\ T2 = on,\ A2 = on,\ Ant1 = failed,\ Ant2 = nominal\}$

Fortunately, the new goal state can still be achieved using the same DGDP and the same execution process. Unlike universal plans, no new plan needs to be generated.

6.2 Algorithm ExecuteDGDP

ExecuteDGDP shown in Algorithm 6.1 essentially performs the process outlined in the previous section. It attempts to achieve each subgoal in inverse topological order. Recall that this algorithm is identical to the concurrent policy execution algorithm in [31].

The first argument to algorithm ExecuteDGDP is the decomposed goal-directed plan $DGDP$. As described in Section 5.2 (page 71), $DGDP$ is a vector of GDPs, where the

Algorithm 6.1 ExecuteDGDP($DGDP, S, G, firstCall$)

```
1: if firstCall then
2:    for i = 1 to |G| do
3:       j = TopologicalNum(G[i])
4:       if (DGDP[j] ∧ S[j] ∧ G[i]) = false then
5:          return failure
6: for i = |G| to 1 do
7:    j = TopologicalNum(G[i])
8:    if G[i] ≠ S[j] then
9:       contAct = ExecuteGDP(DGDP[j], S[i], G[i])
10:      ⟨intG, cmd⟩ = PartitionAct(contAct)
11:      nextCmd = ExecuteDGDP(DGDP, S, intG, false)
12:      if nextCmd = success then
13:         return cmd
14:      else
15:         return nextCmd
16: return success
```

GDPs are ordered in the topological order of the corresponding composed automata. The second argument, S, is a vector of current states also ordered in the topological order. G, a vector of subgoals ordered in the topological order, represents the goal. A goal may be expressed as a full assignment to the state variables of the $\mathcal{CA}$, or even as a partial assignment to the state variables. Thus, the size of the goal vector $|G|$ may be less than the size of the current state vector $|S|$, i.e., the number of SCCs in the TDG of $\mathcal{CA}$. As ExecuteDGDP is a recursive algorithm, the argument $firstCall$ is used to distinguish the initial call from the recursive calls. If $firstCall$ is $true$, that implies that ExecuteDGDP is being called for the first time.

The first step in ExecuteDGDP is to check if the goal is achievable (lines 2–5). If any one of the subgoals is unachievable, it is unnecessary to achieve the remaining subgoals, since even a single unachieved subgoal still implies failure to achieve the goal. For example, consider the simplified telecommunication system. If the desired goal is to turn on the Bus Controller, Transmitter #1, and Amplifier #1, but the amplifier is currently in a nonrecoverable failure state, there is no point in turning on the Bus Controller and Transmitter #1.

Thus, in lines 2–5, each subgoal is checked to ensure that a plan exists in the DGDP

that can achieve all subgoals. If any one of the subgoals are not achievable, $failure$ is returned by ExecuteDGDP (line 5). In more detail, line 3 calls TopologicalNum to determine the topological number associated with the subgoal $G[i]$. Though this algorithm is not defined, it is a matter of looking up a table that maps the OBDD variables used in $G[i]$ to the topological number associated with the corresponding composed automaton. Once the corresponding topological number j is determined, line 4 checks to see if the corresponding GDP, $DGDP[j]$, includes a rule that can achieve subgoal $G[i]$ from the current state $S[j]$. If any of the subgoals are determined unachievable, line 5 returns $failure$. Note that this check is done only once (line 1), because a goal is achievable if and only if all subgoals are achievable. All intermediate subgoals are guaranteed to be achievable, since the DGDP was computed so that all intermediate subgoals are not only reachable, but reversibly reachable.

Once all subgoals are determined achievable, the next step is to compute the necessary action that achieves the goal (lines 6–16). Each subgoal in G is achieved in the inverse topological order (line 6), as required by subgoal serializability discussed in Section 5.1 (page 66). Once again, line 7 determines the topological number associated with the subgoal $G[i]$. For each subgoal, line 8 checks to see if the subgoal has already been achieved, i.e., the subgoal state is the same as the current state of the composed automata associated with the subgoal. If all subgoals have been achieved, *success* is returned in line 16. However, when the first unachieved subgoal is found, the appropriate control action is computed by simply calling ExecuteGDP with the parameters $DGDP[j]$, $S[j]$, and $G[i]$. $DGDP[j]$ is the GDP associated with the subgoal $G[i]$. $S[j]$ is the current state of the composed automaton associated with $G[i]$. For more detail on ExecuteGDP, see Section 4.2 (page 57).

Once ExecuteGDP returns the control action $contAct$ (line 9), it is partitioned into a pair $\langle intG, cmd \rangle$, where $intG$ is a vector of intermediate subgoals in the topological order and cmd is the command that must be issued after achieving the intermediate subgoals (line 10). PartitionAct computes this pair. Now, the set of intermediate subgoals $intG$ must be achieved before achieving subgoal $G[i]$. Line 11 calls ExecuteDGDP with $intG$

instead of G. Also, $false$ flag is passed on to identify that this is a recursive call. Since all intermediate subgoals are guaranteed achievable, the only possible values for $nextCmd$ are either a command or *success*. Line 12 checks to see which value has been returned, and if the value is *success*, i.e., all intermediate subgoals have been achieved, then *cmd* is finally returned (line 13). However, if the value of $nextCmd$ is not *success*, it must be a command that achieves one of the intermediate subgoals in $intG$. This command is returned instead of *cmd* (line 15), since the intermediate subgoals must be achieved first.

6.3 DGDP Execution Time

The execution algorithm consists of two major iterative loops, lines 1–5 and lines 6–15. The number of iterations performed by each loop is n, where n is the number of subgoals. Note that the maximum size of subgoals is equivalent to the number of SCCs computed during $\mathcal{CA}$ decomposition; and the maximum number of SCCs for any given $\mathcal{CA} = \langle \mathcal{A}, \Pi, \Sigma \rangle$ is equivalent to the number of concurrent automata, i.e. $n = |\mathcal{A}|$. Then, the first loop iterates at most n times. Also, this first loop is never iterated in the recursive calls to `ExecuteDGDP`.

As for the second loop, there are two possibilities: either no action is necessary and *success* is returned, or a subgoal must be achieved. In the first case, the loop will be iterated exactly n times. In the second case, `ExecuteDGDP` is recursively called, in which case the maximum number of iterations on the second loop is n. For example, assume that m-th subgoal needs to be achieved. This means that the second loop was iterated m times to find the m-th subgoal. At the same time, the composed automaton associated with this subgoal can have at most $(n-m)$ ancestors, or equivalently $(n-m)$ intermediate subgoals. Continuing on with this process results in at most n iterations for the second loop.

With all polynomial operations within the loops, the execution time of a single command is $O(n)$, where n is the number of components in the system. Thus, if a plan exists for a problem, DGDP will guarantee to issue a command that progresses the system toward the goal in $O(n)$ time.

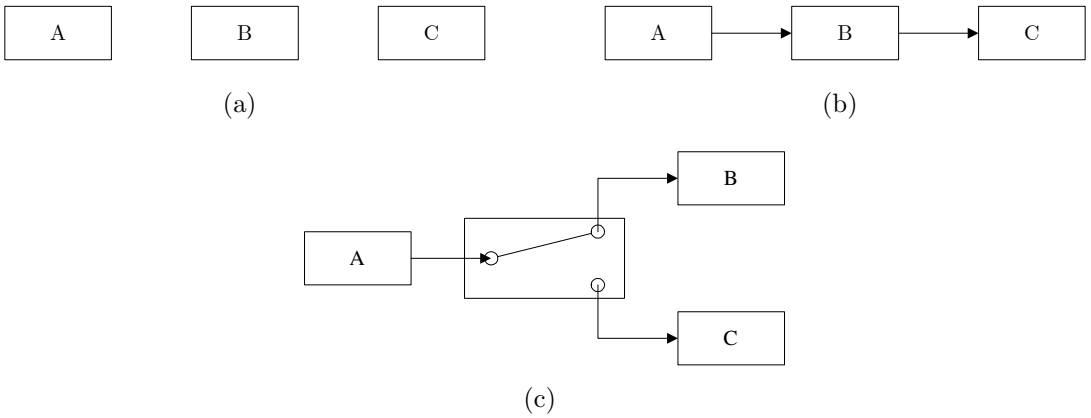

Figure 6-4: Three possible configurations of the transition dependency graph: (a) independent, (b) serial, and (c) branched.

6.4 DGDP Optimality

If an optimal plan is defined as the plan with the least number of actions required to achieve the goal, a GDP of a $\mathcal{CA}$ is guaranteed to be optimal. A DGDP, however, does not guarantee optimality. While each GDP in a DGDP is optimal, depending on the ordering of the subgoal serialization, a DGDP may or may not be optimal. Consider the three possible configurations of $\mathcal{CA}$ TDGs shown in Figure 6-4. In the case of the configuration shown in Figure 6-4(a), the ordering is irrelevant since none of the components, A, B, nor C, are dependent on one another. Thus, the number of actions required to manipulate the components will be independent of the ordering of the components. As such, a DGDP will also be optimal. Next, consider the serial configuration shown in Figure 6-4(b). In this case, due to the serial dependency among components, only one serialization ordering is possible. Thus, the ordering used in a DGDP is also guaranteed to be the optimal one.

In the branched TDG configuration shown in Figure 6-4(c), the Switch serves as a mechanism that routes commands from A to either B or C. Thus, B can be turned on or off by A, only if the Switch connects the path from A to B. Similarly, C can be turned on or off, only if the Switch connects the path from A to C. For this problem, there are two possible subgoal serialization orderings: (B, C, Switch, A) and (C, B, Switch, A).

Now, assume that both B and C are off and the desired goal is to turn both of them

on. Using the ordering (B, C, Switch, A), the following achieves the goal: (1) turn on B, (2) Switch to C path, and (3) turn on C. However, if the ordering (C, B, Switch, A) is used, the plan takes one additional step to achieve the goal: (1) Switch to C path (2) turn on C (3) Switch to B path and (4) turn on B. Note that the Switch was positioned in the C path as a side-effect of turning C on. This is a result of the two components B and C competing for the same resource (i.e., the Switch). However, if the two components were sharing a resource instead of competing for a resource, both orderings would be equally optimal. In summary, a DGDP can be suboptimal if there are branches in the component graph of a TDG, and the sibling components compete for a resource.

Chapter 7

Results

In this chapter, the implementation and some empirical results on relevant examples are discussed.

7.1 Implementation

The reactive planner was implemented in C++ using an OBDD package called BuDDy (Release 2.0), developed by Jørn Lind-Nielsen of the IT-University of Copenhagen [25]. The BuDDy package provides a C++ library that defines the OBDD data structure and provides operators for manipulating OBDDs. As described previously, many of the concepts underlying the decomposed symbolic approach to reactive planning were inspired by the Burton reactive planner [31]. Likewise, the implementation of DGDP was inspired by the implementation of Burton as well.

Many OBDD packages have been developed and their relative performances have been compared [32]. The BuDDy package was chosen for its user-friendliness, simplicity, and C++ interface. The performance of the OBDD package was not an important consideration.

Along with the BuDDy package, Lind-Nielsen provides documentation describing all available OBDD operators [25]. Some of the commonly used OBDD operators are listed in Table 7.1.

While the algorithms were implemented with the Model-based Executive (see Section 1.1 16) in mind, the interface necessary to plug the new reactive planner into the Model-based Executive has yet to be implemented. Once the interface has been implemented, the new reactive planner should operate within the Titan Model-based Executive [29], in

Table 7.1: BuDDy Functions for OBDD Operators

OBDD Operator	Corresponding BuDDy Function
$\neg A$	`bdd_not(A, B)`
$A \wedge B$	`bdd_and(A, B)`
$A \vee B$	`bdd_or(A, B)`
$A \Rightarrow B$	`bdd_imp(A, B)`
$\exists A.B$	`bdd_exist(A, B)`

a manner similar to the Burton reactive planner.

7.2 Empirical Results

A goal-directed plan for a system is computed by composing all concurrent automata of a system into a single composed automaton and then by mapping this automaton to a plan. A difficulty with this encoding is that the size of the plan is exponential in the number of system components being controlled.

The decomposed symbolic approach to reactive planning offers a solution to compactly encoding this plan. A DGDP of a system is computed by decomposing the system into sets of strongly connected components (SCCs), and by using a symbolic encoding to compactly represent the plan corresponding to the composed automaton of each SCC. The compactness of this encoding was proven analytically in Section 5.2 (page 71). This chapter reinforces this point empirically, by comparing the size of a GDP with its corresponding DGDP, for problems with varying state space sizes.

7.2.1 Case Scenarios

A GDP and its corresponding DGDP was computed for six different systems, which vary in the size of the system's state space. These cases were constructed by modifying the number of components within the simplified telecommunication system example (Section 2.3).

Case 1:

Starting with the simplest case, the first example consists of only one component, Amplifier #1. To generate this case, all transition conditions on the Bus Controller and Transmitter #1 were removed, thus assuming the amplifier's transitions do not depend on any other components.

- 1 Amplifier

Case 2:

The second case adds Transmitter #1 to the first case. Since the Bus Controller is not part of the system, the transitions of both the amplifier and the transmitter were assumed to be independent of the Bus Controller's state.

- 1 Transmitter
- 1 Amplifier

Case 3:

The third case adds the Bus Controller to the second case. In this system none of the transition conditions were modified.

- 1 Bus Controller
- 1 Transmitter
- 1 Amplifier

Case 4:

This case is comprised of two transmitter and amplifier pairs, along with the Bus Controller.

- 1 Bus Controller
- 2 Transmitters
- 2 Amplifiers

Case 5:

This case is the original simplified telecommunication system, described in Section 2.3. The simplified telecommunication system includes a Bus Controller, two transmitter/amplifier pairs, and two antennas, as listed below:

- 1 Bus Controller
- 2 Transmitters
- 2 Amplifiers
- 2 Antennas

Case 6:

Case 6 is the largest example tested. This system includes all relevant components within the MESSENGER spacecraft's telecommunication system. The only components missing are two diplexers and four antennas. They were excluded since they are all passive devices, that is, none of them can be commanded on or off.

- 1 Bus Controller
- 2 Transmitters
- 2 Amplifiers
- 2 Antennas
- 2 Receivers
- 6 Switches

7.2.2 Experimental Results

Table 7.2 lists the experimental results for each of the six cases. The size of the state space is specified for each case in the second column. The third and fourth columns list

Table 7.2: Plan Size Comparison

	Number of States	GDP†	Burton‡	DGDP†
Case 1:	3	9	9	9
Case 2:	6	37	36	37
Case 3:	12	63	40	48
Case 4:	72	237	76	93
Case 5:	288	241	84	97
Case 6:	18432	384	116	145

†The size of a plan is the number of nodes in its OBDD representation.
‡Burton indicates the use of only the decomposition method. The size of the plan is the size of the array necessary to represent the plan.

the sizes of the computed GDPs and DGDPs. The sizes of the GDPs and DGDPs are measured in terms of the number of nodes that appear in their OBDD encoding. For cases 1 through 5, the sizes of the GDP and the DGDP as a function of state space size is plotted in Figure 7-1. The sixth case was omitted since its number of states is too large compared to the state space size of the other cases.

Note that neither the GDP size nor the DGDP size grows exponentially with the number of states, as indicated by the trend shown in Figure 7-1. The trend for DGDP size growth was as expected from earlier analytical analysis. However, the trend for GDP size growth is less than expected by the analytical analysis. Nonetheless, this is not surprising given that OBDD encodings are known to be very compact.

According to Table 7.2, what is even more striking is that a goal-directed plan can be compactly represented without the use of the OBDD encoding. The data seems to suggest that the symbolic encoding is unnecessary, and that a plan can be compactly represented using only the decomposition method. However, this is only true if the number of interdependent components is minimal, as is the case for the telecommunication system (i.e., two interdependent components). Assume a total of 15 components as is in the sixth case, but all 15 components are interdependent. Then, without the symbolic encoding, the size of the plan will be in the order of 18000^2. On the other hand, if the symbolic encoding is used, the size of the plan is 384. In this case, the symbolic approach

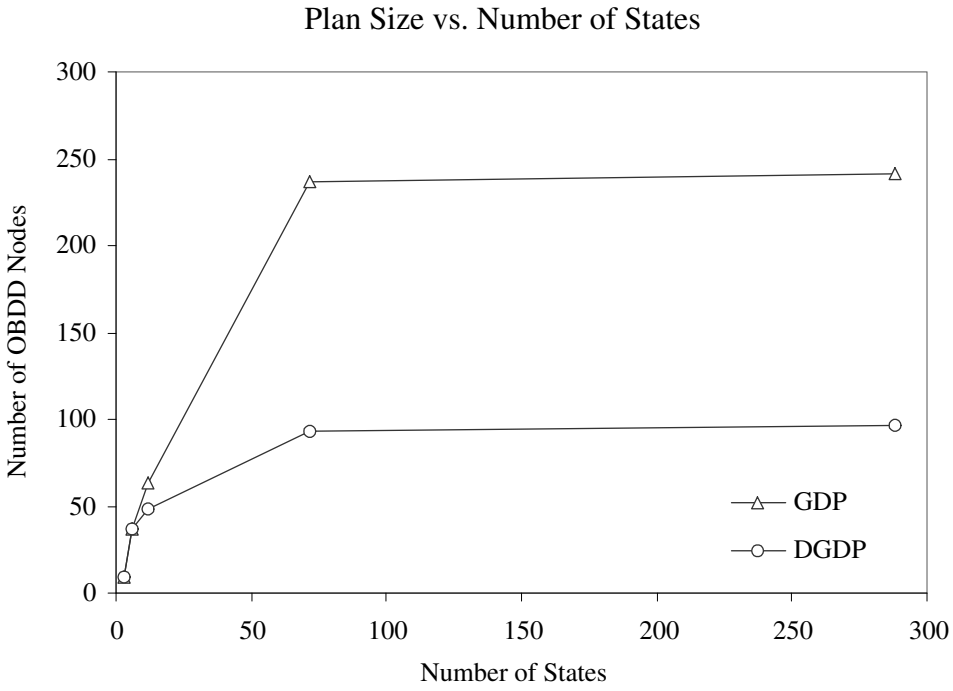

Figure 7-1: A plot of GDP and DGDP size versus number of states.

is desired over the decomposition method. Thus, DGDP combines the benefits of both the decomposition and symbolic approaches to provide a compact representation in all cases.

Chapter 8

Conclusion

The decomposed symbolic approach to reactive planning presented in this thesis is novel in three ways. First, it unifies two complementary approaches: transition-based decomposition and symbolic encoding. When transition-based decomposition is used to solve a problem, the complexity of the problem becomes linear in the size of the SCCs instead of being exponential in the size of $\mathcal{CA}$. As long as the size of the SCCs remain relatively small, the problem is guaranteed to be tractable. However, even if the size of the SCC is large, the OBDD representation of individual GDPs generally will be very compact as was validated empirically. Hence the decomposition-based and symbolic approaches work strongly together to tame state space explosion.

Second, the new approach generalizes the "divide-and-conquer" approach, introduced by the Burton reactive planner, from systems restricted to acyclic interdependencies to general systems of interdependent components.

Third, the new approach generalizes OBDD plan encodings from universal plans, which conditions on only the initial states, to goal-directed plans that are also conditioned on all possible goal states, thus enabling a quick response to rapidly changing goals.

8.1 Implication on Space Missions

The significance and the necessity of a new space technology may be defined by its potential to improve mission robustness, enable new missions, and reduce mission costs. The reactive planner along with the remaining modules that comprise the model-based executive is an emerging space technology that can revolutionize space exploration in three respects:

First, a model-based executive can improve mission robustness by enabling an immediate response to unexpected failures. Without autonomy, valuable science time can be lost due to spacecraft safing. Furthermore, delayed or unreliable communication could inhibit ground operators from promptly reacting to a time-critical hazard, such as a propulsion subsystem fault occurring during orbital insertion.

Second, a Model-based Executive can enable ambitious mission scenarios that would otherwise be infeasible due to long communication delays or uncertainty in the spacecraft's operational environment. An example of such a mission is the Europa Hydrobot mission, under study by NASA Jet Propulsion Laboratory. If an ocean is discovered under the icy surface of Europa, submarine robotic explorers can be deployed to explore the dynamic and hazardous environment with very limited Earth communication.

Finally, a Model-based Executive can also lower mission costs by reducing ground operation staff and the communication bandwidth necessary to carry out a mission. Operations cost reduction will become increasingly significant for long-duration missions and multi-spacecraft constellations, such as the Terrestrial Planet Finder mission.

8.2 Future Work

The first extension to the work presented in this thesis will be the automated random $\mathcal{CA}$ generator. This addition should help better quantify the benefits of the DGDP, identifying the limitations of DGDPs as well as its strengths.

There are also a couple of interesting research topics associated with the optimality of a DGDP. First, as discussed in Section 6.4 (see page 85), in one case, in which DGDPs are known to be less than optimal. In such a case, a simple local search algorithm may be applied to determine the optimal ordering. Also, the definition of optimality can be modified to a minimization of some arbitrary cost function. For example, the transitions of the concurrent automata can be augmented to be probabilistic, as in the case of Concurrent Constraint Automata [30]. This is in essence, a Markov Decision Process (MDP) problem. While the OBDDs are inadequate for representing MDPs, a variant of the OBDDs called Algebraic Decision Diagrams (ADD) [3] have been successfully applied

to MDP problems [19]. Similarly, a DGDP could be adapted to solve an MDP problem encoded in ADDs.

Appendix A

Binary Decision Diagram

Due to the compactness of the Binary Decision Diagram (BDD) [6, 7] representation and the low time complexity of BDD manipulation, BDDs have been popular in the model-checking community [8, 32]. In recent years, BDDs have become popular in the planning community as well [11, 13, 12, 22]. The purpose of this appendix is to introduce the readers to BDDs. This theoretical development is adapted from [6, 7, 2]. The examples used in this appendix were borrowed from [2].

A.1 Ordered Binary Decision Diagram

A BDD is a directed acyclic graph representation of a Boolean function. A Boolean function is composed of Boolean variables and operators. Boolean variables may take on a value of either 0 or 1. Boolean operators include negation $\neg$, conjunction $\wedge$, disjunction $\vee$, implication $\Rightarrow$, and equivalence $\Leftrightarrow$. Equation A.1 is an example of a Boolean function.

$$(x_1 \Leftrightarrow x_2) \wedge (x_3 \Leftrightarrow x_4) \tag{A.1}$$

In a BDD, nonterminal vertices represent the Boolean variables and terminal vertices represent the possible values of the Boolean function, 0 and 1. When variables are ordered in a predetermined sequence, the representation is called Ordered Binary Decision Diagram (OBDD). This ordering is important since it guarantees canonicity of BDDs. The first of the ordered variables corresponds to the root vertex of the OBDD. To generate the OBDD, the Shannon expansion is recursively applied to the Boolean function starting from the root vertex until the terminal vertices are reached. The Shannon expansion of

the Boolean function $f(x_1, x_2, \ldots, x_n)$ is given by:

$$f = \left(\neg x_i \wedge f|_{x_i=0}\right) \vee \left(x_i \wedge f|_{x_i=1}\right) \tag{A.2}$$

where $f|_{x_i=a}$ denotes the Boolean function f for which the value of the variable x_i is restricted to the value a. For example, consider the Boolean function $f(x_1, x_2, x_3, x_4)$ given in Equation A.1 with the ordering sequence of $x_1 \prec x_2 \prec x_3 \prec x_4$. Applying the Shannon expansion to the function according to its ordering results in the following recursive calculations:

1. $f = \left(\neg x_1 \wedge f|_{x_1=0}\right) \vee \left(x_1 \wedge f|_{x_1=1}\right)$
2. $f|_{x_1=0} = \left(\neg x_2 \wedge f|_{x_1=0,x_2=0}\right) \vee \left(x_2 \wedge f|_{x_1=0,x_2=1}\right)$
   $f|_{x_1=1} = \left(\neg x_2 \wedge f|_{x_1=1,x_2=0}\right) \vee \left(x_2 \wedge f|_{x_1=1,x_2=1}\right)$
3. $f|_{x_1=0,x_2=0} = \left(\neg x_3 \wedge f|_{x_1=0,x_2=0,x_3=0}\right) \vee \left(x_3 \wedge f|_{x_1=0,x_2=0,x_3=1}\right)$
   $f|_{x_1=0,x_2=1} = \left(\neg x_3 \wedge f|_{x_1=0,x_2=1,x_3=0}\right) \vee \left(x_3 \wedge f|_{x_1=0,x_2=1,x_3=1}\right)$
   $f|_{x_1=1,x_2=0} = \left(\neg x_3 \wedge f|_{x_1=1,x_2=0,x_2=0}\right) \vee \left(x_3 \wedge f|_{x_1=1,x_2=0,x_3=1}\right)$
   $f|_{x_1=1,x_2=1} = \left(\neg x_3 \wedge f|_{x_1=1,x_2=1,x_2=0}\right) \vee \left(x_3 \wedge f|_{x_1=1,x_2=1,x_3=1}\right)$
4. ...

The time required to generate an OBDD is exponential due to the recursiveness of Shannon expansion. Figure A-1 shows the graphical OBDD representation of Equation A.1. Dashed lines represent *low* branches and the solid lines represent *high* branches, i.e., $low(x_i)$ corresponds to the branch for which $x_i = 0$ and $high(x_i)$ corresponds to the branch for which $x_i = 1$.

A.2 Reduced Ordered Binary Decision Diagram

Although the size of OBDD, as shown in Figure A-1, increases exponentially with the number of variables (bounded to 2^{n+1} vertices for a function with n variables), this size can be further reduced. OBDDs are reduced by removing duplicate terminals, duplicate nonterminals, and redundant tests. This process is called the Reduce operation.

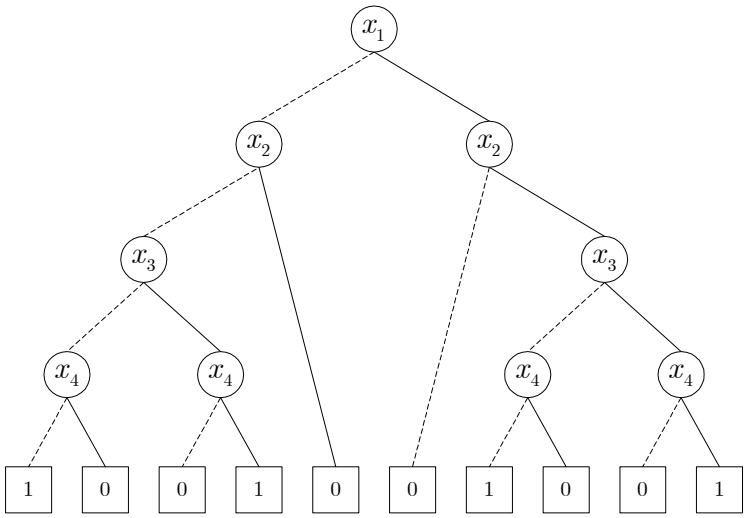

Figure A-1: OBDD of $(x_1 \Leftrightarrow x_2) \wedge (x_3 \Leftrightarrow x_4)$ with ordering sequence $x_1 \prec x_2 \prec x_3 \prec x_4$. Dashed lines represent the *low* branches and the solid lines represent *high* branches, i.e., the $low\,(x_i)$ corresponds to the branch for which $x_i = 0$ and $high\,(x_i)$ corresponds to the branch for which $x_i = 1$.

Duplicate terminals are the terminal vertices whose values are identical. For duplicate terminals, all vertices directed to the duplicate terminals are redirected to a single terminal and the rest of the duplicating terminals are removed. Figure A-2(a) illustrates the removal of duplicate terminals. Duplicate nonterminals are the vertices for which their corresponding variables, *low*s, and *high*s are identical. Formally, vertex v and v' are duplicate terminals if and only if v and v' correspond to the identical variables, $low\,(v) = low\,(v')$, and $high\,(v) = high\,(v')$. For duplicate nonterminals, all vertices directed to the duplicate nonterminals are redirected to a single nonterminal and the rest of the duplicating nonterminals are removed. Figure A-2(b) illustrates the removal of duplicate nonterminals. Redundant tests are the vertices whose *low*s and *high*s are directed to the same single vertex, $low\,(v) = high\,(v)$. For redundant tests, all vertices directed to the redundant test vertex v are redirected to the vertex that corresponds to $low\,(v)$ and the redundant test vertex is removed. Figure A-2(c) illustrates the removal of a redundant test.

Figure A-3 illustrates the reduced OBDD (ROBDD) of Equation A.1 along with the steps taken to generate the ROBDD from the OBDD. This reducing process can

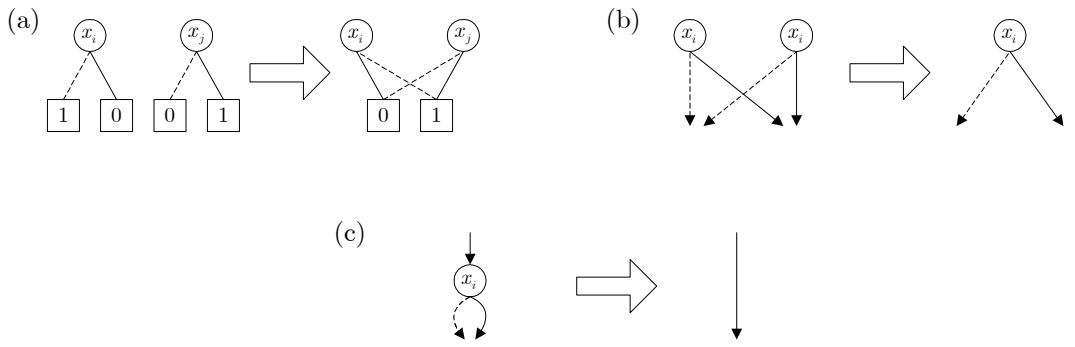

Figure A-2: Three BDD reduction methods: (a) removing duplicate terminals, (b) removing duplicate nonterminals, and (c) removing redundant tests.

be integrated into the OBDD building process. The run time of reducing is minimized through dynamic programming, as follows. Each time a new vertex is visited, its variable name and *low* and *high* vertices (children) are stored in a hash table. When another node with the same variable name and *low* and *high* vertices is encountered, the values in the hash table are used instead of recursing through the children of the new node. Since the run time of searching through the hash table is constant, the time complexity of reducing is $O(|G|)$ where $|G|$ is the number of vertices in OBDD graph G. This not only guarantees that the run time of reducing an OBDD is linear, but also reduces the run time of building the OBDD by limiting the number of necessary recursions. As reduction of OBDDs is an essential operation, in general, all OBDDs used in this thesis are assumed to be ROBDDs. The reducibility of an OBDD, however, depends on the ordering of the variables. If the ordering of the same Boolean function in Equation A.1 changes to $x_1 \prec x_3 \prec x_2 \prec x_4$, the size of the ROBDD becomes considerably larger compared to the ROBDD with variable ordering $x_1 \prec x_2 \prec x_3 \prec x_4$.

The reducibility dependency on ordering is illustrated in Figure A-4. Determination of the optimal ordering that minimizes the size of the ROBDD is a coNP-complete problem. Fortunately, ordering the dependent variable near each other is a good heuristic for reducing OBDD [22]. Encoding a Boolean function with n variables as an ROBDD assures the upper bound on the size of the graph of $\lceil 2^n + 2 \rceil$, superior to classical representations such as truth tables which have a lower bound of $\lfloor 2^n \rfloor$.

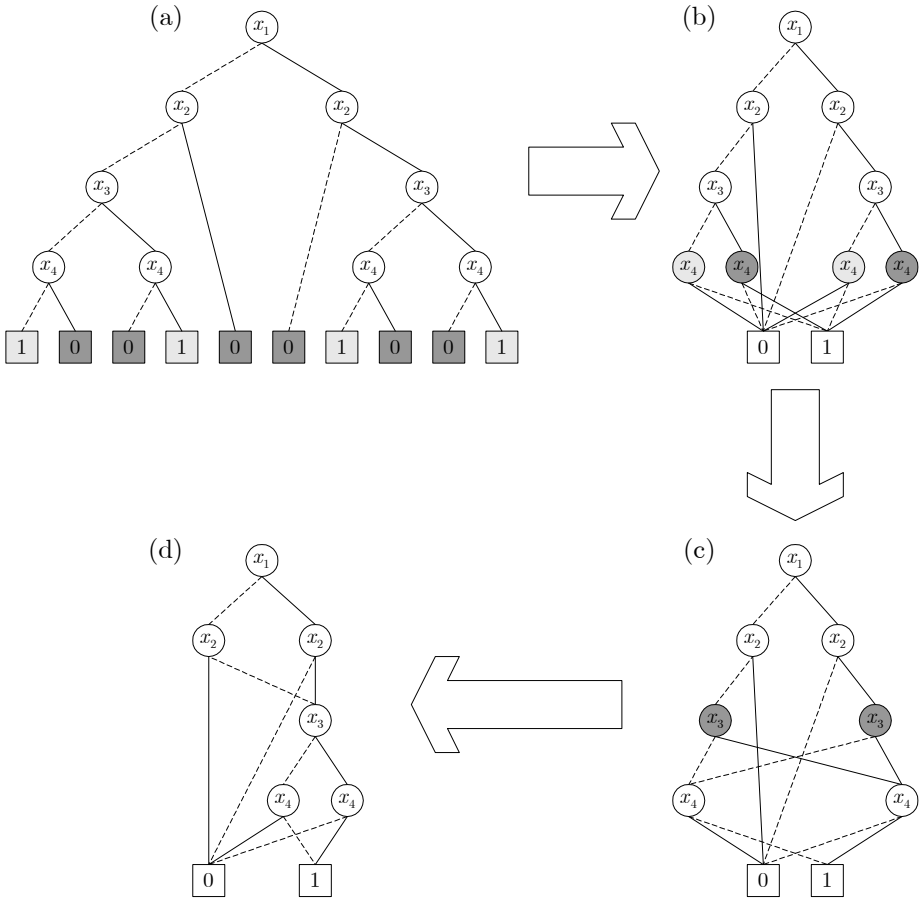

Figure A-3: Reduction steps for $((x_1 \Leftrightarrow x_2) \wedge (x_3 \Leftrightarrow x_4))$ with ordering sequence of $x_1 \prec x_2 \prec x_3 \prec x_4$. (a) OBDD of $((x_1 \Leftrightarrow x_2) \wedge (x_3 \Leftrightarrow x_4))$ contains duplicate terminals 0 and 1. (b) There are two sets of duplicate nonterminals with variable x_4. (c) Vertices that correspond to the variable x_3 are duplicating nonterminals. (d) ROBDD of $((x_1 \Leftrightarrow x_2) \wedge (x_3 \Leftrightarrow x_4))$.

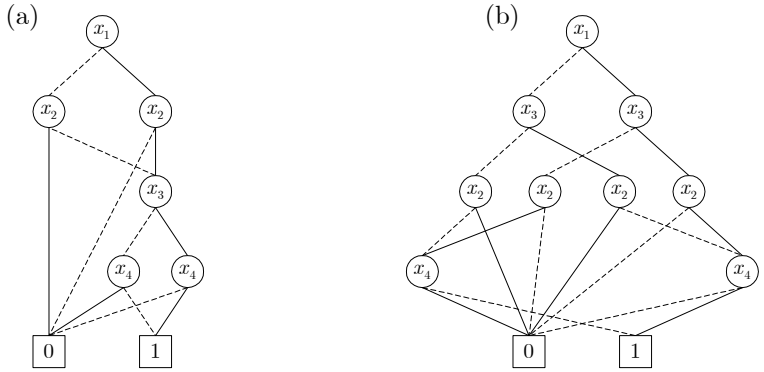

Figure A-4: ROBDD for $((x_1 \Leftrightarrow x_2) \wedge (x_3 \Leftrightarrow x_4))$ with two different ordering sequences: (a) $x_1 \prec x_2 \prec x_3 \prec x_4$ (b) $x_1 \prec x_3 \prec x_2 \prec x_4$

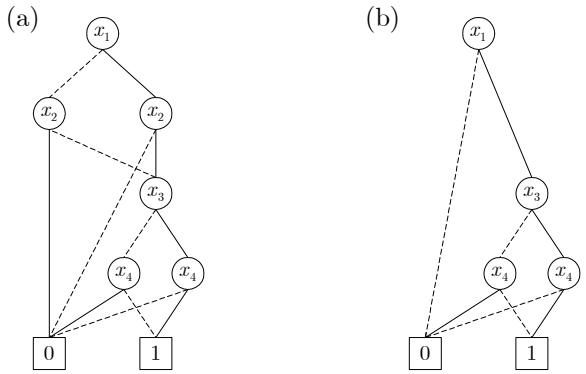

Figure A-5: Example of the `Restrict` operation. Restricting $x_2 = 1$ on Boolean function $f = (x_1 \Leftrightarrow x_2) \wedge (x_3 \Leftrightarrow x_4)$ (a) results in $f|_{x_2=1}$(b).

A.3 OBDD Operators

One of the major operations on OBDDs, `Reduce`, has been already covered. In this section, the additional operations `Restrict`, `Compose`, `Apply`, and existential and universal quantifications are briefly introduced. While more operations are available to OBDDs, these three operations form the basis for OBDD manipulation.

A.3.1 `Restrict`

The `Restrict` operation solves for $f|_{x_i=a}$. It generates the OBDD of a Boolean function f for which the value of the variable x_i is restricted to the value a. This simply involves finding all vertices that point to the vertex of x_i and redirecting them to the vertex to which $x_i = a$ points. For example, consider the f represented by Figure A-5(a). Restricting the variable x_2 to 1, or $f|_{x_2=1}$, results in the OBDD illustrated in Figure A-5(b). The time complexity of `Restrict` is $O(|G|)$ where $|G|$ is the number of vertices in OBDD graph G.

A.3.2 Apply

The Apply operation allows the evaluation of a Boolean operator on two OBDDs. This operation is carried out by applying the distributive property of Shannon expansion:

$$\begin{aligned} f_1 <op> f_2 \;\; = \;\; & \left[\neg x_i \wedge \left(f_1|_{x_i=0} <op> f_2|_{x_i=0}\right)\right] \vee \\ & \left[x_i \wedge \left(f_1|_{x_i=1} <op> f_2|_{x_i=1}\right)\right] \end{aligned} \tag{A.3}$$

where $<op>$ is an arbitrary Boolean operator of two Boolean functions f_1 and f_2. This operation can be optimized by applying dynamic programming, similar to the Reduce operation. For two OBDD graphs G_1 and G_2 that correspond to two Boolean functions f_1 and f_2 respectively, both the time complexity and size are $O\left(|G_1| \cdot |G_2|\right)$ where $|G_i|$ is the number of vertices in the OBDD graph G_i. The Apply operation forms the basis for most types of BDD manipulations. This operation not only applies to logics in which satisfiability or implication can be tested, but also to problems of sets, relations, and others that can be solved using various combinations of Boolean operators. For more details see [7].

A.3.3 Compose

The idea of the Compose operation is similar to that of the Restrict operation; however, instead of restricting a variable to a value, a variable is restricted to another Boolean function. The composition $f_1|_{x_i=f_2}$:

$$f_1|_{x_i=f_2} = \left(\neg f_2 \wedge f_1|_{x_i=0}\right) \vee \left(f_2 \wedge f_1|_{x_i=1}\right) \tag{A.4}$$

By cleverly applying the Restrict and Apply operations, the time complexity of Compose can be bound to $O\left(|G_1|^2 \cdot |G_2|\right)$, where OBDD graphs G_1 and G_2 correspond to Boolean functions f_1 and f_2, respectively, and $|G_i|$ is the number of vertices in the OBDD graph G_i. This operation is important for a function f_1 that may contain multiple occurrences of some subfunction f_2.

Table A.1: Worst Case Time Complexity of OBDD Operations [6].

Operations	Description	Time Complexity
`Reduce`	Reduce OBDD to the canonical form by removing duplicate terminals, duplicate nonterminals, and redundant tests.	$O(\|G\|)$
`Restrict`	$f\|_{x_i=a}$	$O(\|G\|)$
`Apply`	$f_1 <op> f_2$	$O(\|G_1\| \cdot \|G_2\|)$
`Compose`	$f_1\|_{x_i=f_2}$	$O(\|G_1\|^2 \cdot \|G_2\|)$

A.3.4 Quantification

Both the existential, $\exists$, and the universal, $\forall$, quantifications can be implemented using `Restrict` and `Apply` operators, where:

$$\exists x.f = (f|_{x=0}) \vee (f|_{x=1}) \tag{A.5}$$

and

$$\forall x.f = (f|_{x=0}) \wedge (f|_{x=1}) \tag{A.6}$$

The time complexity for both the existential and the universal quantifications is $O(|G|^2)$, where $|G|$ is the number of vertices in the OBDD graph G that corresponds to the Boolean function f.

A.4 Summary

The advantage of using the OBDD representation is that it is compact and the time complexity of BDD manipulations is generally low. Table A.1 lists the worst case time complexity of most widely used operations. In this table, G represents an OBDD graph with $|G|$ vertices. It should be noted that all of the operations are linear or polynomial.

This appendix has introduced OBDDs and provided some intuition on building and manipulating them. More detailed descriptions and algorithms for generating and manipulating OBDDs can be found in [6, 7].

Bibliography

[1] NASA Discovery Mission MESSENGER Preliminary Design Review. PDR Presentation, May 22–24 2001.

[2] H. R. Andersen. An Introduction to Binary Decision Diagrams. Lecture Note for 49285 Advanced Algorithms E97, October 1997.

[3] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Machi, A. Pardo, and F. Somenzi. Algebraic Decision Diagrams and Their Application. In *Proceedings of the International Conference on Computer-Aided Design*, pages 188–191, Santa Clara, CA, November 1993.

[4] G. Berry and G. Gonthier. The Synchronous Programming Language ESTEREL: Design, Semantics, Implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[5] A. Blum and M. Furst. Fast Planning Through Planning Graph Analysis. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI 95)*, volume 2, pages 1636–1642, Montréal, Canada, August 1995.

[6] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[7] R. E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Computing Surveys*, 24(3):293–318, July 1992.

[8] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang:. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, 98(2):142–170, June 1992.

[9] T. Bylander. Complexity Results for Planning. In *Proceedings of the Twlefth International Joint Conference on Artificial Intelligence (IJCAI-91)*, volume 1, pages 274–279, Sydney, Australia, August 24–30 1991.

[10] J. Casani, C. Whetsler, A. Albee, S. Battel, R. Brace, G. Burdick, P. Burr, D. Dippoey, J. Lavell, C. Leising, D. MacPherson, W. Menard, R. Rose, R. Sackheim, and A. Schallenmuller. Report on the Loss of the Mars Polar Lander and Deep Space 2 Missions. Technical Report JPL D-18709, Jet Propulsion Laboratory, California Institute of Technology, March 22 2000.

[11] A. Cimatti, F. Giunchiglia, E. Giunchiglia, and P. Traverso. Planning via Model Checking: A Decision Procedure for $\mathcal{AR}$. In *Proceedings of the Fourth European Conference on Planning (ECP'97)*, Toulouse, France, September 24-26 1997.

[12] A. Cimatti, M. Roveri, and P. Traverso. Automatic OBDD-based Generation of Universal Plans in Non-Deterministic Domains. In *Prodeedings of the Fifteenth National Conference on Artificial Intelligence (AAAI'98)*, Madison, Wisconsin, July 26–30 1998.

[13] A. Cimatti, M. Roveri, and P. Traverso. Strong Planning in Non-Deterministic Domains via Model Checking. In *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems (AIPS'98)*, Pittsburgh, Pennsylvania, June 7–10 1998.

[14] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[15] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 2nd edition, 2001.

[16] R. Dechter and J. Pearl. Tree Clustering for Constraint Networks. *Artificial Inteligence*, 38(3):353–366, April 1989.

[17] E. C. Freuder. A Sufficient Condition for Backtrack-Bounded Search. *Journal of the ACM (JACM)*, 32(4):755–761, October 1985.

[18] M. L. Ginsberg. Universal Planning: An (Almost) Universally Bad Idea. *AI Magazine*, 10(4):40–44, 1989.

[19] J. H., R. St-Aubin, A. Hu, and C. Boutilier. SPUDD: Stochastic Planning using Decision Diagrams. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence (UAI 99)*, pages 279–288, Stockholm, Sweden, 1999.

[20] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The Synchronous Dataflow Programming Language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[21] R. M. Jensen. OBDD-based Universal Planning in Multi-Agent, Non-Deterministic Domains. Master's thesis, Technical University of Denmark, June 7 1999.

[22] R. M. Jensen and M. M. Veloso. OBDD-based Universal Planning: Specifying and Solving Planning Problems for Synchronized Agents in Non-Deterministic Domains. *Journal of Artificial Intelligence Research*, 13:189–226, 2000.

[23] R. E. Korf. Planning as Search: A Quantitative Approach. *Artificial Intelligence*, 33(1):65–68, September 1987.

[24] N. G. Leveson. *Safeware: System Safety and Computers*. Addison-Wesley Publishing Company, Inc., Reading, MA, 1995. ISBN-0-201-11972-2.

[25] J. Lind-Nielsen. *BuDDy: Binary Decision Diagram Package Release 2.0*, May 2001. http://www.it-c.dk/research/buddy/.

[26] N. Muscettola, P. P. Nayak, B. Pell, and B. C. Williams. Remote Agent: To Boldly Go Where No AI System Has Gone Before. *Artificial Intelligence*, 103(1–2):5–47, August 1998.

[27] B. Pell, D. E. Bernard, S. A. Chien, E. Gat, N. Muscettola, P. P. Nayak, M. D. Wagner, and B. C. Williams. A Remote Agent Prototype for Spacecraft Autonomy. In *Proceedings of the SPIE Conference on Optical Science, Engineering and Instrumentation*, pages 74–90, Marina del Rey, CA, October 1996.

[28] M. J. Schoppers. Universal Plans for Reactive Robots in Unpredictable Environments. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence (IJCAI'87)*, volume 2, pages 1039–1046, Milan, Italy, August 1987.

[29] B. C. Williams, M. D. Ingham, S. H. Chung, and P. H. Elliott. Model-based Programming of Intelligent Embedded Systems and Robotic Space Explorers. In *Proceedings of the IEEE*, volume 91 of *1*, pages 212–237, January 2003.

[30] B. C. Williams and P. P. Nayak. A Model-based Approach to Reactive Self-Configuring Systems. In *Proceedings of Thirteenth National Conference on Artificial Intelligence (AAAI'96)*, Portland, Oregon, August 4–8 1996.

[31] B. C. Williams and P. P. Nayak. A Reactive Planner for a Model-based Executive. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI'97)*, Nagoya, Japan, August 23–29 1997.

[32] B. Yang, R. E. Bryant, D. R. O'Hallaron, A. Biere, O. Coudert, G. Janssen, R. K. Ranjan, and F. Somenzi. A Performance Study of BDD-Based Model Checking. In *Proceedings of the Second Internaitonal Conference on Formal Methods in Computer-Aided Design (FMCAD'98)*, pages 255–289, Palo Alto, CA, November 4–6 1998.