Hazard Elimination Using Backwards Reachability Techniques in Discrete and Hybrid Models

by

Natasha Anita Neogi

M.Phil. Honours, Theoretical Physics
Cambridge University, 1997

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

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

December 3rd, 2001
February 2002

Author ....
Department of Aeronautics and Astronautics
December 3rd, 2001

Certified by ....
Nancy G. Leveson, Professor
Committee Chair, Department of Aeronautics and Astronautics

Certified by ....
Nancy A. Lynch, Professor
Department of Electrical Engineering and Computer Science

Certified by ....
Eric Feron, Associate Professor
Department of Aeronautics and Astronautics

Certified by ....
Munther Dahleh, Professor
Department of Electrical Engineering and Computer Science

Accepted by ....
Wallace Vander Velde
Chairman, Department Graduate Committee

Hazard Elimination Using Backwards Reachability Techniques in Discrete and Hybrid Models

by

Natasha A. Neogi

Submitted to the Department of Aeronautics and Astronautics on February 1st, 2002 in partial fulfillment of the requirements for the Degree of Doctor of Philosophy

ABSTRACT

One of the most important steps in hazard analysis is determining whether a particular design can reach a hazardous state and, if it could, how to change the design to ensure that it does not. In most cases, this is done through testing or simulation or even less rigorous processes -- none of which provide much confidence for complex systems. Because state spaces for software can be enormous (which is why testing is not an effective way to accomplish the goal), the innovative Hazard Automaton Reduction Algorithm (HARA) involves starting at a hypothetical unsafe state and using backwards reachability techniques to obtain enough information to determine how to design in order to ensure that state cannot be reached.

State machine models are very powerful, but also present greater challenges in terms of reachability, including the backwards reachability needed to implement the Hazard Automaton Reduction Algorithm. The key to solving the backwards reachability problem lies in converting the state machine model into a controls state space formulation and creating a state transition matrix. Each successive step backward from the hazardous state then involves only one n by n matrix manipulation. Therefore, only a finite number of matrix manipulations is necessary to determine whether or not a state is reachable from another state, thus providing the same information that could be obtained from a complete backwards reachability graph of the state machine model. Unlike model checking, the computational cost does not increase as greatly with the number of backward states that need to be visited to obtain the information necessary to ensure that the design is safe or to redesign it to be safe. The functionality and optimality of this approach is proved in both discrete and hybrid cases.

The new approach of the Hazard Automaton Reduction Algorithm combined with backwards reachability controls techniques was demonstrated on a blackbox model of a real aircraft altitude switch. The algorithm is being implemented in a commercial specification language (SpecTRM-RL).

SpecTRM-RL is formally extended to include continuous and hybrid models. An analysis of the safety of a medium term conflict detection algorithm (MTCD) for aircraft, that is being developed and tested by Eurocontrol for use in European Air Traffic Control, is performed. Attempts to validate such conflict detection algorithms is currently challenging researchers world wide. Model checking is unsatisfactory in general for this problem because of the lack of a termination guarantee in backwards reachability using model checking. The new state-space controls approach does not encounter this problem.

Thesis Supervisor: Nancy G. Leveson, Ph.D.
Title: Professor of Aeronautics and Astronautics

To my Family, without whom I would not have been myself, and thus none of this would ever have been possible.
"La perfection est atteinte non quand il ne reste rien à ajouter, mais quand il ne reste rien à enlever."

Most especially to "you-know-who"! For my very own ChristopherRobin...
Pooh!" he whispered.
"Yes?"
"Nothing," he said, taking Pooh's paw. "I just wanted to be sure of you."

And to Mr. John Pivnick, my own "October Sky" story. You are living proof that grade 10 math teachers do make a difference.

When we ask advice, we are usually looking for an accomplice.

Lao Tze (604-531 B.C.)

Acknowledgements

As I come to the end of my Ph.D at MIT, I find myself with so many people deserving of my honest thanks for their interest, help and attention to my thesis, I find myself at a loss as to where to begin! I can only hope to acknowledge a fraction of the people who have contributed to making my tenure at MIT, both academic and personal, a joy and pleasure.

First and foremost, I must thank my advisor, Prof. Nancy Leveson, without whom I would never have gotten my degree. After a traumatic qualifier exam experience and a trying spring semester, her kindness, encouragement and sheer love of academic research, all of which she conveyed to me upon our first meeting, persuaded me that a Ph.D could be a very rewarding experience. I will be forever indebted to Nancy Leveson for introducing me to the exciting new field of software safety while simultaneously supporting and steering me in directions in which I could flourish both academically and personally. I cannot count the times I spent in her office, benefiting from her insight in determining what was a realistic problem, and her knack for asking the correct questions that no one else was thinking about. Her ability to mentor her students while simultaneously lending a sympathetic ear to any troubles, be they scholastic or not, will be a trait that I shall always strive to emulate.

A semester into my work on software safety, I was privileged to be introduced to Prof. Nancy Lynch. I will always stand in awe of her mathematical brilliance, and her constant patience with my attempts to formally define and prove properties in a thoroughly logical manner. Her enthusiasm towards my research, and her great attention to detail formed the cornerstone of the theoretical backbone of my thesis. I could never have articulated the algorithm which serves as an essential part of my thesis with one-tenth of the precision that I managed without her constant guidance, and perpetual willingness to challenge any point that may have been misconstrued or unclear. I can only aspire to one day achieve her level of clarity, fluency and succinctness in the language of mathematics.

I was fortunate enough to have the help of two of the best control theorists that I have ever known. Prof. Eric Feron was always ready to offer council whenever I hit a roadblock, and provided essential advice over lunches at the Royal East. His ability to provide a humourous slant to any trying aerodynamical problem was always enough to motivate me to continue! I am left continually amazed by Prof. Munther Dahleh's ability to reduce even the most difficult of problems into the simplest of concepts. I cannot quantify the amount of wisdom gleaned from our early morning cups of coffee at Toscinini's speaking of reachability, controllablity and manifold theory. Thanks are also due to Prof. J.P. Clarke, who provided a firm grounding in all things human-factors related, as well as a great deal of knowledge on all things M.I.T related.

A year and a half ago, I was gifted with the opportunity to meet “Dr.” Professor Kristina Lundqvist. Not only did she provide academic, moral and emotional support through the latter part of this degree, she is also the best roommate anyone could have hoped for! I will always treasure your thoughtfulness and friendship, along with your wit and wisdom. I can never repay all of the kindness and consideration showered on me by Karen Marais. She is easily the wisest person I have ever met, and all of the times we have shared are a pearl beyond price at a place like MIT. Karen, you have the most generous spirit of anyone I know, and you deserve the best out of life! I would like to thank Prof. Margaret Storey, for all of those pep talks, which helped me continue when I felt hopeless! I could never forget to acknowledge the help and advice provided to me by my officemate, Dr. Ed Bachelder, and his wife Laura who are a veritable whirlwinds of thought and motion! Both of your compassion and caring make you treasured friends. Mr. Marc Zimmerman was essential to addicting me to all things internet, and providing a shoulder to cry on as well as an always favourable outlook to any sad situation. Tom Reynolds and Hayley Davison are two of my very favourite people in the world, and always have an encouraging word for me whenever we meet! Your sensitivity and steadfast friendship have provided me with an anchor for my time at MIT!

It is said that it takes an entire village to raise a child, and I must concur completely, as it took the entire software engineering lab to cultivate my thesis. I must thank Mirna Daouk, for the wonderful Friday gossip sessions, and JK Srinivassen for his tea and coffee runs, both of which were indispensable. Maxim de Villepin, Israel Navarro, Masafumi and Sayori Katahira, and John Bellingham kept me sane during the past year, and helped organise countless birthday lunches creating an atmosphere of camaraderie. Victor Chong, Polly Allen, Katie Weiss, Nick DuLac Elwin Ong and Tomas Viguier have provided an invigorating new spirit to the lab this year!

Many of my closest friends from MIT I met the first year I was here, at the MVL, and we subsequently went through qualifiers together, acting to cement an already formidable bond. Miwa Hayashi is one of the most perceptive people it has ever been my pleasure to meet, and her friendship has provided a strong keystone to put myself up against during the most trying times. I would never have had the courage to take my written exams without your support. Susanne 'the mommy' Essig has my eternal admiration for being able to have it all: pass qualifiers, get married, have a baby and do your PhD! This, of course, leads directly to our resident lab genius, Joe Saleh. After four and a half years, the scope of his knowledge and the breadth of his experience leaves me completely awestruck: I can never recall a conversation with you in which I did not learn something about either you, or I, or the world in general. I am proud to have you as such a good friend.

I am indebted to Emilio Frazzoli and Kazutaka Takahashi, for their wisdom in all matters pertaining to controls. I could not imagine any acknowledgements without mentioning my dear friends Tulika Bose and Zeeya Merali, X-Stalkers from Cambridge Extrodinaire. I also have to thank Thalia Papadoupolou (classicist goddess), Misun (mommy) Yun and Anabela (Aliens?) Carvalho, Flatmates from Cambridge Extrodinaire.

And, most importantly, to my family, and to you-know-who: "...it is much more Friendly with two". You are everything to me, and will always be.

Table of Contents

List of Figures

"Then you should say what you mean," the March Hare went on.

"I do, " Alice hastily replied; "at least I mean what I say, that's the same thing, you know."

"Not the same thing a bit!" said the Hatter. "Why, you might just as well say that "I see what I eat" is the same thing as "I eat what I see!"

Lewis Carroll, Alice in Wonderland

Nomenclature

Chapter 2

δ:	a transition relation, where $\delta = Q \times Q$
Q:	a non-empty set of states
Q_0:	a non-empty set of start states
q:	single state in a state machine

Chapter 3

Chapter 4

Chapter 5

Q: Set of States of the Hazard Automaton A

q_0: Start State of the Hazard Automaton A

Q_H: Set of High Risk States of the Hazard Automaton A

q_H: High Risk State of the Hazard Automaton A, $q_H \in Q_H$

Q_L: Set of Low Risk States of the Hazard Automaton A

q_L: Low Risk State of the Hazard Automaton A, $q_L \in Q_L$

Z: Set of Hazardous States of the Hazard Automaton A

z: Hazardous State of the Hazard Automaton, $z \in Z$

Σ: Input Alphabet of the Hazard Automaton A

δ: Transition function of the Hazard Automaton A where $\delta: Q \times \Sigma \to Q$

A: Hazard Automaton where $A = (Q, q_0, Q_H, Q_L, Z, \Sigma, \delta)$

w: Finite string of input elements of Σ in A

Σ^*: Set of finite strings of elements of Σ

σ: Input symbol of the Hazard Automaton A where $\sigma \in \Sigma$

q: State of the Hazard Automaton A

P_q: Set of Predecessor States of q in the Hazard Automaton A

S_q: Set of Successor States of q in the Hazard Automaton A

δ^*: Reachability function for the Hazard Automaton A where $\delta^*: Q \times \Sigma^* \to Q$

R_q: Set of Reachable States from the state q in the Hazard Automaton A

A_q: Set of Ancestor States from the state q in the Hazard Automaton A

C: Set of Critical States for the Hazard Automaton A

q_c: Critical State for the Hazard Automaton A, has a low risk sucessor and a hazardous descendent

A': Reduced Hazard Automaton of A, where $A' = (Q, q_0, Q_L, Q_H, Z, \Sigma, \delta')$ and $\delta' \subseteq \delta$

q_i: ith state in a valid execution $\ldots q_i \sigma_i q_{i+1} \sigma_{i+1} \ldots$ of the Hazard Automaton A

σ_i: ith input in a valid execution $\ldots q_i \sigma_i q_{i+1} \sigma_{i+1} \ldots$ of the Hazard Automaton A

$P_{q, A'}$: Set of Predecessor States of q for the reduced Hazard Automaton A'

W: External Variables in a hybrid Hazard Automaton HA

X: Internal Variables in a hybrid Hazard Automaton HA

V: Variables in a hybrid Hazard Automaton HA. $V = W \cup X$.

Q: States of the hybrid Hazard Automaton HA. $Q = val(X)$.

Θ: Start states of the hybrid Hazard Automaton HA.

E: External Actions of the hybrid Hazard Automaton HA

H: Internal Actions of the hybrid Hazard Automaton HA

Ac: Actions of the hybrid Hazard Automaton HA, $Ac = E \cup H$.

a: Action of the hybrid Hazard Automaton HA, $a \in Ac$

D: Discrete transition of the hybrid Hazard Automaton HA, $D = val(X) \times Ac \times val(X)$

T: Set of valuations of V that obey prefix, suffix and concatenation closure for the hybrid Hazard Automaton HA, set of Trajectories of the hybrid Hazard Automaton HA.

τ: Trajectory of hybrid Hazard Automaton HA, $\tau \in T$

HA: hybrid Hazard Automaton, $HA = (W, X, Q, \Theta, E, H, D, T)$.

a_i: i-th action in a valid execution of the hybrid Hazard Automaton HA, $a_i \in Ac$

τ_i: i-th trajectory in a valid execution of the of hybrid Hazard Automaton HA, $\tau_i \in T$

α: Execution Fragment of a hybrid Hazard Automaton HA. $\alpha = \ldots\tau_i a_{i+1}\tau_{i+1}a_{i+2}\ldots$

$frags_{HA}$: Set of valid execution fragments of HA, $\alpha \in frags_{HA}$

$trace(\alpha)$: External behaviour of the hybrid Hazard Automaton HA during the execution fragment α

$traces_{HA}$: Set of traces of external behaviour of the hybrid Hazard Automaton HA

$firstq\tau_i$: First state in the trajectory τ_i of the hybrid Hazard Automaton HA

$lastq\tau_i$: Last state in the trajectory τ_i of the hybrid Hazard Automaton HA

T_L: Set of low risk trajectories for the hybrid Hazard Automaton HA

T_H: Set of high risk trajectories for the hybrid Hazard Automaton HA τ_c

τ_c: Critical trajectory, has a low risk trajectory sucessor and a hazardous trajectory descendent.

Alice came to a fork in the road.
"Which road do I take?" she asked.
"Where do you want to go?" responded the Chesire Cat.
"I don't know," Alice answered.
"Then," said the cat, "it doesn't really matter."

Lewis Carrol: Alice in Wonderland

CHAPTER 1

When one does something right, one only confirms what is already known: how to do it. A mistake is an indicator of a gap in one's knowledge. Learning takes place when a mistake is identified, its producers are identified and it is corrected.

R.L. Ackoff, 'Its a Mistake!', Systems Practice, 1994

One can only show the presence of errors, not their absence.

John Djikstra

Introduction

Etymologically, the word *safe* is traceable to several sources. For example, the Latin *salvus* translates into safe, whole, or healthy and is akin to *salus*, which may be translated as health or safety. The derivation from the Greek relates to the word *holos*, which means complete or entire; and the Sanskrit word *sarva* means unharmed or entire. The process by which these roots were transformed into the modern adjective *safe* becomes evident through an examination of the old French variations, *salf*, *sauf*, *sof*, and *sal*, and the variations used in Middle English, *sauf*, *saf* and *save*.

The Oxford English Dictionary (2nd Ed.), defines the adjective *safe* as:

1. Free from hurt or damage; unharmed:

   Unhurt, uninjured, unharmed; having been preserved from or escaped some real or apprehended danger. Chiefly (now only) with quasi-adverb. force after verbs of coming, going, bringing, etc.

2. Free from danger; secure:

   Not exposed to danger; not liable to be harmed or lost; secure. Of a place or thing: Affording security or immunity; not exposing to danger; not likely to cause harm or injury.Of an action, procedure, undertaking, plan, etc.: Free from risk, not involving danger or mishap, guaranteed against failure. Sometimes = free from risk of error, as in *it is safe to say.* In stronger sense: Conducive to safety.

There are, of course, a variety of satisfactory definitions for safety, any one of which can be used as a starting point for system safety considerations. The definition used in the following text for safety is derived from [71]:

The freedom from accidents and losses, in the absolute sense.

It can, of course, be argued that there is no such thing as absolute safety, and thus safety is often defined in terms of *acceptable loss*. However, the dilemma then becomes defining what loss is deemed acceptable, and to whom. Thus, for the purposes of this text, absolute freedom from loss can be regarded as the ideal state, and the actual state of the system would wish to asymptotically approach this state.

The antithesis of safety, and the villain of the text, is the *hazard.* A hazard is defined [71] as a state or set of conditions of a system (or an object) that, together with other conditions in the environment of the system (or object), will lead inevitably to an accident (loss event). It must be noted that a hazard is defined with respect to the environment of the system or component. In addition, what constitutes a hazard depends upon where the boundaries of the system are drawn. In summary, the definition of a hazard can also be considered to be somewhat arbitrary, and one of the first steps in designing a system is to decide what conditions will be considered to be hazards that need to be eliminated or controlled.

1.1 Brief Historical Overview

Until the advent of modern scientific theory, technical progress was made by a sophisticated process of trial and error. Engineers and designers learnt not only by their own mistakes but also from other people's misfortunes. This process was quite successful, as evidenced by the rapid progress made by master builders in the design of the great twelfth- and thirteenth-century cathedrals and abbeys. Admittedly, there were many building collapses, when attempts were made to build vaults too high or columns too slim, but the survival of so many of these magnificent buildings provides evidence of the development of their builders' skills.

The role of safety in society was intensified at the time of the Industrial Revolution. New sources of power, using water or steam, not only gave great opportunities for the rapid development of manufacturing technology but also provided a terrible potential for death and injury when things went wrong. The demand for new machinery and factory premises thus increased. In designing the necessary machines and buildings it had become possible to make use of the growing body of scientific knowledge, although designers still relied heavily upon past experiences. Scientific development at that time was along strongly deterministic lines; theories strove to provide an exact and unambiguous account of natural phenomena. Failure to produce such an account was invariably considered to be a limitation of the theory rather than a fundamental impossibility.

Among the first industrial artifacts to give serious concern about public safety were boiler pressure vessels. In the second half of the nineteenth century, pressure vessel explosions were almost regarded as 'acts of God', since the underlying mechanisms of failure were poorly understood. Early steamships used sea-water in the boilers, and the boilers had to be shut down at regular intervals so that the accumulated salt could be removed. The corrosive properties of salt were apparently not known about, or else not heeded. Gradually, however, codes of practice for design and operation came into use, and the needs of insurance dictated regular inspections for any defects. A boiler explosion in Massachusetts in 1905 was largely responsible for the introduction of the American Society of

Mechanical Engineering (ASME) design codes [89]. Also, a better understanding of the mechanics of metal failure evolved during and after the First World War.

The chemical industry expanded greatly following the First World War. Because of the overtly hazardous nature of many of the materials being handled, and the need to have some sort of independent assessment of the hazards of a process plant for the calculation of insurance premiums, an actuarial approach to safety assessment was adopted within the industry. Indeed, the term used to this day in the chemical industry, loss prevention, has its origins in insurance; the loss refers to the financial loss of plants, third party claims and lost production.

In the years following the Second World War, the growth in military electronics (still largely based on thermionic valves) began to generate new problems. One study revealed that only one-third of US military electronic equipment in the late 1940's was available at any given time, the remainder was under repair [107]. This difficulty lead to the appearance of reliability engineering. Electronic reliability improved greatly in the 1950's as the vacuum tube was replaced by transistors. Reliability was also of importance in spacecraft, and the high failure rate of space missions in the late 1950's and early 1960's was steadily improved in later years.

The nuclear industry grew rapidly in the years following the Second World War. The industry was exempted from the requirements of full third-party insurance coverage in some countries (e.g. the Price-Anderson Act in the U.S.A) in an effort to promote development of the industry, and because its associated hazards were not, at that time, fully understood. Although, as was pointed out [89], "all other engineering technologies have advanced not on the basis of their successes but on the basis of their failures", the nuclear industry could not afford to do likewise, because of its associated hazards. Indeed, as the scale of other technologies has increased, many technologies now have the potential to cause unacceptable damage, and *progress through failure* is seldom nowadays justifiable. Nevertheless, the nuclear industry has learned much from a number of non-catastrophic accidents, notably the Windscale fire in 1957 and the Three Mile Island accident in 1979 [111]. Much more is being learned from the Chernobyl accident in 1986, which represented, in terms of the magnitude of the consequences, about as bad a nuclear power station accident as is conceivable.

Figure 1.1: Three Mile Island

At first, the nuclear industry adopted an approach to safety assessment based on the concept of a 'maximum credible' accident. In this approach, a worst possible accident was proposed and the plant was designed to accommodate or minimize the effects of the accident. The difficulty with this approach is that it presupposes that any more severe accidents are 'incredible'. It has been suggested that a more rigorous approach to the assessment of nuclear plant safety, using probability, was more representative of circumstances. The essence of the proposal was as follows: for any given factory or other industrial installation, the acceptable frequency of accidents that may harm third parties varies inversely with the magnitude of the consequences of those accidents. It was therefore proposed that nuclear power stations should have to meet a safety criterion expressed in terms of probability and consequence.

Whilst the scope and complexity of science and technology have grown at an ever increasing rate in the last one hundred years, the full implications of these advances did not, in many instances, begin to be understood by the public until the 1980's. Advances in transport, power generation and chemicals in particular have often been 'sold' on their direct and obvious benefits, rather than on a full disclosure of their consequences. The designers of engineering systems have frequently been able to place their main priority on performance, with the consideration of safety a secondary objective [115].

The nuclear reactor incident at Three Mile Island (TMI) prompted a very critical look at the overall safety of that source of power. Paradoxically, the TMI event demonstrated to scientists and engineers that the overall safety of the plant was in many ways satisfactory, in that a major catastrophe was prevented despite the events that occurred. Not unreasonably, this optimistic interpretation was not shared by the general public [111].

1.2 System Safety

Much of the development of system safety is tied to the development of aerospace safety directly following World War II. The Air Force was experiencing many aircraft accidents in which both planes and pilots were lost. Most of these accidents were blamed on the pilots. However, industry flight engineers argued that the cause was not so simple: Safety must be designed and built into an aircraft, just as are the qualitites of performance, stability and control [71].

System safety arose out of the intercontinental ballistic missile program. When the Air Force began to develop intercontinental ballistic missiles (ICBM's), there were no pilots to blame for accidents, yet the liquid-propellant missiles frequently blew up. In the fifties, when the Atlas and Titan ICBM's were being developed, intense pressure was focused on building a nuclear warhead with delivery capability as a deterrent to nuclear war. On these first missile projects, system safety was not identified and assigned as a specific responsibility. Instead, each designer, manager, and engineer was assigned responsibility for safety. Within 18 months after the fleet of 71 Atlas F missiles became operational, four blew up in their silos during operational testing. Not only were the losses themselves costly, but the resulting investigations detected serious safety deficiencies in the system that would require extensive modifications to correct. The decision was made to retire the entire weapons system and accelerate deployment of the Minuteman missile system. Thus, a major weapon system, originally designed to be used for ten years, was in service for less than two years [106].

Aside from the economic aspects of neglecting safety requirements, the advent of nuclear fission presented a unique problem. The catastrophic consequences of an inadvertent nuclear explosion are so serious that even one accident cannot be tolerated. For safety reasons, the Atomic Energy Commission established stringent controls on the use and handling of nuclear materials. In addition, the Department of Defense (DoD), through the Defense Atomic Support Agency, maintained tight control over all nuclear weapon designs and uses. Meeting the controls of these agencies was a major influence in identifying system safety as a separate discipline in the late 1950's.

The first military specification on system safety was published by the Air Force (Ballistic Systems Division) in 1962, and the Minuteman ICBM became the first weapons system to have a contractual system safety program. The first system safety specification was a document created by the Air Force in 1966 (MIL-S-38130A). In June 1969, this became MIL-STD-882, *System Safety Program for Systems and Associated Subsystems and*

Equipment: Requirements for, and a system safety program became mandatory on all DoD-procured products and systems [1].

The space program was the second major application area to use system safety approaches in a formalized fashion. Until the Apollo 204 fire in 1967 at Cape Kennedy, in which three astronauts were killed, NASA had basically ignored the issue of system safety. The accident alerted NASA, and they commissioned the General Electric Company (among others) to develop policies and procedures that became the model for civilian aerospace safety.

As computers became increasingly important components of complex systems, concern about the safety aspects of software began to emerge in both NASA and DoD programs. Some of the earliest software safety activities were attempted on the Space Transportation System (STS) program in the 1970's.

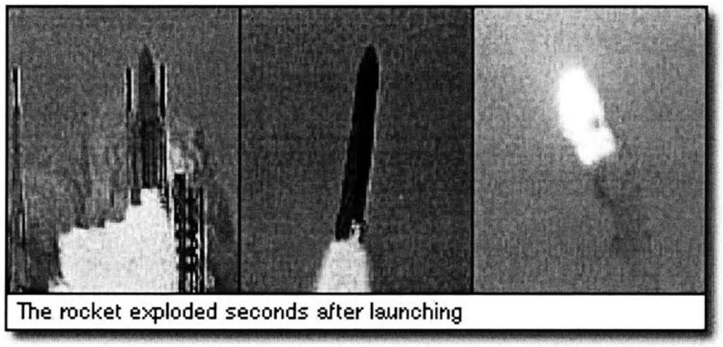


Figure 1.2: Ariane 5 Accident

A recent example of a software failure is the Ariane 5 rocket, which exploded on June 4, 1996, less than forty seconds after it was launched. The committee that investigated the accident found that it was caused by a software error in the computer that was responsible for calculating the rocket's movement. During the launch, an exception occurred when a large 64-bit floating point number was converted to a 16-bit signed integer. This conversion was not protected by code for handling exceptions and caused the computer to fail.

The same error also caused the backup computer to fail. As a result incorrect attitude data was transmitted to the on-board computer, which caused the destruction of the rocket. The team investigating the failure suggested that several measures be taken in order to prevent similar incidents in the future, including the verification of the Ariane 5 software.

Similarly, NASA's Mars Exploration program, under its "Faster, Cheaper, Better" philosophy has been plagued with software problems. The Mars Climate Orbiter (MCO) failed to achieve Mars Orbit on September 23rd 1999 due to a navigation error that resulted in the spacecraft entering Mars atmosphere instead of going into Mars orbit. Spacecraft operating data needed for navigation were provided to the Jet Propulsion Laboratory navigation team by prime contractor Lockheed Martin in Imperial units rather than specified Metric units. A lack of proper testing, and improper review of the interface specifications can be cited as the primary cause for this failure [120]. The MCO project cost approximately $115 million, not including the launch vehicle.

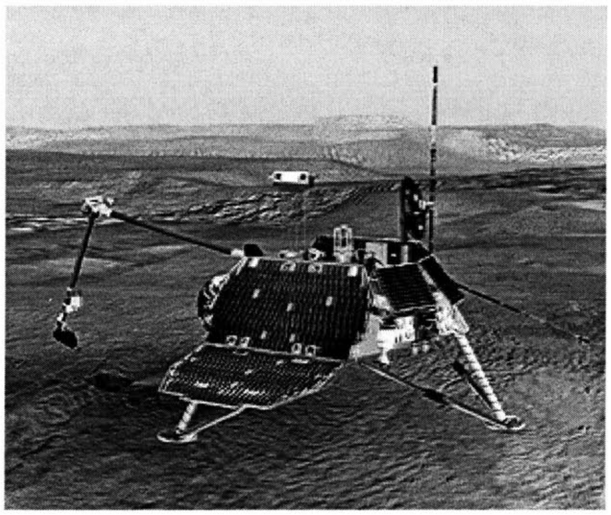


Figure 1.3: Mars Polar Lander (Simulation)

The Mars Polar Lander (MPL), along with the two Deep Space 2 microprobes, was launched on January 3rd 1999. After an 11-month cruise, the spacecraft arrived at Mars on

December 3rd 1999, targeted for a landing zone near the edge of the south polar layered terrain. The planned communication after landing did not occur, resulting in the determination that the MPL mission had failed. Extensive tests have demonstrated that the most probable cause of failure is that spurious signals were generated when the lander legs were deployed during descent. The spurious signals gave a false indication that the lander had landed, resulting in a premature shutdown of the lander engines and the destruction of the lander when it crashed into the Martian surface [120].

It is not uncommon for sensors involved with mechanical operations, such as the lander leg deployment, to produce spurious signals. For MPL, there was no software requirements to clear spurious signals prior to using the sensor information to determine that landing had occurred. During a test of the lander system, the sensors were incorrectly wired due to a design error. As a result, the spurious signals were not identified by the systems test, and the systems test was not repeated with properly wired touchdown sensors. While the most probable direct cause of the failure is premature engine shutdown, it is important to note that the underlying cause is inadequate software design and systems testing. The MPL mission cost roughly $120 million dollars in total, not counting the cost of the launch vehicle or the microprobes [120].

Clearly the need for safe and functional hardware and software systems is critical. As the involvement of such systems in our lives increases, so too does the burden for insuring their safety. Unfortunately, it is no longer feasible to shut down a malfunctioning system in order to restore safety: In many cases, a system is less safe when it is shut down, such as an aeroplane. Even when the failure is non life-threatening, the consequences of having to replace critical code or circuitry can be economically devastating.

1.3 Current Software Safety Techniques

Software in computer based control systems is ever increasing. Computer software and hardware replace more and more of the functionality of mechanical and electromechanical system parts. The traditional engineering disciplines are founded on science and mathematics, enabling modelling and prediction of different designs' behaviours. Software engineering has become, however, a craft based more on trial and error. Computers differ from regular physical systems on two key issues:

1. They exhibit discontinuous behaviour.
2. Software lacks physical restrictions (like mass, energy, size etc.) and lack structural/ functional intrinsic properties (like strength, density etc.)

The main physical entity that can be modelled and measured by software engineers is time. There exists sound work and theories on the verification of systems' temporal properties and attributes [111]. Several general approaches to software safety and reliability are addressed in the following sections.

1.3.1 Abstraction and Modularity

Having no physical limitations, complex software designs are possible and no physical effort is necessary to accomplish this complexity. Complexity is a source for design faults. Design faults are often due to a failure to anticipate certain interactions between a system's components. As complexity increases, design faults are more prone to occur as more interactions make it harder to identify all possible behaviours. The most formidable weapon against complexity is abstraction. Abstraction allows the user to concentrate on the general problem and disregard the low level details. However, a danger lies in basing abstraction on modular decomposition *in absurdum* [106]. Complexity increases if the system is decomposed modularly further then necessary, due to unforeseen interactions between modules. Similarly, the notion of information hiding, upon which Object-Oriented methodologies are based, leads to increased abstraction and reuse. However, information hiding decreases testability, and, if used too zealously, increases complexity. It also does not necessarily lead to fewer faults.

1.3.2 Robustness

Software is not a physical entity, it is purely a design construct. Software cannot be worn-out or "broken", *per se.* All system failures due to errors in the software are design faults and are built into the system from the beginning. Generally, software is not designed to be *robust*[1] since its focus is primarily on what the system should do, and not on what it should not do; as a consequence, testing usually does not cover abnormal inputs or outputs. In order for software to be robust, its state machine must satisfy the following [71]:

1. Every state must have a behaviour (transition) defined for every possible input.
2. The logical *OR* of the conditions on every transition out of any state must form a tautology[2].
3. Every state must have a software behaviour (transition) defined in case there is no input for a given period of time.

Applying robust design to software in order to accommodate all design flaws does not yield completely safe software. If an impossible or unspecified event does happen, a local reaction may have unfortunate global results and the complex interaction between modules cannot be determined.

1.3.3 Redundancy

In order for a redundant system to function properly, it must avoid common mode failures. Design faults are the main source for common mode failures, so fault tolerance against design faults seems futile. Adaptations of the redundancy concept have been applied to software, most commonly: *N*-version programming and Recovery Blocks. Both approaches use multiple version of dissimilar software produced from a common specifi-

1. Robust systems are designed to cope with unexpected inputs, changed environmental conditions and errors in the model of the external system.
2. A tautology is a logically complete expression (i.e. always true).

cation. Unfortunately, empirical studies have concluded that the benefit of using *N*-version programming is questionable [71,106]. Both approaches suffer from the same major design flaw: they try to compensate for design faults using diverse designs. Thus, they will not be able to recuperate from faults in the requirements specification and are likely to be afflicted by common mode faults relating to how people think in general.

1.3.4 Verification and Validation

The task of considering all system behaviours and all the circumstances it might encounter during operation may be intractable. Software behaviour is generally not continuous in nature: quantization errors are propagated and boundaries to the representation of numbers can affect the output. The software's execution path changes for every decision depending on whether or not a condition is true. For example, a simple sequential list of 20 *if*-statements may, in the worst case, yield 2^{20} possible execution paths. A small change in input can have a severe effect on which execution path is taken, which in turn may yield an enormous change in output [105].

1.3.5 Formal Methods

Just as traditional engineers can model their designs with different kinds of continuous mathematics, formal methods attempt to supply computer software engineers with mathematical logic and discrete mathematics as a modelling framework. Formal methods can be used in two fashions [89]:

1. They can be used as a syntax to describe the semantics of specifications which are later used as a basis for the development of systems.
2. They can be produced as in the above point, and then used as fundamental tool for the verification of the design

If both fashions are employed, then it is possible to prove the equivalency of the program and the specification. Unfortunately a proof, when possible, cannot guarantee correct functionality or safety. In order to perform a proof the *correct* behaviour of the software must first be specified in a formal mathematical language. The task of specifying the correct behaviour can be as difficult and error-prone as writing the software [71]. The difficulty comes from the fact that it cannot be known whether or not the actual system has accurately been modelled. Thus, it is impossible to ascertain whether or not the specification is complete. This distinction between model and reality attends all applications of mathematics in engineering, however, physical validation of mathematical models is possible for most engineering disciplines.

Nonetheless, using formal methods to verify correspondence between specification and design does seem like a possible pursuit to gain confidence. The fact that more than half of all software errors can be traced to the requirements and specifications problems [80,71] gives the application of formal methods some weight. The mathematical verification of large software systems is currently intractable for most cases, but may become feasible in the future with more compact and usable formal methods.

1.3.6 Testing

Software does not wear out over time. It is therefore reasonable to assume that as long as faults are uncovered, reliability increases for each fault that is eliminated. This notion relies on the supposition that maintenance does not introduce any new faults. According to many reliability growth models [32], failures are distributed exponentially with time. Initially, a system fails frequently, but after faults are discovered and amended the frequency of failures decreases. One problem with this method is that it would take years to remove a sufficient amount of errors to achieve a critical standard of reliability. For safety-critical systems where the required failure rate is 10^{-9} failures per hour, testing would have to be performed for at least 115 000 years in order to achieve the required rate. What makes matters even worse is the fact that more than half of the errors in many systems are due to ambiguous or incomplete requirements specifications. The intention of testing is often to verify that a specific input will yield a specific output, defined by the specification. No mention is made of how the system will behave in response to non-specified inputs, and many testing techniques overlook hazards that can be generated in this fashion. Thus, the confidence gained by testing software can be severely limited by the specification. This brings into focus the issues of safety and reliability: a system may be reliable, but not necessarily safe, and *vice versa.*

1.4 Safety vs. Reliability

Reliability, as defined by Leveson [71], is the characteristic of an item expressed by the probability that it will perform its required function in the specified manner over a given period of time and under specified or assumed conditions. Safety, on the other hand, is defined as the freedom from accidents and losses, in the absolute sense. There are other definitions of safety that are expressed in terms of 'acceptable risk'. However, risk, acceptable or otherwise, is merely probability taken personally; in other words, it is the science of bad mathematics.

Thus, one can say that safety and reliability are overlapping quantities, but not identical. Techniques that increase reliability, such as parallel redundancy and standby sparing, may not necessarily increase safety, and in some cases, may deteriorate safety performance. Safety can be seen as having a broader scope than failures, and failures do not necessarily compromise safety. Many accidents can occur without component failure. A system may have high reliability, yet fail catastrophically in a particular fashion or mode. Generalized probabilities and reliability analyses may not apply to specific, localized conditions, and so no conclusions can be drawn about the safety of such localized systems. More significantly, accidents are often not the result of a simple combination of component failures.

When components are operating together at a system level, safety is regarded as an *emergent* property. Reliability is a component property, unlike safety, that cannot be defined or measured without considering the environment. The events leading to an accident can be a complex combination of faults, failures and mishaps, to say nothing of ordinary contributing circumstances and coincidences. Reliability only quantifies the frequency of failures, disregarding the consequences of a failure. From a safety point of view, it is important to consider the consequences of failures, especially the failures that lead to hazards. Reliability analysis only embraces the possibility that an accident is related to a failure, it does not consider the potential damage that could result from a successful operation of the individual components.

An accident can be the result of a sequence of events, none of which involved a component failure: individual components work as specified, but together create a hazardous system state. Reliability uses a bottom-up approach to evaluate the effect of component failures on system function, while safety requires a top-down approach that evaluates how hazardous states can occur from a combination of both incorrect and correct component behaviour [71].

One of the most critical trade-offs between reliability and safety results from the fact that redundancy, used to increase reliability, will at the same time decrease safety. The more reliable a component, the more likely it is to operate spuriously. In many cases, spurious operation may be more hazardous than the failure of the system to function at all. This does not even consider the fact that, if redundancy is employed without using design diversity properly, then the system may fall prey to common mode failures. Redundant components increase complexity, which acts to decrease safety. One can even say that, as error rates in a system decrease and reliability increases, the safety of the system may be decreasing. This can be due to complacency on the behalf of the operators and/or the environment.

This is not meant to paint reliability in an unflattering light; while it cannot replace system safety, it can certainly supplement it, if used correctly. There must be a clarity in regards to the purpose of reliability engineering; that is, to improve the system's tolerance to hazardous random failures. Applying the techniques of reliability assessment towards system safety can be perilous indeed. Reliability assessment measures the probability of random failures, not the probability of hazards or accidents. Absurd risk estimates based on failure rates can result from this type of analysis. Also, if a design error is found, the simplest solution is to remove the error, not to convince someone that it will never cause an accident. Hence it can be said that the major drawback in reliability models are not what they include, but what they do *not* include.

1.5 Objectives: Can We Get to Where We Want to Go?

Software by itself is not hazardous. It can be conceived that software will be hazardous when executed on a computer, but even then there exists no real danger. A computer actually does nothing physical except generate electrical signals. In reality, hazards first occur when the computer and software start monitoring and controlling physical components. Thus, safety is a *system* property, and not a software property.

Traditionally, in order to design and assert that a safety critical system is not only correct with respect to functionality but also safe, a hazard analysis is undertaken. A hazard analysis determines what hazards are afflicting the system as a whole. Once a list of hazards has been found, a cause-effect analysis is usually performed. However, to design a safe system it is not sufficient to only identify the hazards in a system. The knowledge of their existence must also be taken advantage of during the system design process. The goal is to eliminate the existence of hazards at the lowest reasonably practical level (the ALARP principle, meaning As Low As Reasonably Practical). When the hazard cannot be eliminated, one must try to reduce the impact of its existence. If this is not possible, an attempt must be made to control the hazard. There are several safety design principles that apply to computer based systems [71].

The approach suggested in this work is to identify the hazards, then attempt to design them out of the system, using a backwards reachability technique. Basically, once a system has been modelled, and the hazards have been identified, one can attempt to trace the path of propagation of the hazards in a backwards manner. If one begins with the hazardous state, and considers all the possible states to which this hazard is a successor, one can attempt to divert the system to a non-hazardous path using design techniques. For all predecessor states of the hazard, the ability to take the hazardous path is then blocked, thereby removing the hazard from this particular behaviour of the system. Thus, the hazard can then be controlled effectively, if not completely designed out of the system. This dissertation applies this approach to actual aerospace examples in order to verify that a specific hazard has been eliminated or controlled.

1.6 Scope of Dissertation

The first chapter of this dissertation serves as an introduction to the topic of system safety, and provides motivation for the ensuing discussion and hazard elimination techniques. Traditional methods of hazard analysis are investigated in the second chapter of this thesis. However, for extremely large systems, such as the Flight Management System of an aeroplane, these methods may become impractical. Formal methods are also explored, with the domains of model checking and theorem proving being emphasized. Unfortunately, these methods are often intensely mathematical, and can be difficult to use.

Many systems nowadays have both continuous and discrete characteristics. These systems are aptly named hybrid systems. Different approaches for modelling these systems are then addressed, with the advantages and disadvantages of each being considered. A brief survey of model checking tools is conducted, and the modelling language Specification Tools and Requirements Methodology-Requirements Language (SpecTRM-RL) is introduced. The discrete modelling language SpecTRM-RL is then formally extended to encompass the modelling of hybrid systems.

The notion of reachability is then introduced, in the context of several disciplines including computer science, operations research and control systems. A technique for determining the reachability of state space controls systems is developed. Then, a general mathematical approach to establish backwards reachability as a tool of state machine hazard analysis is proposed. This novel approach involves converting state machines into state space formulations in order to use control theory techniques to determine reachability. A general bound is achieved on the complexity of the approach.

An innovative approach in order to eliminate hazards from an automaton without generating the entire backwards reachability graph of the automaton is then expounded in a theoretical manner. The notion of a hazard automaton is introduced, and the Hazard Automaton Reduction Algorithm (HARA) is formally specified. The algorithm is regarded as being optimal in the sense that it only eliminates hazardous behaviours from the system, and does not eliminate non-hazardous or potentially desirable behaviours. This is formally proved for both the discrete and hybrid cases of the algorithm. There are subtle differences when the algorithm is applied to a continuous system as opposed to the discrete case, and these are elucidated fully.

The algorithm is then applied to two real systems. One is modelled as a purely discrete system, while the other is modelled as a hybrid system. The altitude switch is a simple discrete example, and the mathematics of the algorithm is worked through explicitly. The Medium Term Conflict Detection (MTCD) is a hybrid example, and only a portion of the entire system is analyzed. Conclusions are then made as to the viability of this approach, and its scalability for large systems.

CHAPTER 2

If I have seen further [than certain other men] it is by standing upon the shoulders of giants.

Isaac Newton (1642–1727), Letter to Robert Hooke, February 5, 1675.

Literature Review

2.1 Hazard Analysis

Hazard analysis is at the heart of any effective safety program. Although hazard analysis alone cannot ensure safety, it is a necessary first step. Hazard analysis is not just performed at the start of a project or during fixed steps, it is iterative, and should be continuous throughout the life of the system. Different models allow for various types of analysis or manipulation of the model to learn more about the system. The models and analysis techniques also imply different underlying accident and human error models, which influence the hazards and causes that will be identified and considered. There is often a tradeoff between the difficulty of building and analyzing the model and the quality of information that can be derived from it. No one model or analysis technique is useful for all purposes, and more than one type may be required for a project.

2.1.1 Checklists

Checklists are a way to pass on hard-earned experience garnered from projects in engineering. They serve as a repository for mistakes, and provide feedback to the engineering process. They are the most useful in the design of a well-understood system, for which standard design features and knowledge have been developed over time. Basic checklists are simply lists of hazards or specific design features, while other, more thought-provoking variations stimulate inquiry by providing open-ended questions that require more than a yes or no answer. Checklists are commonly used in all life-cycle phases of a project, including hazard identification, design and operation. They are an excellent way to pass on lessons learned, especially for hazard identification. However, they may encourage over-reliance on the part of the user, and they may be cumbersome in length, in order to be comprehensive. Also, if the circumstances under which the checklist is employed are not carefully considered, then a more hazardous situation might be created, due to false confidence on the part of the user. Thus, more sophisticated analysis techniques than a checklist are needed for most complex systems [71].

2.1.2 Fault Tree Analysis

Fault tree analysis (FTA) is utilized extensively in nuclear industries, electronics and, of course, the aerospace industry. FTA is primarily a means of analyzing the causes of hazards, and does not aid in the identification of hazards. FTA uses Boolean logic to describe the combinations of individual faults that can constitute a predetermined hazardous event. Each level refines the one immediately above it by listing more basic events that are necessary and sufficient to cause the problem shown in the level above. Thus, FTA is a top-down search method. It has four basic steps:

1. System definition
2. Fault Tree Construction
3. Qualitative analysis
4. Quantitative analysis

System definition requires determining top events, initial conditions, existing events and impermissible events. Once the system is defined, a particular system state and top event are assumed. Then, the causal events relating to the top event and the logical relations between them are constructed. This process continues with each level of the tree being refined until primary events, or leaf nodes, are reached. Qualitative analysis can be performed by reducing the tree to a logically equivalent form showing the specific intersections of basic events sufficient to cause the initial hazard (top event). Quantitative analysis can be accomplished by assigning probabilities to the occurrence of basic events [57].

Fault tree analysis in software can be used for verification of existing code. It might be applied to the software design representations to locate problems early. Probabilistic analysis is not applicable when software logic is described by fault trees [94]. Although generic fault trees can be constructed before the details of design and construction are known, they are of limited usefulness. FTA may be applied to completed or existing systems to prove they are safe. Fault trees can help identify scenarios leading to hazards and can suggest possibilities for hazard elimination or control. Common-cause failures can best be identified from the minimum cut sets of the fault tree, and a solution for their removal can be proposed. However, the most useful fault trees can be constructed only after the system is completely designed, and thus safety measures are often difficult to implement so late in the life cycle. FTA shows little more than cause and effect relationships between events, and is not always sufficient to design an effective safety measure. The relative simplicity of a fault tree can be misleading, and simple AND/OR gates do not provide any temporal aspects to the analysis. Finally, transitions between states are not represented; partial failures and multiple failures can cause difficulties [37].

2.1.3 Management Oversight and Risk Tree Analysis

Management Oversight and Risk Tree Analysis (MORT) is basically a standard fault tree augmented by an analysis of managerial functions, human behaviour, and environmental factors. It aims to identify problems, defects, and oversights that create hazards or prevent their early identification, by use of an extensive checklist. MORT has the advantages of any checklist, but it also considers organizational, managerial, and information factors. It is not used very often due to the complexity of the checklist, which possesses 1 500 basic events or factors [109].

2.1.4 Event Tree Analysis

Event Tree Analysis (ETA) uses forward search to identify the various possible outcomes of a given initiating event, by determining all sequences of events that could follow. The initiating event might be a hazardous event, or even some circumstance external to the system. The states in the forward search are then determined by the success or failure of other components or pieces of equipment. The event tree is drawn from left to right, with branches under each heading corresponding to two alternatives:

1. Successful performance of the protection system (upper branch)
2. Failure of the protection system (lower branch)

After the tree is drawn, paths through it can be traced by choosing a branch under each successive heading, corresponding to each accident scenario.

Event trees tend to become quite large. They are usually applied using a binary state system, where each branch of the tree has one failure state and one success state. A probability can then be assigned to each branch of the event tree. If a greater number of discrete states are defined for each branch, then a branch must be included for each state. The path explosion problem quickly becomes the dominant drawback to this form of analysis. Timing issues can cause problems in event tree construction, as can possible dependencies between the various probabilities arising from common-cause failures [1].

Like FTA, ETA is appropriate only after most of the design is complete. Fault trees lay out relationships between events, while event trees display relationships between sequences of events linked by conditional probabilities. Thus, one could say that while fault trees are more powerful in identifying and simplifying event scenarios, event trees are better at handling notions of continuity. Event trees are practical when the chronology of events is stable and the events are independent of one another [70]. However, event trees can become exceedingly complex, especially when a number of time-ordered system interactions are involved [29]. A separate tree is required for each initiating event, making it difficult to represent interactions between event states in the separate trees or to consider the effects of multiple initiating events. The usefulness of event trees depends on being able to define the set of initiating events that will produce all the important accident sequences. Defining the functions across the top of an event tree and their order is difficult. To solve the ordering problem, a detailed understanding of all plant systems, and how they operate and interact, is necessary. Of course, as with fault trees, continuous, non-action systems are inappropriate for event tree analysis.

2.1.5 Cause-Consequence Analysis

Cause-Consequence Analysis (CCA) starts with a critical event and determines the causes of the event (using top-down or backwards search) and the consequences that could result from it (forwards search). The cause-consequence diagram shows both time dependency and causal relationships among events. The initiating events should be traced back to spontaneous events covered by statistical data. Several cause charts may be attached to a consequence chart. Logic symbols used in the charts to describe the relationship between events are primarily gates (AND, OR), while vertices (AND, OR, XOR, Either OR etc.) are used to describe the relations between consequences [98].

CCA shows the sequence of events explicitly, which makes the diagrams useful for studying startup, shutdown and other sequential control problems. They allow the representation of time delays, alternative consequence paths, and combination of events. They also take account of external conditions and the temporal ordering of events. Unfortunately, the diagrams can become unwieldy, separate diagrams are necessary for each initiating event, and outcomes are related only to the cause being analyzed, even though they could have been caused by other initiating events [37].

2.1.6 Hazards and Operability Analysis

Hazards and Operability Analysis (HAZOP) is based on a systems theory model of accidents that assumes accidents are caused by deviations from the design or operating intentions. The technique focuses not only on safety, but also on efficient operations. HAZOP is a qualitative technique whose purpose is to identify all possible deviations from the design's expected operation and all hazards associated with these deviations. HAZOP is able to elicit hazards in new designs as well as hazards that have not been considered previously. Hence, the hazards do not all have to be identified before the analysis, which is a major asset. HAZOP will consider several factors of a process plant:

1. The design intention of the plant
2. The potential deviations from the design intention
3. The causes of these deviations
4. The consequences of such deviations

There is an automated variant of HAZOP, called Deviation Analysis, which can be applied to software requirements specification [105].

HAZOP uses detailed process descriptions, and by the time such information is available, it is too late to make changes in the design. Thus, hazards end up being controlled by protection devices rather than removed by design changes. HAZOP does not attempt to provide quantitative results, but systematizes a qualitative approach. This method is simple and easy to use, and has an open-ended approach to identifying potential problems. Unfortunately, HAZOP is labour-intensive and is limited by the search pattern that determines the factors that will be considered. HAZOP covers hazards caused by process deviations, but still leaves out hazards that have more stable determining factors as the only contributors [110].

2.1.7 Interface Analysis

Various analysis methods are used to evaluate connections and relationships between components, including incompatibilities and the possibilities for common-cause or common-mode failures. The relationships examined can be categorized as physical, functional, or flow [49]. These analysis methods generally use structured walkthroughs to examine the interface between components and to determine whether a connection provides a path for failure propagation. Interface analyses are similar to HAZOP, but generalized somewhat, and thus have the same benefits and limitations. Effectiveness depends upon the procedures used and the thoroughness with which the analysis is applied [71].

2.1.8 Failure Modes and Effects Analysis

Failure Modes and Effects Analysis (FMEA) uses a forward search process based on an underlying chain-of-events model, where the initiating events are failures of individual components. The first step is to identify and list all components and their failure modes, considering all possible operating modes. For each failure mode, the effects on all other system components are determined along with the effect on the overall system. Then the probabilities and seriousness of the results of each failure mode are calculated. The results are documented in a table with column headings such as component, failure probability, failure mode, percent failures by mode, and effect. FMEA is appropriate when a design has progressed to the point where hardware items may be easily identified on engineering drawings and functional diagrams [71].

FMEA is effective for analyzing single units or single failures to enhance individual item integrity. It can be used to identify redundancy and fail-safe design requirements, single-point failure modes, and inspection points and spare part requirements. The strength of the technique is its completeness, but that means it is also very time consuming and can become very tedious and costly if applied to all parts of a complex design. All the significant failure modes must be known in advance, so FMEA is most appropriate for standard parts with few and well-known failure modes. The technique itself does not provide any systematic approach for identifying failure modes or for determining their effects and no real means for discriminating between alternate courses of improvement or mitigation. FMEA does not normally consider effects of multiple failures. FMEAs pay little attention to human errors in operating procedures, hazardous characteristics of the equipment, or adverse environments [48]. FMEAs can be used in safety analyses as long as their limitations are known and understood: Not all failures lead to accidents, and not all accidents are caused by failures.

On a similar note, Failure Modes, Effects and Criticality Analysis (FMECA) is basically just a FMEA with a more detailed analysis of the criticality of the failure. Two additional steps are added to the FMEA:

1. Means of control already present or proposed are determined.
2. The Findings modified with respect to these control procedures.

Sometimes a Critical Items list is generated from the results of the FMEA or FMECA. The same advantages and disadvantages apply to the FMECA as to the FMEA.

Fault Hazard Analysis (FHA) is basically a FMEA or FMECA with both a broader and more limited scope. The scope is broadened by considering human error, procedural deficiencies, environmental conditions, and other events that might result in a hazard caused by normal operations at an undesired time [38]. At the same time, the scope is more restricted than that of a FMEA or FMECA, since supposedly only failures that could result in accidents are considered. Two new pieces of information are added about upstream and downstream effects:

1. Upstream components that could command or initiate the fault in question
2. Factors that could lead to secondary failures

Like FMEAs and FMECAs, FHA primarily provides guidance on what information to obtain, but it provides no help in actually getting that information. It also tends to concentrate primarily on single events or failures [71].

2.1.9 State Machine Hazard Analysis

A state machine is a model of the states of a system and the transitions between them. State machine models are often used in computer science. The main problem with such models is that a large number of states must be specified in order to model any realistically complex system. If a model of a system is created and its entire state space generated, it is theoretically possible to determine if the state space contained any hazardous states. This involves a forward search that starts from the initial state of the system, generates all possible paths from that state, and determines whether any of them are hazardous.

State Machine Hazard Analysis (SMHA) was first developed to identify software-related hazards. SMHA can be used to analyze a design for safety and fault tolerance, to determine software safety requirements (including timing) directly from the system design, to identify safety-critical software functions, and to help in the design of failure detection and recovery procedures as well as fail-safe requirements. SMHA works on a model, not the design itself, thus it can be used at any stage of the life cycle, including early in the conceptual stage to evaluate alternative designs and design features.

SMHA's most important limitation is that a model must be built, which may be difficult and time consuming. A second limitation of SMHA is that the analysis is performed on a model, not on the system itself–it will apply to the as-built system only if the system matches the model. Other types of mathematical models, such as logic or algebraic models of software or systems, also could be used for hazard analysis by using mathematical proof methods to show that the models satisfy the safety requirements. The most important limitation of these algebraic and logic languages is that they are usually very hard to learn and use. In addition, models and languages used must match the way that engineers think about the systems they are building, or the translation between the engineer's or expert's mental model and the written formal model will be error prone. The advantage of using state machine models is that they seem to match the internal models many people use in trying to understand complex systems [71].

Recall that SMHA initially involved a forward search from the initial state to determine whether or not any hazardous states could be reached. However, the enormous number of states makes this approach impractical, even with computerization. In order to avoid this state explosion problem, models that abstract away from all the states to a smaller number of higher-level states, from which the entire state machine can be generated, must be used. The complete state space may never be generated, but many properties of the state-space can be inferred from the higher-level model.

Furthermore, backwards and top-down search methods would entail starting with the hazardous states and working backward from each to see if the initial state is reached. The number of backwards paths is enormous for most real systems, even if only those ending in hazardous states is considered. A practical solution is to start from the hazardous state and only work far enough back along the paths to determine how to change the model to make the hazardous state unreachable. Only a small number of the states will need to be generated in most cases. The drawback, although not serious, is that the hazardous states eliminated from the design might not actually have been reachable, so more hazards may be eliminated than were actually present.

2.2 Model Checking

Model checking is a technique for verifying finite-state concurrent systems such as sequential circuit designs and communication protocols. It has a number of advantages over traditional approaches that are based on simulation, testing, and deductive reasoning. In particular, model checking is automatic and usually quite fast. Also, if the design contains an error, model checking will produce a counter-example that can be used to pinpoint the source of the error.

The main challenge in model checking is dealing with the state space explosion problem. This problem occurs in systems with many components that can interact with each other or systems with data structures that can assume many different values. In such cases, the number of global states can be enormous.

A state machine is a model for computational processes. Systems of interacting components are best modelled as some kind of combinable state machines. Complex systems must sometimes be viewed at a high level of abstraction. A state machine consists of:

$$\begin{aligned} Q, &\quad \text{a non-empty set of states} \\ Q_0 \subseteq Q, &\quad \text{a non-empty set of start states} \\ \delta \subseteq Q \mathrm{x} Q, &\quad \text{a translation relation} \end{aligned} \tag{2.1}$$

and we can write:

$$q \rightarrow q' \quad \text{to denote} \quad (q, q') \in \delta \tag{2.2}$$

Now, a state machine (Q, Q_0, δ) is deterministic if: $|Q_0| = 1$ and $\forall q \in Q, \exists$ at most one q' such that $q \rightarrow q'$. That is, the next state is always determined by the previous state, and the start state is unique. Otherwise, the state machine is non-deterministic. The behaviour of the computational process is then modelled by executions of the state machine. An execution is a (possibly infinite) sequence $q_0, q_1, q_2, \ldots$ such that:

$$\begin{aligned} q_0 \in Q_0 &\quad \wedge \\ \forall i \geq 0[q_i \rightarrow q_{i+1}] & \end{aligned} \tag{2.3}$$

A state is reachable if it is the final state in some finite-length execution. The set of reachable states of a state machine are of interest, as they define the possible behaviours of the state machine.

Although the restriction to finite state systems may seem to be a major disadvantage, model checking is applicable to several very important classes of systems. In some cases, systems that are not finite state may be verified using model checking in combination with various abstraction and induction principles. Finally, in many cases errors can be found by restricting unbounded data structures to specific instances that are finite state.

2.2.1 Model Checking Process

Applying model checking to a design involves three separate tasks:

1. Modelling
2. Specification
3. Verification

The first task, modelling, involves converting the design into a formalism accepted by a model checking tool. In many cases, this is simply a compilation task. In other cases, due to limitations on time and memory, the modelling of the design may require the use of abstraction to eliminate unnecessary details. After the system has been modelled, it becomes necessary to state the properties that must be satisfied by the design. The specification is usually given in some logical formalism. It is common to use temporal logic to state behaviour of the system as it evolves over time. Model checking provides a means for checking that the model of the design satisfies a given specification, but it is impossible to determine, via model checking, whether the given specification covers all the properties that the system should satisfy. Thus, completeness of the design is an external issue. Finally, in an ideal model checker, verification of the properties is automatic. However, in practice, it often involves human assistance. For instance, the analysis of verification results must be done manually. In the case of a negative result, an error trace is generated. The error-trace is a counter-example that illustrates a behaviour of the model which violates the property being checked. It requires human intuition to determine whether or not the error trace has resulted from an actual behavioural violation, or from incorrect modelling of the system or specification. A final possibility is that the verification task will fail to terminate due to the size of the model [25].

2.2.2 Temporal Logic

Temporal logics allow for the ordering of events in time without introducing time explicitly. Temporal logics are often classified according to whether time is assumed to have a linear or a branching structure. The meaning of a temporal logic formula is always determined with respect to a labelled state-transition graph, called a Kripke structure [58].

Several researchers, including Burstall [21], Kroger [62], and Pnueli [102], have proposed using temporal logic for reasoning about computer programs. However, Pnueli [102], was the first to use temporal logic for reasoning about concurrency. His approach involved proving properties of the program under consideration from a set of axioms that described the behaviour of the individual statements in the program. However, since proofs were constructed by hand, the technique was often difficult to use in practice.

The introduction of temporal-logic model checking algorithms by Clarke and Emerson [22,39] in the early 1980's allowed this type of reasoning to be automated. At roughly the same time, Quielle and Sifakis [103] gave a model checking algorithm for a subset of computation tree logic, but they did not analyze its complexity. Later, Clarke, Emerson and Sistla [22] devised an improved algorithm that was linear in the product of the length of the formula and the size of the state transition graph. The algorithm was implemented in the EMC model checker, which was widely distributed and used to check a number of network protocols.

Pnueli and Lichtenstein [78] reanalyzed the complexity of checking linear time formulas and discovered that although the complexity appears exponentially in the length of the formula, it is linear in the size of the global state graph. Based on observation, they argued that the high complexity of linear-time model checking might still be acceptable for short formulas.

Alternative techniques for verifying concurrent systems have been proposed by a number of other researchers. Many of these approaches use automata for specifications as well as for implementations. The implementation is checked to see whether its behaviour conforms to that of the specification. Because the same type of model is used for both implementation and specification, an implementation at one level can also be used as a specification at the next level of refinement.

The use of language containment is implicit in the work of Kurshan [63], which ultimately resulted in the development of a powerful verifier called COSPAN [50]. Vardi and Wolper [117] first proposed the use of ω-automata (automata over infinite words). They showed how the linear temporal logic model checking problem could be formulated in terms of language containment between ω-automata. Other notions of conformance between the automata have also been considered, including observational equivalence and various refinement relations [28].

2.2.3 Symbolic Algorithms

In the fall of 1987, McMillan [95], a graduate student at Carnegie Mellon University, realized that by using a symbolic representation for the state transition graphs, much larger systems could be verified. The new symbolic representation was based on Bryant's ordered binary decision diagrams (OBDD) [18]. OBDD's provide a canonical form for boolean formulas that is often substantially more compact than the conjunctive or disjunctive normal form, and very efficient algorithms have been developed for manipulating them. This model checking system developed by McMillan for his doctoral thesis is called Symbolic Model Verifier (SMV) [95]. It is based on a language for describing hierarchical finite state concurrent systems. Programs in the language can be annotated by specifications expressed in temporal logic. The model checker extracts a transition system represented as an OBDD from a program in the SMV language and uses an OBDD-based search algorithm to determine whether the system satisfies its specification.

A number of other researchers have independently discovered that OBDDs can be used to represent large state-transition systems. Coudert, Berthet and Madre [30] have developed an algorithm for showing equivalence between two deterministic finite-state automata by performing a breadth-first search of the state space of the product automata.

They use OBDD's to represent the transition functions of the two automata in their algorithm. Similar algorithms have been developed by Pixley [101]. In addition, several groups including Bose and Fisher [14], Pixley [101] and Coudert, Madre, and Berthet [31] have experimented with model checking algorithms which use OBDDs. In related work Bryant, Seger and Beatty [19] have developed an algorithm based on symbolic simulation for model checking in a restricted linear time logic.

2.2.4 Model Checking for Software

Verifying software causes some problems for model checking. Software tends to be less structured than hardware. In addition, concurrent software is usually asynchronous, that is, most of the activities taken by different processes are performed independently. without a global synchronizing clock. For these reasons, the state explosion phenomenon is a particularly serious problem for software. Consequently, model checking has been used infrequently for software verification.

The most successful technique, to date, for dealing with these software model checking problems is based on partial order reduction [45,100]. This technique exploits the independence of concurrently executed events. Two events are independent of each other when executing them in either order results in the same global state. The most common model for representing concurrent software is the interleaving model, in which all of the events in a single execution are arranged in a linear order called an interleaving sequence.

Thus, concurrently executed events appear arbitrarily ordered with respect to one another. The partial order reduction technique makes it possible to decrease the number of interleaving sequences that must be considered. When a specification cannot distinguish between two interleaving sequences that differ only by the order in which concurrently executed events are taken, it is sufficient to analyze only one of them. As a result, the number of states that are needed for model checking is reduced [116].

The idea of reducing the state space by selecting only a subset of the ways one can interleave independently executed transitions has been studied by many researchers. One of the first researchers to propose such a reduction technique was Overman [99]. However, he only considered a restricted model of concurrency that did not include looping and nondeterministic choice. The proof system of Katz and Peled [60] suggests using an equivalence relation between interleaving sequences that correspond to the same partially ordered execution. Their system includes proof rules for reasoning about a selection of interleaved sequences rather than all of them. Model checking algorithms that incorporate the partial order reduction are described in several different papers. The *stubborn sets* of Valmari [116], the *persistent* sets of Godefroid [44], and the *ample* sets of Peled [100] differ on the actual details, but contain many similar ideas.

2.2.5 State Explosion: A Way Forward

Although symbolic representations and partial order reduction has greatly increased the size of systems that can be verified, many realistic systems are still too large to be handled. Thus, it is important to find techniques that can be used in conjunction with the symbolic methods to extend the size of the system that can be verified.

Compositional reasoning exploits the modular structure of complex protocols [27]. Many finite state systems are composed of multiple processes running in parallel. The specifications for such systems can be decomposed into properties that describe the behaviour of small parts of the system. An obvious strategy is to check each of the local properties, using only the part of the system that the property describes. If it is possible to show that the system satisfies each local property, and if the conjunction of the local properties implies the overall specification, then the complete system must satisfy this specification as well. If there are interdependencies in the components, a form of assume-guarantee reasoning can be employed. When proving a property about one component, assumptions are made about the behaviour of all the other components. The assumptions must then be proved when the correctness of the other components is established [47].

Symmetry can also be used to reduce the state explosion problem [24]. Finite state concurrent systems frequently contain replicated components or structures. Having symmetry in a system implies the existence of a non-trivial permutation group that preserves the state transition graph. Such a group can be used to define an equivalence relation on the state space of the system and to reduce the state space. The reduced model can be used to simplify the verification of the properties of the original model express by a temporal logic formula.

Induction involves reasoning automatically about entire families of finite-state systems [26]. Such families can arise in the design of reactive systems in software, as well as hardware. A process control system can be parameterized, defining an infinite family of systems. The goal is to prove that every system in a given family satisfies some temporal logic property. In general the problem is undecidable, but it is possible to provide a form of invariant process that represents the behaviour of an arbitrary member of the family. Using the invariant, the property can be checked for all members of the family at once. An inductive argument is then used to verify that the invariant is an appropriate representative.

Finally, the technique employed to the greatest advantage is called abstraction [7,27]. This technique appears to be essential for reasoning about reactive systems that involve data paths. The use of abstraction is based on the observation that the specifications of systems that include data paths usually involve fairly simple relationships among the data values in the system. The abstraction is usually specified by giving a mapping between the actual data values in a system and a small set of abstract data values. By extending the mapping to states and transitions, it is possible to produce an abstract version of the system under consideration. The abstract system is often much smaller than the actual system, and as a result it is usually much simpler to verify properties at the abstract level.

Thus, it seems plausible, that by a combination of clever modelling techniques, and assiduously chosen abstracted state variables, it is possible to generate an algorithm that would be able to check a given design, be it software or hardware, for the presence of identifiable hazards.

CHAPTER 3

The sciences do not try to explain, they hardly even try to interpret, they mainly make models. By a model is meant a mathematical construct which, with the addition of certain verbal interpretations, describes observed phenomena. The justification of such a mathematical construct is solely and precisely that it is expected to work.

John Von Neumann (1903–57)

Process Control System Modelling

The hazard analysis procedure is influenced by the underlying model of the system being considered. Thus, when a process control system is being modelled, careful consideration must be given to how elements are modelled. For instance, software is primarily modelled as a discrete system, making logical changes given the state of the process system and the inputs to the system. However, software is employed to control processes, such as the flight of an aeroplane, which directly involve continuously evolving variables as time progresses. Depending on how the software and system are modelled, certain hazards can be masked. Thus, a decision must be made, whether or not to model a component as being discrete or continuous. Systems which have both discrete and continuous components are commonly referred to as *hybrid* systems.

3.1 Discrete (Logical) System Modelling

Consider a machine that can exist in any one of a number of different states. It changes state depending on an input and its current state. Such a machine is called a *finite automaton*, a simple idealized computer. Finite automata are defined in terms of their states, the inputs that they allow, and their reaction to the inputs. Finite automata come in two types, deterministic and non-deterministic, depending on how well defined the ability to change states is. The main characteristics of finite automata are that they have discrete inputs and outputs, and they have a finite number of internal states.

3.1.1 Deterministic Finite Automata

Formally, a deterministic finite automaton (DFA) *M* is defined as a collection of five things:

1. An input alphabet Σ.
2. A finite collection of states Q.
3. A start state $q_0 \in Q$
4. A finite collection of accept states $F \subseteq Q$
5. A transition function $\delta : Q \times \Sigma \to Q$

It should be noted that there can be zero accepting states in the automaton, that is, $F = \varnothing$. For a deterministic finite automata, the transition function δ specifies exactly one next state for each possible combination for a state and an input variable. Consequently, there is a one-to-one mapping between $Q \times \Sigma$ and Q, and the state is fully determined by the information present in the state-input pair (q_i, σ), $q_i \in Q, \sigma \in \Sigma$. Thus, a deterministic finite automaton *M* is described as:

$$M = (Q, \Sigma, \delta, q_0, F) \tag{3.1}$$

A string *x* is accepted if $\delta(q_0, x) = q \in F$. The language *L*(*M*) of a DFA *M* is defined as $L(M) = \{x | \delta(q_0, x) \in F\}$, which is the collection of all strings that move *M* from its initial state to an accepting state. One final thing to note is that finite strings of any length are regarded as being acceptable input for a DFA, however, infinite strings are not acceptable as input.

3.1.2 Non-Deterministic Finite Automata

Non-determinism is a useful concept that has a great impact on the theory of computation. When a deterministic finite automaton is in a given state and reads the next input symbol, the next state is always known, that is, it is determined. This is referred to as deterministic computation. In a non-deterministic machine, several choices may exist for the next state at any point. Non-determinism is a generalization of determinism, so every deterministic finite automaton (DFA) is automatically a non-deterministic finite automaton (NFA). The main difference between an NFA and a DFA is the transition rule. For an NFA, the transition rule associates pairs (q_i, σ), $q_i \in Q, \sigma \in \Sigma$ with zero or more next states. The rule relates the pairs (q_i, σ), $q_i \in Q, \sigma \in \Sigma$ with a collection of states. Thus, the transition relation is a rule between $Q \times \Sigma$ and Q, or a relation *on* $(Q \times \Sigma) \times Q$. In addition, consider that the empty string ε can trigger a state change in a non-deterministic finite automaton. Define $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$, to be the alphabet of the automaton with the epsilon transition. Furthermore, define $P(Q)$ to be the powerset of Q, that is, the collection of all subsets of Q.

Hence, a formal definition of a non-deterministic finite automata N is given by:

1. An input alphabet Σ.
2. A finite collection of states Q.
3. A start state $q_0 \in Q$
4. A finite collection of accept states $F \subseteq Q$
5. A transition relation $\Delta : Q \times \Sigma_\varepsilon \to P(Q)$

Thus, a non-deterministic finite automaton N is described as:

$$N = (Q, \Sigma, \Delta, q_0, F) \tag{3.2}$$

Deterministic and non-deterministic finite automata recognize the same class of languages. Thus, every nondeterministic finite automaton has an equivalent deterministic finite automaton. This is exceedingly useful, for if a property can be proved about a DFA, it can also be proven for its equivalent NFA.

3.2 Continuous (Evolving) System Modelling

For a causal system with m inputs $u_j(t)$ and p outputs $y_i(t)$ (hence m+p manifest variables), an nth order state space description is one that introduces n latent variables x_l(t) called state variables in order to obtain a particular form for the constraints that define the model. Letting:

$$u(t) = \begin{bmatrix} u_1(t)) \\ \dots \\ u_m(t) \end{bmatrix}, y(t) = \begin{bmatrix} y_1(t)) \\ \dots \\ y_p(t) \end{bmatrix}, x(t) = \begin{bmatrix} x_1(t)) \\ \dots \\ x_n(t) \end{bmatrix} \tag{3.3}$$

an nth order state space description takes the form:

$$\dot{x}(t) = f(x(t), u(t), t) \tag{3.4}$$

$$y(t) = g(x(t), u(t), t) \tag{3.5}$$

where equations (1.4) and (1.5) are, respectively, the state evolution equations and instantaneous output equations.

A key feature of a state-space description is the *state property*:

> Given the initial state $x(t_0)$ and input $u(t)$ for $t_0 \leq t < t_f$ (with t_0 and t_f arbitrary), the output $y(t)$ can be computed for $t_0 \leq t < t_f$ and the state $x(t)$ for $t_0 < t \leq t_f$.

Thus, the state at any time t_0 summarizes everything about the past that is relevant to the future. Note that the state property holds for functions $f(.)$ that are well behaved enough for the state evolution equations to have a unique solution for all inputs of interest and over the entire time axis. Furthermore, if the functions $f(.)$ and $g(.)$ are both linear and time-invariant, the state space description simplifies to:

$$f(x(t), u(t), t) = Ax(t) + Bu(t) \tag{3.6}$$

$$g(x(t), u(t), t) = Cx(t) + Du(t) \tag{3.7}$$

It is of note that if a continuous time system is implemented digitally, the continuous system is sampled and a discrete system is obtained. Thus, the evolution of the discrete sampled system is formulated by replacing:

$$\dot{x}(t) = x(t+1), t \in \text{Positive Integers} \tag{3.8}$$

3.3 Hybrid System Modelling

Hybrid systems involve both continuous dynamics as well as discrete phenomena. The continuous dynamics of hybrid systems may be continuous-time, discrete-time, or mixed (sampled-data), but are generally given by differential equations. The discrete-variable dynamics of hybrid systems are generally governed by a digital automaton, or input-output transition system with a countable number of states. A hybrid system can be assumed to be a run with a sequence of steps. Within each step the system state evolves continuously according to a dynamical law until a transition occurs. Transitions are instantaneous state changes that separate continuous state evolutions. There are several theoretical formal models for hybrid systems, a few of which are presented in the following sections.

3.3.1 Hybrid Input/Output Automaton

The Hybrid Input/Output Automaton (HIOA) of Lynch [88], is based on the concept of infinite state machines whose states can change by discrete actions or by continuous trajectories. The discrete transitions are labelled with actions, which allows for the synchronization of transitions of different automata when they are composed in parallel. The evolution described by a trajectory may be described as either a continuous or discontinuous function. The variables and actions are divided into the categories of internal and external. External behaviour of an automaton is characterized by hybrid traces, which are the external actions and trajectories of the evolving external variables. The actual model is composed of the seven-tuple:

$$A = (W, X, Q, \Theta, E, H, D, T) \tag{3.9}$$

where W is defined as a set of external variables and X is defined as a set of internal variables, disjoint from each other. V is defined as: $V = W \cup X$. The set of states, Q, is defined as $Q \subseteq val(X)$. The non-empty set of start states, Θ, is defined as $\Theta \equiv \mathrm{val}(X)$. The set E of external actions and the set H of internal actions are disjoint from each other. Ac is defined as $Ac = E \cup H$. The set D, of discrete transitions is defined as $D = \mathrm{val}(X) \times Ac \times \mathrm{val}(X)$. Finally, T is the set of trajectories for V, which obey prefix, suffix, concatenation and initial external valuation closure (i.e. point trajectories). This model allows one automaton to be implemented by another, which is useful when modelling both the system and required properties, as well as for stepwise refinement, where different levels of abstraction are used. The model includes the notions of implementation, simulation, composition and hiding operations, and receptiveness.

3.3.1.1 Executions and Traces

Sets of trajectories in HIOAs are described using differential and algebraic equations and inclusions. Let us consider the time domain **T** to be **R**, and τ to be a fixed trajectory over some set **V** where $v \in$ **V** is a variable for the HIOA. If we misuse the variable name v to denote the *projection* of the trajectory τ on the variable v, we have a means of expressing the value of the variable v at all times during the trajectory τ. Similarly, if we view any expression e containing variables from **V** as a function over the domain of the trajectory τ, we can say that τ satisfies the algebraic equation:

$$v = e \tag{3.10}$$

which means that the constraint on the variables expressed by the equation $v = e$ holds for each state on the trajectory τ. Furthermore, suppose that the expression e is integrable. We can say that τ satisfies:

$$\dot{v} = e \tag{3.11}$$

iff for every time t in the domain of the trajectory,

$$v(t) = v(0) + \int_0^t e(t')dt' \tag{3.12}$$

This interpretation of the differential equation makes sense even at points where v is not differentiable.

An execution fragment of a hybrid automaton A is an action-trajectory sequence:

$$\alpha = \tau_0 a_1 \tau_1 a_2 \ldots \tag{3.13}$$

where:

1. Each τ_i is a trajectory in T and
2. If τ_i is not the last trajectory in α then the last state in τ_i can map, under some action a_{i+1}, to a first state in some trajectory τ_{i+1}.

An execution fragment records all the details of a particular run of a system, including all the discrete state changes and all the changes to the state and external variables that occur while time advances. The set of execution fragments of A is defined as $frags_A$. An execution fragment α is defined to be an execution if the first state of α is a start state of the hybrid automaton A. The set of executions of A is denoted by $execs_A$. A state is defined to be *reachable* if it is the last state of some closed execution of A.

The external behaviour of a hybrid automaton is captured by the set of "traces" of its execution fragments, which record external actions and the trajectories that describe the evolution of external variables. Formally, if α is an execution fragment, then the trace of α, denoted by $trace(\alpha)$, is the restriction of α to the external actions and external variables. A *trace fragment* of a hybrid automaton A, from a state $\mathbf{x}$ of A, is the trace of an execution fragment of A whose first state is $\mathbf{x}$. A *trace* of A is a trace fragment of A from a start state of A, that is the trace of an execution of A. The set of traces of A is denoted by $traces_A$.

Hybrid automata A_1 and A_2 are comparable if they have the same external interface (i.e. W_1=W_2 and E_1=E_2). If A_1 and A_2 are comparable, then we can say that A_1 implements A_2 if the traces of A_1 are included among those of A_2 (i.e. $traces_{A_1} \subseteq traces_{A_2}$).

3.3.1.2 Simulation Relations

Simulation relations between hybrid automata may be used to show that one hybrid automaton implements another, in the sense of inclusion of sets of traces.

Let A and B be comparable hybrid automata. A *simulation* from A to B is a relation $R \subseteq Q_A \times Q_B$ satisfying the following conditions, for all states x_A and x_B of A and B respectively:

1. If $x_A \in \Theta_A$ then there exists a state $x_B \in \Theta_B$ such that $x_A R x_B$.
2. If $x_A R x_B$ and α is an execution fragment of A consisting of one discrete action surrounded by two point trajectories, with the first state of α being x_A, then B has a closed execution fragment β with the first state of β being x_B, $trace(\beta) = trace(\alpha)$, and the last state of α maps to the last state of β via the relation R.
3. If $x_A R x_B$ and α is an execution fragment of A consisting of one trajectory, with the first state of α being x_A, then B has a closed execution fragment β with the first state of β being x_B, $trace(\beta) = trace(\alpha)$, and the last state of α maps to the last state of β via the relation R.

The definition of a simulation from A to B yields a correspondence for open trajectories of A. We state the following theorem without proof:

Theorem 1:

Let A and B be comparable hybrid automata and let R be a simulation from A to B. Then $traces_{A_1} \subseteq traces_{A_2}$.

The proof of the preceding theorem can be found in Lynch et al. [84].

3.3.1.3 Composition

The operation of parallel composition for hybrid automata allows an automaton representing a complex system to be constructed by composing automata representing individual system components. The composition operation of Lynch et al. [84] identifies external actions with the same name in different component automata, and likewise for external variables. When any component automaton performs a discrete step involving an action a, so do all component automata that have a in their action sets. Likewise, when any component automaton performs a trajectory involving a particular evolution of values for an external variable v, then so do all component automata that have v in their variable set.

Composition is defined by Lynch et al. [84] as a partial, binary operation on hybrid automata. Since internal actions of an automaton A_1 are intended to be unobservable by any other automaton A_2, we allow A_1 to be composed with A_2 only if the internal actions of A_1 are disjoint from the actions of A_2. Similarly, we require disjointness of the internal variables of A_1 and the variables of A_2. Formally, the hybrid automata A_1 and A_2 are compatible if $H_1 \cap A_2 = H_2 \cap A_1 = \varnothing$ and$X_1 \cap V_2 = V_2 \cap X_1 = \varnothing$.

If A_1 and A_2 are compatible, then their composition $A_1 \| A_2$ is defined to be the structure $A = (W, X, Q, \Theta, E, H, D, T)$ where:

1. $W = W_1 \cup W_2$ and $X = X_1 \cup X_2$
2. $Q = \{x \in val(X) | (x \text{ restricted to } X_1) \in Q_1 \wedge (x \text{ restricted to } X_2) \in Q_2\}$
3. $\Theta = \{x \in val(X) | (x \text{ restricted to } X_1) \in \Theta_1 \wedge (x \text{ restricted to } X_2) \in \Theta_2\}$
4. $E = E_1 \cup E_2$ and $H = H_1 \cup H_2$
5. For each $x, x' \in Q$ and each $a \in A$, $x \xrightarrow{a_A} x'$ iff for i=1,2, either
   - $a \in A_i$ and $x \text{ restricted to } X_i \xrightarrow{a_i} x' \text{ restricted to } X_i$ or
   - $a \notin A_i$ and $x \text{ restricted to } X_i = x' \text{ restricted to } X_i$
6. $T \subseteq trajectories(V)$ is given by $\tau \in T \Leftrightarrow \tau \text{ restricted to } V_1 \in T_1 \wedge \tau \text{ restricted to } V_2 \in T_2$

Another theorem stated without proof is:

Theorem 2:

If A_1 and A_2 are hybrid automata, then $A_1 \| A_2$ is a hybrid automata.

Again, the proof is found in Lynch et al. [88]. A form of projection lemma can be derived from this, which says that the executions of a composition of hybrid automata project to give the executions of the component automata.

3.3.1.4 Hiding

There are two hiding operators for hybrid automata, one which hides external actions, and another which hides external variables. The hiding operation reclassifies external actions or external variables as internal actions or internal variables. We shall call the action hiding function ActHide and the variable hiding function VarHide.

1. If $E \subseteq E_A$, then ActHide(E,A) is the hybrid automaton B that is equal to A except that $E_B = E_A - E$ and $H_B = H_A \cup E$

2. If $W \subseteq W_A$, then VarHide(W,A) is the hybrid automaton B given by:
   - $W_B = W_A - W$
   - $X_B = X_A \cup W$
   - $Q_B = \{x \in val(X_B) | (x \text{ projected on } X_A) \in Q_A\}$
   - $\Theta_B = \{x \in val(X_B) | (x \text{ projected on } X_A) \in \Theta_A\}$
   - $D_B = \{(x, a, x') \in val(X_B) \times A_B \times val(X_B) | (x \text{ restricted to } X_A, a, x' \text{ restricted to } X_A) \in D_A\}$
   - $E_B = E_A, H_B = H_A, T_B = T_A$

3.3.1.5 Hybrid Input/Output Automaton

A hybrid input/output automaton (HIOA) A is a 5-tuple (H, U, Y, I, O) where:

1. $H = (W, X, Q, \Theta, E, H, D, T)$ is a hybrid automaton.
2. U and Y partition W into input and output variables respectively
3. I and O partition E into input and output actions, respectively

The following additional axioms are satisfied:

- Input Action Enabling: For every $x \in Q$ and every $a \in I$, there exists $x' \in Q$ such that $x \rightarrow x'$
- Input Trajectory Enabling

Input action enabling is the input enabling condition for ordinary I/O automata. Input trajectory enabling is a new, corresponding condition for interaction over time intervals. It says that an HIOA should be able to accept any input trajectory, that is, any trajectory for the input variables, either by letting time advance for the entire duration of the input trajectory, or by reacting with a locally controlled action after some part of the input trajectory has occurred. For a more in depth discussion of the HIOA and its properties, several papers by Lynch et al. are available for reference [84,88].

3.3.2 Graphical Model (Timed Reactive Modules)

Many different approaches have been taken to modeling hybrid systems. A hybrid model can be viewed as a finite automaton that is equipped with a set of variables. The control locations of the automaton are labeled with evolution laws, and at a control location the values of the variables change continuously with time according to an associated law. The transitions of the automaton are labelled with guarded sets of assignments. A transition is enabled when the associated guard is true. The execution of the transition modifies the values of the variables according to the associated law. Each location is also labelled with an invariant condition that must hold when the control resides at the location. The model proposed by Alur et al. [2] is based on a graphical method whereby hybrid systems can be specified as graphs whose edges represent discrete transitions and whose vertices represent continuous activities. A hybrid system H consists of six components:

$$H = (Loc, Var, Lab, Edg, Act, Inv) \tag{3.14}$$

where *Loc* is a finite set of vertices called locations. *Var* is a finite set of real valued variables. A valuation v for the variables is a function that assigns a real value $v(x) \in R$ to each variable $x \in Var$. A state is a pair (l, v) consisting of a location l and a valuation $v(x) \in \text{Reals},\ x \in \text{Variables}$. Then Σ is considered to be the set of states. The Alur-Henzinger model describes *Lab* as a finite set of synchronization labels that contains the stutter label τ. *Edg* is a finite set of edges called transitions. Each transition consists of $e = (l, a, \mu, l')$ where $l \in Loc$ is a source location, $l' \in Loc$ is a target location, $a \in Lab$ is a synchronization label and $\mu = V \times V$ is a transition relation. *Act* is a labelling function that assigns to each location l a set of activities. Finally, *Inv* is an invariant function that assigns to each location l an invariant. The hybrid system *H* is *time-deterministic* if for every location $l \in Loc$ and every valuation $v \in V$, there is at most one activity $f \in Act(\ell)$ with $f(0) = v$. The activity, *f*, is denoted by $\phi_l[v]$.

3.3.2.6 Runs of a Hybrid System

At any time instant, the state of a hybrid system is given by a control location and values for all variables. The state can change in two ways [3]:

1. By a discrete and instantaneous transition that changes both the control location and the values of the variables according to the transition relation
2. By a time delay that changes only the values of the variables according to the activ-

ities of the current location.

The system may stay at a location only if the location invariant is true; that is, some discrete transition must be taken before the invariant becomes false. A run of the hybrid system H, then, is a finite or infinite sequence:

$$\rho: \quad \sigma_0 \xrightarrow[f_0]{t_0} \sigma_1 \xrightarrow[f_1]{t_1} \sigma_2 \rightarrow \tag{3.15}$$

of states $\sigma_i = (l_i, v_i) \in \Sigma$, non-negative reals $t_i \in R^{\geq 0}$, and activities $f_i \in Act(l_i)$, such that for all $i \leq 0$,

1. $f_i(0) = v_i$
2. $\forall [0 \leq t \leq t_i] f_i(t) \in Inv(l_i)$
3. the state σ_{i+1} is a transition successor of the state $\sigma_i' = (l_i, f_i(t_i))$

The state σ_i' is called a time successor of the state σ_i; the state σ_{i+1} a successor of σ_i. The set of runs of the hybrid system H is written as $[H]$.

If all activities are required to be smooth functions, then the run ρ can be described by a piecewise smooth function whose values at the points of higher-order discontinuity are sequences of discrete state changes. Also, for time deterministic systems, the subscripts f_i can be omitted from the *next* relation $\rightarrow$.

The run ρ diverges if ρ is infinite and the infinite sum $\sum_{i \geq 0} t_i$ diverges. The hybrid system is nonzeno if every finite run of H is a prefix of some divergent run of H. Nonzeno systems can be executed.

3.3.2.7 Parallel composition of Hybrid Systems

Let $H_1 = (Loc_1, Var, Lab_1, Edg_1, Act_1, Inv_1)$ and $H_2 = (Loc_2, Var, Lab_2, Edg_2, Act_2, Inv_2)$ be two hybrid system over a common set *Var* of variables. The two hybrid systems synchronize on the common set $Lab_1 \cap Lab_2$ of synchronization labels; that is, whenever H_1 performs a discrete transition with the synchronization label $a \in Lab_1 \cap Lab_2$, then so does H_2 [3].

The *product* $H_1 \times H_2$ is the hybrid system $(Loc_1 \times Loc_2, Var, Lab_1 \cup Lab_2, Edg, Act, Inv)$ such that:

- $((l_1, l_2), a, \mu, (l_1', l_2')) \in Edg \Leftrightarrow$
  1. $(l_1, a, \mu_1, l_1') \in Edg_1$ and $(l_2, a, \mu_2, l_2') \in Edg_2$
  2. Either $a_1 = a_2 = a$ OR $a_1 \notin Lab_2$ and $a_2 = \tau$, OR $a_1 = \tau$ and $a_2 \notin Lab_1$
  3. $\mu = \mu_1 \cap \mu_2$
- $Act(l_1, l_2) = Act_1(l_1) \cap Act_2(l_2)$
- $Inv(l_1, l_2) = Inv_1(l_1) \cap Inv_2(l_2)$

It follows that all runs of the product system are runs of both component systems:

$$[H_1 \times H_2]_{Loc_1} \subseteq [H_1] \text{ and } [H_1 \times H_2]_{Loc_2} \subseteq [H_2] \tag{3.16}$$

where $[H_1 \times H_2]_{Loc_i}$ is the projection of $[H_1 \times H_2]$ on Loc_i. Note that the product of two time-deterministic hybrid systems is also time-deterministic. It is stated here without proof that for every hybrid system, the set of runs is closed under prefixes, suffixes, stuttering and fusion. For more detail, see Alur et al. [2,3,4].

3.3.3 Unified Hybrid System Model

Several other models are also used in control systems [15,16]. Branicky presents a unified model for most control approaches. The discrete state space is specified as $Q = Z^{\geq 0}$ (positive integers). The continuous state space for $x(.)$ is $X = \{X_i\}_{i=0}^{\infty}$ where each X_i is a subset of some Euclidean space R^{d_i}, $d_i \in Z^{\geq 0}$. The regions $A_i, C_i, D_i \in X_i$ are all specified *a priori*. There are the autonomous jump sets, controlled jump sets and jump destination sets, respectively. Let *A*, *C*, and *D* denote the unions $\bigcup_i A_i \times \{i\}$, $\bigcup_i B_i \times \{i\}$, and $\bigcup_i D_i \times \{i\}$, $i \in Z^{\geq 0}$, respectively. The *U*, *V* be the sets of continuous and discrete controls, respectively. The following maps are assumed to be known:

1. Vector fields f_i: $(X_i \times X_i \times U \rightarrow R^{d_i})$, $i \in Z^{\leq 0}$
2. Jump transition maps G_i: $A_i \times V \rightarrow D$
3. Autonomous transition delay $\Delta_{a,i}$: $A_i \times V \rightarrow R^{\leq 0}$
4. Controlled transition delay $\Delta_{c,i}$: $C_i \times V \rightarrow R^{\leq 0}$

As shorthand, define G: $A \times V \rightarrow D$ in the obvious manner, and similarly for Δ_a and Δ_c.

The dynamics of the control system can now be described as follows. There is a sequence of pre-jump times $\{\tau_i\}$ and another sequence of post-jump times $\{\Gamma_i\}$ satisfying $0 = \Gamma_0 \le \tau_1 < \Gamma_1 < \tau_2 < \Gamma_2 < \ldots \le \infty$, such that on each interval $[\Gamma_{j-1}, \tau_j)$ with non-empty interior $x(.)$ evolves according to the differential equation:

$$\dot{x}(t) = \xi(t), \qquad t \ge 0 \tag{3.17}$$

for some X_i, $i \in Z^{\le 0}$. At the next pre-jump time (say, τ_j) it jumps to some $D_k \in X_k$ according to one of the two possibilities:

1. $x(\tau_j) \in A_i$, in which case it *must* jump to $x(\Gamma_j) = G_i(x(\tau_j), v_j) \in D$ at time $\Gamma_j = \tau_j + \Delta_{a,i}(x(\tau_j), v_j)$, $v_j \in V$ being a control input. This phenomenon is called an autonomous jump.
2. $x(\tau_j) \in C_i$ and the controller chooses to (it does not have to) move the trajectory discontinuously to $x(\Gamma_j) \in D$ at time $\Gamma_j = \tau_j + \Delta_{c,i}(x(\tau_j), x(\Gamma_j))$. This is called a controlled or impulsive jump.

For $t \in [0, \infty)$, let $[t] = max_j\{\Gamma_j | \Gamma_j \le t\}$. The vector field $\xi(t)$ of equation (3.17) is given by:

$$\xi(t) = f_i(x(t), x[t], u(t)) \tag{3.18}$$

where *i* is such that $x(t), x[t] \in X_i$ and $u(.)$ is a *U*-valued control process.

This model encapsulates all of the characteristics of Witsenhausen's Model, Tavernini's Model, the Back-Guckenheimer-Meyers Model, the Nerode-Kohn Model, the Antsaklis-Stiver-Lemmon Model and Brockett's Model. For more detail, there are several papers by Branicky et al. [15,16].

3.3.4 Temporal Logic of Actions

The Temporal Logic of Actions (TLA) can be used to model continuous systems. A temporal formula is built from elementary formulas using Boolean operators and the unary operator (defined as Always). The semantics of temporal logic is based on behaviours, where a behaviour is an infinite sequence of states. We interpret a temporal formula as an assertion about behaviours. Formally, the meaning of [F] of a formula F is a Boolean valued function on behaviours. Letting $\sigma[F]$ denote the Boolean value that the formula F assigns to the behaviour σ, we can state that σ satisfies F iff $\sigma[F]$ equals true. Thus, we can say that $\Box F$ asserts that F is always true. We can also define $\Diamond F = \neg\Box\neg F$, which asserts that F is eventually true. There are several other derived semantic operators in TLA, which can be found in [68,69]. The Raw Temporal Logic of Actions (RTLA) is obtained by letting the elementary temporal formulae be actions. An action [A] is a Boolean valued function that assigns the value s[A]s' to the pair of states s, s'. So we define s,s' to be an A step iff s[A]s' equals true. Thus, we can define [A] to be true for a behaviour iff the first pair of states in the behaviour is an A step. RTLA formulas are built from actions using logical operators and the temporal operator. TLA is a subset of RTLA, which adds the notion of stuttering steps to RTLA. The notions of liveness and fairness, as specified by Lamport, can be added as well. To finish the definition of the syntax and semantics of simple TLA, the addition of the unchanged step, which preserves the state of the function, is needed.

TLA has a limited domain of applicability. TLA is moderately useful for proving simple invariant properties of programs., and type correctness. Eventuality properties are also relatively easy to ascertain. Proving one program implements another via simulation relations is not very easy, as it is occasionally difficult to determine if one has expressed the correct relation as a valid TLA formula, and intuitive reasoning is sometimes misleading. TLA is primarily useful for specifying and verifying safety, and to some extent, liveness properties of discrete systems. This is because one can regard a safety property as specifying that something bad does not happen, and that a liveness property asserts that something good does eventually happen. Thus, temporal operators are natural constructs to frame these requirements.

The most significant drawback of TLA is that TLA properties are True or False for an individual behaviour. Thus one cannot express statistical properties of entire sets of behaviours [68].

TLA can be used to reason about discrete systems even if its behaviour depends on continuous physical values. Best and worst case time bounds on algorithms can be expressed as safety properties and proved with TLA. A real-time algorithm can be specified by conjoining timing constraints to the TLA specification of the untimed algorithm. In a real-time specification, the variable *now* is different from all of the others because the continuous nature of time is not abstracted away. The specification allows *now* to assume any of a continuum of values. The discrete states in a behaviour mean that we are observing the state of the system, and hence the value of *now* at a sequence of discrete instants.

There are of course, quantities other than time whose continuous nature we wish to specify. For instance, in an air traffic control system, we wish to represent the positions and velocities of the aircraft as continuous variables. Such a system which has inherent continuously varying quantities is a hybrid system. In general, hybrid systems are treated by TLA in a manner similar to that of real time system, except that in the specification the formula *RTNow(v)* is replaced by one that describes the changes to all variables that represent continuously changing physical quantities. The Integrate operator will allow you to specify those changes for many hybrid systems. Some systems will require different operators. For instance, describing the evolution of some physical quantities might require an operator for describing the solution for partial differential equations. Theoretically, though, if you can describe the evolution equations in a mathematical format, you can technically specify these equations in TLA [68].

3.4 Model Checking Tools for Hybrid Systems

3.4.1 HyTech

HyTech is a tool that allows automatic model verification for linear hybrid systems, a subclass of hybrid systems. Given a temporal requirement, HyTech computes the condition under which the requirement is satisfied by a linear hybrid system. Hybrid systems are specified as collections of automata with discrete and continuous components, and temporal requirements are verified by symbolic model checking. If the verification fails, then HyTech generates a diagnostic error trace. HyTech was developed by Henzinger, Ho, and Wong-Toi [52,53].

A hybrid system typically consists of several components that operate concurrently and communicate with each other. The component automata coordinate through shared data variables and synchronization labels. The hybrid automaton that models the entire system is then constructed from the component automata using a product operation. The definition of the product operation can be found in several papers [55,56].

HyTech can only deal with linear hybrid systems. Here we define a linear hybrid system [55]:

> A linear term φ over a set of variables V is a linear combination of the variables in V with rational coefficients. A linear formula is a boolean combination of inequalities between linear terms. A linear hybrid system is a hybrid system where invariant, initial, jump, and flow conditions are all defined by linear formula. Furthermore, the flow conditions are defined by linear expressions over $\dot{x}$ only. (i.e., the flow conditions can not depend on variables x).

The main function that HyTech performs is the verification of safety properties: given an initial region and an unsafe region, HyTech verifies whether the system starting with the initial region ends up within the unsafe region. The verification is done by forward or backward reachability analysis. The verification procedure is not necessarily decidable, i.e., the computation could go on with no guarantee of termination [54,55,56]. The reachability analysis is not necessarily decidable even for linear hybrid automata. In fact, it is not decidable except for some special cases. It is decidable for timed automata (systems that have only clocks that run with an identical rate) or simple multi-rate systems. The simple multi-rate system is a system with clocks only, and no stopwatches. The clocks can run at different rates but all invariant conditions and guards in jump conditions are of the form $x \leq k$ and the assignments are of the form $x = k$ or $x = x$. For the problems that are decidable, most of them are PSPACE-hard (see papers [53,54,55,56]).

3.4.2 Stanford Temporal Prover (STeP)

STeP provides a toolset for verifying linear-time temporal properties of reactive and real-time systems [91]. Its deductive methods include verification rules, verification diagrams, decision procedures as well as a tool for invariant generation. STeP contains a model checker that can automatically verify or disprove linear-temporal properties of finite-state systems [8]. Assertion graphs are used to simplify and summarize the models of safety formulas [9], and proofs of specifications generally can be presented using verification diagrams [8].

The basic inputs are a reactive system expressed as a transition system, and a system property to be proved, represented by a temporal logic formula. However, STeP has been extended to include verification of safety properties of real-time systems using the clock transition system computational model. Three interface components that STeP contains are the Top-level Prover, the Interactive Prover, and the Verification Diagram Editor [10].

Specification for system descriptions are given in the form of a Simple Programming Language (SPL) program. The input to STeP is an SPL program and a temporal-logic formula that represents a property to be verified. When an SPL program is loaded, a fair transition system is automatically generated. The syntax and semantics are shown in the STeP manual [13]. Also, an SPL program is compiled into a fair transition system. To describe specific types of systems, STeP provides syntax (support) for Fair Transition Systems, Modular Transition Systems, Clocked Transition Systems, and Hybrid Transition Systems (manual).

STeP contains tools for automatic generation of invariants based on analysis of transition systems for reactive and real-time systems [11,12]. Rules shown in papers by Manna et. al. [91] are used to prove hybrid system properties. The general validity of a set of first-order verification conditions is obtained by using the rules to reduce the system validity of a temporal formula. Moreover, to specify properties (specifications) of reactive systems linear-time logic is used. Proofs of temporal system specification can often be represented with verification diagrams [8].

Two techniques for automatically generating hybrid systems invariants are described in Manna et. al. [91] as follows. The first technique characterizes the set of states that is either an initial state or can be reached by either a discrete transition or a time-step transition, starting from anywhere in the state space. The second technique takes advantage of time invariance properties. Time invariance ensures that the possible effects of taking two successive transitions of duration D1 and D2 are the same as taking one transition of duration D1 + D2.

Two approaches to verifying temporal specification of reactive and concurrent systems are bottom-up or top-down. Bottom-up is referred to as forward propagation and is a symbolic forward execution of the system yielding an invariant that characterizes the set of reachable states. Top-down is referred to as backward propagation and is a symbolic backward execution of the system from the states satisfying the invariance property being proved. The result is an invariant that characterizes the states that maintain the invariant property. Unfortunately, there is no guarantee for success in forward or backward analysis.

Additionally, Bjorner et al. [8] suggests that verification of real-time systems should first involve verification that the system description is non-Zeno. If the description is Zeno, then it contains computation prefixes that can be extended to computations in which time can grow beyond any bound.

3.4.3 UPPAAL

UPPAAL is an integrated tool environment for modeling, validation and verification of real-time system modeled as networks of timed automata, extended with data types (bounded integers, arrays, etc.). UPPAAL is developed jointly by Basic Research in Computer Science at Aalborg University in Denmark and the Department of Computer Systems (DoCS) at Uppsala University in Sweden. It is appropriate for systems that can be modeled as a collection of non-deterministic processes with finite control structure and real-valued clocks, communicating through channels or shared variables [119,65]. Typical application areas include real-time controllers and communication protocols, in particular those where timing aspects are critical.

UPPAAL consists of three main parts: a description language, a simulator and a model-checker. The description language is a non-deterministic guarded command language with simple data types (e.g. bounded integers, arrays, etc.). It serves as a modeling or design language to describe system behavior as networks of automata extended with clock and data variables. The simulator is a validation tool that enables examination of possible dynamic executions of a system during early design (or modeling) stages and thus provides an inexpensive means of fault detection prior to verification by the model-checker, which covers the exhaustive dynamic behavior of the system. The model-checker is used to check invariant and reachability properties by exploring the state-space of a system, i.e. reachability analysis in terms of symbolic states represented by constraints.

The basis of the UPPAAL model is the timed automata of Alur and Dill [2,17] derived from an extension of the classic finite state automata with clock variables. To provide a more expressive model and ease the modeling task, timed automata are extended with more general types of data variables, such as Boolean and integer variables. In the current implementation of UPPAAL, a system description consists of a collection of timed automata extended with integer variables in addition to clock variables. The edges of the automata possess several types of labels: guards, actions, clock resets and assignments. A guard is a condition on the values of clock and integer variables that must be satisfied in order for the edge to be taken. A synchronization action is performed when the edge is finally taken. Clock resets and assignments are then optional means by which variables can be changed before entry into the new state. Control nodes may possess invariants, which are constraints on the clock values in order for control to remain in the node.

Formally, states of a UPPAAL model are of the form (l,v) where l is a control vector indicating the current control node for each component of the network and v is an assignment given the current value for each clock and integer variable. The UPPAAL model determines two types of transitions between states: delay transitions and action transitions. However, these two types of transitions may be overruled by the presence of urgent channels and committed locations.

The UPPAAL model checker is designed to check for simple invariant and reachability properties. A number of other properties, including bounded reachability properties, may be checked by reasoning about the system in the context of testing automata.

The two main design criteria for UPPAAL have been efficiency and ease of usage. The application of on-the-fly searching technique has been crucial to the efficiency of the UPPAAL model-checker. Another important key to efficiency is the application of a symbolic technique that reduces verification problems to that of efficient manipulation and solving of constraints [119,66,67,6]. To facilitate modeling and debugging, the UPPAAL model-checker may automatically generate a diagnostic trace that explains why a property is (or is not) satisfied by a system description. The diagnostic traces generated by the model-checker can be loaded automatically to the simulator, which may be used for visualization and investigation of the trace.

3.5 Specification Tools and Requirements Methodology

Most accidents in which software plays a part can be traced to requirements errors, not coding errors [71]. However, few techniques exist for validation of requirements. Specification Tools and Requirements Methodology (SpecTRM) is an experimental systems engineering development environment for heterogeneous safety-critical systems that includes specification and analysis tools integrated with a safety information system.

Modern high-tech systems are usually complex and made up of electromechanical, digital, and human components that must work together to achieve a system goal without creating hazardous states. SpecTRM includes a set of tools to assist in development and documentation of the complete system specification including requirements traceability and design rationale, a formal modeling language called SpecTRM-RL for those aspects that can benefit from the use of formal methods, and a set of analysis tools to assist the system engineer in detecting requirements and specification flaws and omissions.

The formal modeling language, SpecTRM-RL (SpecTRM Requirements Language), can be used to specify the blackbox functional requirements for the system components, including the software, hardware, and human tasks. To validate system design and component requirements, various analysis tools can be applied to the system specification. Current requirements analysis tools or those in development include completeness and consistency analysis, software deviation analysis (robustness of the software when operating in an imperfect environment), execution and animation of SpecTRM-RL models, system and software hazard analysis, analysis of the potential for inducing human errors such as mode confusion, test data generation and requirements coverage analysis, and operator task analysis.

SpecTRM-RL is a formal modeling language that is both executable and analyzable. At the same time, it is easy enough to read by engineers and programmers that it can serve as the system specification. A previous language based on similar concepts, named RSML, is being used for the official system specification of TCAS II, a collision avoidance system required on most commercial aircraft that fly in U.S. airspace [76]. SpecTRM-RL builds on what was learned while designing and using RSML in order to make the language even more readable and reviewable, to eliminate error-prone features (such as internally broadcast events), and to provide more guidance in building models [75].

SpecTRM Requirements Language (SpecTRM-RL) acts as a formal specification language that overlays the low level requirements state machine (RSM) that forms the basis for the language. The RSM is based on a simple Mealy automaton with outputs on the transitions between states. The RSM is very low level and is not appropriate as a modeling language for complex systems. As long as the mapping between SpecTRM-RL to the RSM is unambiguous and well-defined, formal analysis is possible on both the underlying RSM formal model as well as the higher-level SpecTRM-RL specification itself [97].

The notation has been designed to be practical for specifying very large and complex systems and to enhance readability and reviewability. Each component of the system is specified using SpecTRM-RL models. The models are blackbox in that only externally visible behavior is specified–no internal (implementation) design is included. The system specification is the composition of the blackbox component models. A slightly different notation is used for specifying human procedures to make them more easily understood by human factors experts [72]; the underlying formal model is the same, however, so they can be executed and analyzed in conjunction with the other system component models.

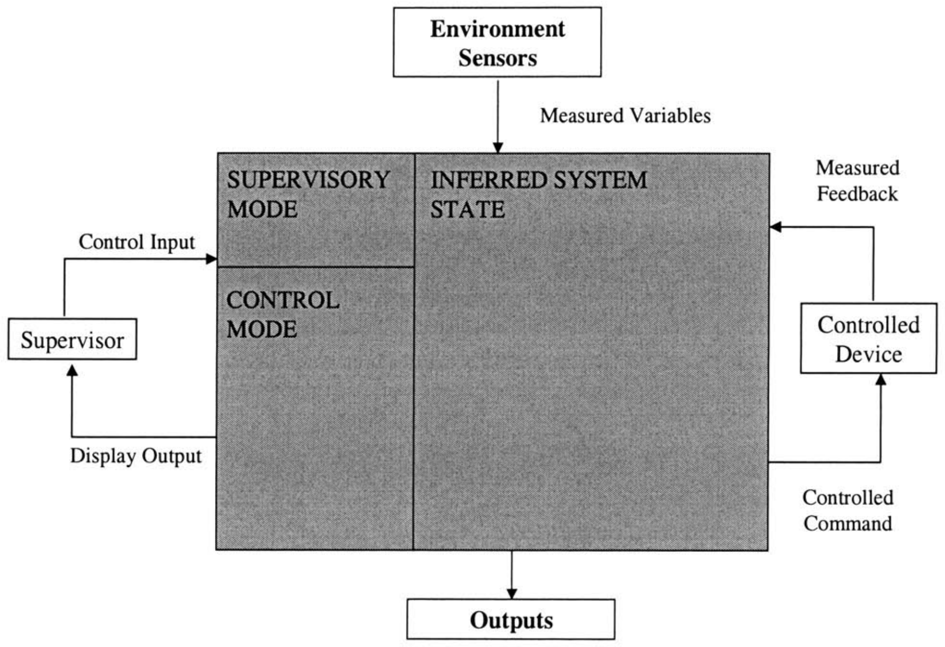


Figure 3.1: Form of a SpecTRM-RL Model

A SpecTRM-RL model has three components: (1) a specification of the supervisory interface to the component, (2) a specification of the control modes for the component, and (3) a model of the controlled process or plant including relevant operating modes, state variables, and interface variables.

State Variable

Altitude

Obsolescence: 2 sec
Timing Behaviour:
Exception Handling: Because the Altitude Status Signal changes to obsolete after 2 seconds, Altitude will change to unknown if all input signals are lost for two seconds
Description: The altitude variable is used by the altitude switch to indicate whether the threshold has been reached.
Comments:
References:
Appears in: DOI_Power_On

DEFINITION

= Unknown **IF**

Powerup	T	*	*
Controls.Reset	*	T	*
Analog-Alt=Unknown	*	*	T
Dig-Alt1=Unknown	*	*	T
Dig-Alt2=Unknown	*	*	T

Figure 3.2: Sample And/Or Table from the Altitude Switch Specification

The events and conditions causing transitions between states are described in AND/OR tables. The behaviour in real systems is too complex to write on arrows between circles. Instead, we use a tabular representation in disjunctive normal form of a predicate logic statement over the various states, variables, and modes in the specification. The far left column of an AND/OR table lists the logical phrases of the predicate. Each of the columns is a conjunction of those phrases and contains the logical values of the expressions.

If one of the columns evaluates to true, then the entire table evaluates to true and the transition is enabled. A column evaluates to true if all of its elements match the truth-values of the associated predicates (a dot indicates "don't care"). Timing statements are allowed as well as references to previous values of state variables.

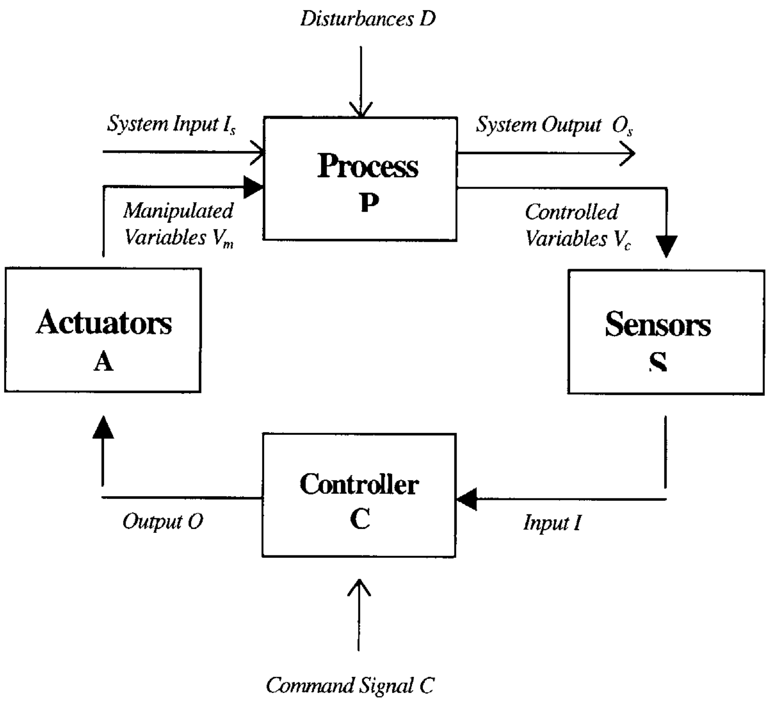


Figure 3.3: Process Control Loop

A multivariable controlled process can, in its most generalized sense, be modeled as above (Fig 3.3). The process, here denoted by the plant P(s), has the controlled input, I_s, and the disturbance input D, along with the controlled variables V_c and the system output O_x. The sensor (S) output I is fed into the controller, which is modeled by C(s). The controller then creates the control signal O in order to actuate (A) the correct behavior in the plant.

The Requirements State Machine (RSM) is defined as a seven-tuple $(\Sigma, Q, q_0, P_t, P_o, \gamma, \delta)$, with respect to the control loop above where [59]:

1. S is the set of input and output variables, I and O to the controller.

2. Q is the finite set of states of the controller C and q_0 is the initial state of C.

3. P_t is the set of Boolean functions over S; they represent predicates on the values and timing of the inputs I from the sensors. These predicates are called trigger predicates because they trigger a state change in the RSM.

4. P_0 is the set of Boolean functions over S; they represent predicates on the outputs O of the controller.

5. γ is the trigger-to-output relationship mapping from $Q \times P_t$ to P_0. That is, $\gamma(q, p)$, where $q \in Q$ and $p \in P_t$ gives the predicate describing the output O to the actuators to be generated when the transition with input predicate p is taken out of state q.

6. δ is the state transition function mapping $Q \times P_t$ to Q. That is, $\delta(q, p)$ where $q \in Q$ and $p \in P_t$ defines the next state when the system is in state q and takes the transition having p as the input predicate.

In addition, the RSM has the following properties [59]:

1. Predicates in P_t and P_0 are expressed using the standard Boolean operators and ordinary arithmetic operators. The expression $X \rceil$ represents an input or output occurrence of X. This expression evaluates to true the moment input X arrives at the black-box boundary or output X is produced and presented at the black-box boundary. The value of a variable X is denoted by val(X).

2. When an input I arrives at the black box boundary, it is denoted as I_j or simply I. The previous occurrence of the same input is denoted I_{j-1} and so forth. The ordering of outputs is expressed in the same manner. The first variable I arriving at the black box boundary is referred to as I_1, not I_0.

3. A clock and a function giving the absolute time of an event are needed to express timing. The expression $t(I \rceil)$ denotes the time when I arrives at the black-box boundary. The clock is started when the system receives the signal to startup.

3.5.1 Extending SpecTRM-RL to Hybrid Systems

For many hazard analysis problems a discrete model suffices. However, for some analysis problems, a hybrid model is preferable. Adapting this formalism into a language capable of representing a hybrid model, that is, a model containing both discrete and continuous components, is relatively simple. The only element missing in the present formulation of SpecTRM-RL is the ability to allow the state of the system to evolve over time, without any discrete actions occurring.

Thus, in order to adapt the present formalism of SpecTRM-RL, there only remains to be added the final qualification of a set of time-passage steps:

$$q \xrightarrow{\Delta t} q' \tag{3.19}$$

asserting that "from state q the system can move to state q' during a positive amount of time Δt in which no trigger predicate is received and thus no discrete action occurs". The key characteristic of the continuous state evolution under time is the specific interpretation of the individual time steps. The system must always satisfy the following two properties. First, if time can advance by a particular amount $\Delta t = \frac{1}{2}\Delta t + \frac{1}{2}\Delta t$ in two distinct steps (with no intervening discrete actions), it can also advance by Δt in a single step. Secondly, if time can advance by Δt in one step from state q to state q', then there exists a trajectory assignment that maps all times in the interval $\Delta t = [t_{init}, t_{fin}]$ to automaton states in a consistent manner in order to explain the evolution of the system from state q to state q'. For hybrid systems, these trajectories usually have physical significance. They often describe physical parameters that evolve in a continuous fashion with respect to time, such as position, velocity, acceleration, flow of information, etc. In such cases, the trajectories, denoted by τ, are descriptions of continuous functions of time. However, no such assumption is made in this formulation, and the trajectories are not required to be continuous. If we regard the time argument as ranging over the interval $t \in \{0\} \cup \{R^+\} \cup \infty$, that is, the positive real numbers, along with zero and infinity, we can define a trajectory formally over a left-closed interval $I \in [t_{init}, t_{fin})$ on the range of t such that:

$$\tau(t_1) \xrightarrow{t_2 - t_1} \tau(t_2) \quad \text{for all } t_2, t_1 \in I \text{ such that } t_1 < t_2 \tag{3.20}$$

Note that with this definition, it is required that the latest time of the open-ended interval $I \in [t_{init}, t_{fin})$ be the supremum of *I*. If *I* is an infinite interval, then the latest time is regarded as being infinity. So the trajectory over the interval is such that $q = \tau(t_{init})$ and

$q' = \tau(\sup(t_{fin}))$. Thus, the trajectory assigns a state to each time in the interval I in a consistent manner.

The two mathematical properties of hybrid traces can be specified as the following axioms [83]:

Axiom 1:

$$\text{If } q \xrightarrow{\Delta t_1} q' \text{ and } q' \xrightarrow{\Delta t_2} q'' \text{ then } q \xrightarrow{\Delta t_1 + \Delta t_2} q'' \tag{3.21}$$

Axiom 2:

Over the interval $\Delta t = [t_{init}, t_{fin})$, in which no discrete action occurs, there exists the trajectory such that

$$\tau(t_{init}) = q \xrightarrow{\Delta t} q' = \tau(\sup(t_{fin})) \tag{3.22}$$

Essentially, Axiom 1 allows for the combination of time intervals that possess no intervening discrete actions, and Axiom 2 allows for the assignment of states to intervening instances in time in the aforesaid intervals. The case of the zero-time interval (with no intervening discrete transition) can be considered to be the case such that:

$$q \xrightarrow{0} q' \quad \Leftrightarrow \quad q = q' \tag{3.23}$$

With the notion of trajectories defined, in order to map the evolution of the continuous states during time passage in the SpecTRM-RL framework, it now becomes possible to specify hybrid systems using SpecTRM-RL.

The general form of a SpecTRM-RL model must be augmented to include an additional section which captures the continuous nature of the evolving hybrid trajectory of the system. A new heading, entitled Inferred System Trajectory, is added to the generic SpecTRM-RL model form (see Fig. 3.4 on next page):

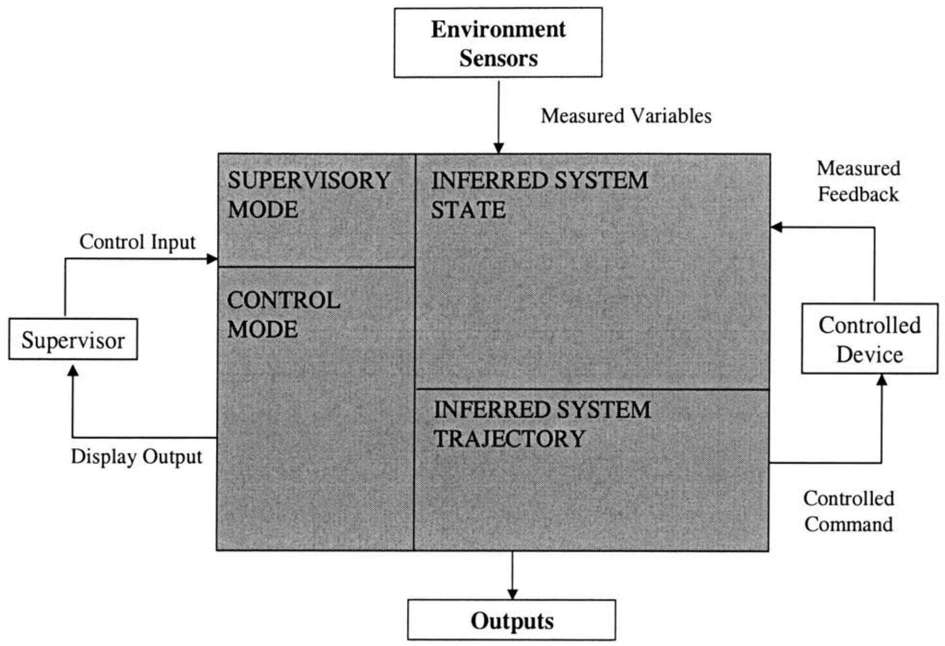


Figure 3.4: Form of a Hybrid SpecTRM-RL Model

The Inferred System Trajectory consists of trajectory elements. These trajectory elements can be comprised of combinations of input, output, interface or state variables. Taken together as a whole, the trajectory elements reflect the continuous trajectory of the system over time. The same sorts of logical predicates that can be formed with state variables can be formed with trajectory elements (Conjunction, Intersection, Negation etc.). The trajectory elements have a far reaching impact on the system's reachability. No longer is the reachability of the system quantized into discrete states, it is mapped onto regions or reachable space. These issues are further examined in the next chapter.

CHAPTER 4

I could be bounded in a nutshell and count myself a king of infinite space...

William Shakespeare, Hamlet, II.ii.270-273

Reachability

The notion of reachability spans several disciplines, such as automata theory, operations research, control theory, and mobile computing, to name a few. While the general concept of reachability is intuitively the same in most topics, a rigorous study of the idea yields some surprising results. Techniques for identifying reachable states in some disciplines (i.e. control theory) are mathematically formulated and relatively easy to implement, while in others (i.e. automata theory) the entire state space may need to be generated. The ability to convert finite automaton models to state space models allows for reachable states to be more easily identified. This helps to curtail the state explosion problem encountered in most reachability analyses.

4.1 Finite Automaton Models

For a deterministic finite automaton, $M = (Q, \Sigma, \delta, q_0, F)$ as introduced in Section 3.1.1, the reachability of a state, can be formally defined as:

> A state q_j in a DFA is reachable from the state q_i if there is a sequence of inputs $\sigma = \sigma_1\sigma_2\ldots\sigma_k$ that enables the automaton to transition from q_i to q_j through a valid set of states.

Thus, in order to find out whether or not one state is actually reachable from another, a search must be performed through the reachability graph of the automaton. One classification of search techniques is forward or backward [71]. A forward, or inductive, search takes an initiating state and traces it forward in time. The result is a set of states or conditions that represent the effects of the initial event. Tracing a state forward can generate a large number of states, and the problem of identifying a particular reachable state from an initial state may be unsolvable using a reasonable set of resources. For this reason, forward analysis is often limited to only a small set of initial states. In a backwards, or deductive, search, the analyst starts with a final event or state and identifies the preceding events or states. However, the backwards reachability graph that is generated may be even larger than the forwards reachability graph.

Another classification of search technique is depth-first and breadth-first. Depth-first searches take an initial state, and then pursue one successor-path to its completion, and checks whether or not the path has encompassed the desired state. It is obvious that unless you are very lucky, and the desired state is on the first path pursued, a great many unneeded paths are generated, which is computationally intensive. Breadth-first searches generate all successor states to the initial states, check to see if the desired state is within that set, and then generate all the successor states to the previous set, in order to check again. As before, it can be very computationally intensive to generate a growing number of successor states, and this approach rapidly leads to state explosion.

Thus, it can become very computationally intensive in order to determine whether or not a particular state is reachable within a DFA, or to even generate the set of reachable states.

4.2 Markov Models

A discrete Markov process or chain can literally be thought of as a finite automaton with a probabilistic description of its inputs. A Markov process is a stochastic system for which the occurrence of a future state depends only upon the immediately proceeding state. Thus, if $t_0 < t_1 < \ldots < t_n$ represent points in time, and ξ_{t_k} is the random variable which characterizes the state of the system at t_k, then the probability that the system is in the state x_n at t_n given that it was in x_{n-1} at t_{n-1} is given by: $p_{x(n-1),xn} = P\{\xi_{tn} = x_n|(\xi_{t(n-1)} = x_{n-1})\}$. This is called the *transition* probability, or *one step transition* probability. Given these one-step transition probabilities for each state of the system, it is possible to construct a transition matrix **P** where p_{ij} is the one step transition probability from state x_i to state x_j. This is a pivotal notion, in that it introduces the idea that given a state x_i, all of the possible successor (or predecessor) states can be identified using a simple matrix operation.

Final states of a finite automaton are represented as absorbing states of the Markov chain, that is, $p_{i,i}=1$. Initial states are represented in the vector $\mathbf{a^0}=\{a_1^0, a_2^0, ..., a_n^0\}$ of initial probabilities associated with the likelihood of starting in any given state x_i. Thus, the transition matrix **P** together with the initial probabilities $\mathbf{a^0}=\{a_1^0, a_2^0, ..., a_n^0\}$ associated with the states $X=\{x_1, x_2, ..., x_n\}$ completely define a Markov chain. The notion of reachability for a Markov chain can be defined as:

> A state x_j in a Markov chain is reachable from a state x_i if it is possible to go from x_i to x_j in a finite number of transitions.

The concept of Markov chains and reachability acts to bridge the gap between finite automata and the next topic, state space systems.

4.3 Discrete Time State Space Systems

A detailed understanding of how inputs impact the states of a given system can be termed as a discussion of the *reachability* of the system. Consider an n-th order discrete time system, as defined in Section 3.2:

$$x(i+1) = Ax(i) + Bu(i) \tag{4.1}$$

Now, for an arbitrary initial condition $x(0)$, in k steps the system will be in the state:

$$x(k) = A^k x(0) + \sum_{i=0}^{k-1} A^{k-i-1} B u(i) \tag{4.2}$$

or more explicitly:

$$x(k) = A^k x(0) + [A^{k-1}B|A^{k-2}B|\dots|B]\begin{bmatrix} u(0) \\ u(1) \\ \dots \\ u(k-1) \end{bmatrix} \tag{4.3}$$

$$x(k) = A^k x(0) + \Re_k U_k \tag{4.4}$$

where $\Re_k$ and U_k are defined from inspection of (4.3) and (4.4). Now, consider whether one may choose the input sequence *u*(*i*), $i \in [0, k-1]$, so as to move the system from *x*(0) to a desired target state *x*(*k*)=*d* at a given time *k*. If there is such an input, it can be said that the state *d* is reachable in *k* steps. Now, assuming there are no constraints placed on the input, the set of reachable states from the origin in exactly *k* steps is precisely the range of the matrix $\Re_k$. The *k*-reachable set is therefore a subspace, and the matrix $\Re_k$ is called the *k*-step *reachability matrix.*

Theorem 4.1:

For $k \le n \le l$,

$$Range(\Re_k) \subseteq Range(\Re_n) = Range(\Re_l) \tag{4.5}$$

so the set of states reachable from the origin in some finite number of steps by appropriate choice of control input is precisely the subspace of states reachable in *n* steps.

Proof:

The fact that $Range(\Re_k) \subseteq Range(\Re_n)$ for $k \le n$ follows trivially from the fact that the columns of $\Re_k$ are included among those of $\Re_n$. To show that $Range(\Re_n) = Range(\Re_l)$ for $l \ge n$, the Cayley-Hamilton theorem says that A^i for $i \ge n$ can be written as a linear combination of $A^{n-1}, A^{n-2}, \dots, A, I$, so that all the columns of $\Re_l$ for $l \ge n$ are linear combinations of the columns of $\Re_n$. Thus, the result $Range(\Re_k) \subseteq Range(\Re_n) = Range(\Re_l)$ follows directly.

□

In view of Theorem 4.1, the subspace of states reachable in n steps $Range(\Re_n)$ is referred to as the principle reachable subspace. Hence, any reachable target state exists in $Range(\Re_n)$, and is reachable in n steps or less. The matrix

$$\Re_n = [A^{n-1}B|A^{n-2}B|\ldots|B] \tag{4.6}$$

is termed the *reachability matrix*. If the entire n x n space is reachable, then the condition:

$$Rank(\Re_n) = n \tag{4.7}$$

that is, the columns of $\Re_n$ are linearly independent, and span the entire space.

Note that (4.4) shows that getting from a non-zero starting state $x(0)=s$ to a target state $x(k)=d$ requires that there be a U_k such that:

$$d - A^k s = \Re_k U_k \tag{4.8}$$

For arbitrary d and s, the requisite condition is the same as that for reachability from the origin. Thus, we can get from an arbitrary initial state to an arbitrary final state if and only if the system is reachable from the origin, and we can make the transition in n steps or less, when the transition is possible.

Now, it follows from the previous analysis that the reachable subspace $Range(\Re_n)$ is A-*invariant*. That is, if

$$x \in Range(\Re_n) \Rightarrow Ax \in Range(\Re_n) \tag{4.9}$$

This can easily be seen from the fact that

$$A\Re_n = [A^n B|A^{n-1}B|\ldots|AB] \tag{4.10}$$

where the last n-1 blocks are present in $\Re_n$, and the Cayley-Hamilton theorem allows $A^n B$ to be expressed as a linear combination of blocks in $\Re_n$. This establishes the fact that

$$Range(A\Re_n) \subset Range(\Re_n) \tag{4.11}$$

It follows directly that

$$x = \Re_n\alpha \Rightarrow Ax = A\Re_n\alpha = \Re_n\beta \in Range(\Re_n) \tag{4.12}$$

It is generally true that any *A-invariant* subspace is the span of some eigenvectors and generalized eigenvectors of A. It turns out that $Range(\Re_n)$ is the smallest A-invariant subspace that contains $Range(B)$, but this shall not be proven or pursued. The upshot of this result is that an interpretation of the reachable space of the system can be realized using the eigenvectors or generalized eigenvectors of the matrix A. Thus, the system represented by equation (4.1) may be thought of as having a collection of "Jordan chains" or generalized eigenvectors at its core. Reachability, which was first introduced in terms of reaching target states, turns out also to describe the ability of the system to independently "excite" or drive the Jordan chains. This is the implication of the reachable subspace being an A-invariant subspace. The critical issue for achieving reachability of a particular chain, is to be able to excite the beginning of the chain; this excitation can then propagate down the chain. An additional condition is needed if several chains have the same eigenvalue; in this case, we need to be able to *independently* excite the beginning of each of these chains. With distinct eigenvalues, we do not need to impose this independence condition; the distinctness of the eigenvalues permits independent motions.

4.4 Continuous Time State Space Systems

The definition of continuous time reachability is identical to that of discrete time reachability. However, while in the discrete time case reachability can be checked through simple matrix conditions, it is not so clear that one can derive simple matrix conditions for continuous time systems. It is somewhat surprising to find that the reachability condition for continuous systems is the same as for discrete systems.

Consider now the n-th order continuous time model:

$$\dot{x}(t) = Ax(t) + Bx(t) \tag{4.13}$$

Consider whether one can choose the input $u(t), t \in [0, L]$, so as to move the system from $x(0)=0$ to a desired target state $x(L)=d$ at a given time $L>0$. If there is such an input, we say that the state d is reachable in time L. It can be shown that the choice of L is not critical, similar to the discrete time scenario. The relationship of $x(L)$ to $u(t)$ under the above conditions is given by:

$$x(L) = \int_0^L [e^{(L-t)A}]Bu(t)dt \tag{4.14}$$

$$x(L) = \int_0^L F^T(t)u(t)dt \tag{4.15}$$

$$x(L) = \langle F(t), u(t)\rangle_L \tag{4.16}$$

where $F^T(t) = [e^{(L-t)A}]B$ and the Gram product of $F(t)$ and $u(t)$ is defined in (4.15). The set $\Re$ of reachable states forms a subspace, because:

$$\left.\begin{aligned} x_a(L) &= \langle F(t), u_a(t)\rangle_L \\ x_b(L) &= \langle F(t), u_b(t)\rangle_L \end{aligned}\right\} \Rightarrow \alpha x_a(L) + \beta x_b(L) = \langle F(t), \alpha u_a(t) + \beta x_b(t)\rangle_L \tag{4.17}$$

that is, any linear combination of reachable states is reachable. This assumes, of course, that there are no constraints placed on $u(t)$. Thus, $\Re$ is referred to as the reachable subspace of the system. Strictly speaking, $\Re$ is the reachable space for target states at time L, but the choice of L, it will soon be shown, turns out to be irrelevant.

Now, let us define the *reachability Gramian* (at time L) of the system as:

$$P_L = \langle F(t), F(t)\rangle_L \tag{4.18}$$

Theorem 4.2

The reachable subspace $\Re$ is related to the *reachability Gramian* (at time *L*) P_L as follows:

$$\Re = Range(P_L) \tag{4.19}$$

$$\Re = Range\left(\int_0^L F^T(t)F(t)dt\right) \tag{4.20}$$

Proof:

To first show that

$$\Re \subset Range(P_L) \tag{4.21}$$

it is equivalent to show that:

$$Range^{\perp}(P_L) \subset \Re^{\perp} \tag{4.22}$$

For this, note that:

$$\begin{aligned} q^T P_L = 0 \Rightarrow q^T P_L q &= 0 \\ \Leftrightarrow \langle F(t)q, F(t)q\rangle &= 0 \\ \Leftrightarrow q^T F^T(t) &= 0 \\ \Rightarrow q^T x(L) &= 0 \end{aligned} \tag{4.23}$$

where the last implication comes from equations (4.2-4). So, any vector in $Range^{\perp}(P_L)$ is also in $\Re^{\perp}$. Now, it can be shown that $\Re = Range(P_L)$ by showing that any target state $d \in Range(P_L)$ is also in $\Re^{\perp}$. Suppose $d = P_L\alpha$, and pick $u(t) = F(t)\alpha$. Then:

$$\begin{aligned} x(L) &= \int_0^L F^T(t)F(t)\alpha dt \\ &= P_L\alpha \\ &= d \end{aligned} \tag{4.24}$$

□

From here, a further conclusion about the state space system can be drawn:

Theorem 4.3

$$\begin{aligned} Range(P_L) &= Range([A^{n-1}B|A^{n-2}B|\ldots|B]) \\ &= \Re \end{aligned} \tag{4.25}$$

Proof:

This can be proved by showing that the orthogonal complements of the above two subspaces are equal.

$$q^T P_L = 0 \Rightarrow q^T e^{A(L-t)} B = 0 \tag{4.26}$$

as in the proof of Theorem 4.1. Then:

$$q^T e^{A(L-t)} B = 0 \Rightarrow \begin{cases} q^T B = 0 & \text{set } t = L \\ q^T AB = 0 & \text{differentiate and set } t = L \\ \ldots \\ q^T A^{n-1} B = 0 & \text{differentiate n times and set } t = L \end{cases} \tag{4.27}$$

$$\Leftrightarrow q^T \Re_n$$

Conversely:

$$q^T \Re_n = 0 \Rightarrow q^T e^{A(L-t)} B = 0 \tag{4.28}$$

since by Cayley-Hamilton, $e^{A(L-t)}$ can be written as time varying combinations of $A^{n-1}, A^{n-2}, \ldots, A, I$. This leads directly to:

$$q^T \Re_n = 0 \Rightarrow q^T e^{A(L-t)} B = 0 \Rightarrow q^T P_L = 0 \tag{4.29}$$

□

Note that from Theorem 4.2 and the fact that $\Re_n$ does not depend on *L*, the reachable subspace is independent of the choice of *L*. However, the characteristics of the control input used to attain a particular target state will depend on *L*; the smaller *L* is, the "larger" *u*(*t*) is expected to be, in some sense.

Thus, it has been shown that the overall conditions for reachability of both continuous and discrete time systems are the same. Hence, any reachability analysis applied to discrete models can also quite easily be extended to continuous or hybrid models.

4.5 Converting DFA models into State Space Models

The heart of analyzing a system from a safety perspective is identifying and analyzing the system for hazards, which are states or conditions of the system that combined with some environmental conditions can lead to an accident or loss event. Once hazards are identified, steps can be taken to eliminate them, reduce the likelihood of their occurring, or mitigate their effects on the system [71]. A hazard analysis requires some type of model of the system, which may be an informal model in the mind of the analyst, a written informal or formatted specification of the system, or a formal mathematical model. Different models allow for different types of analyses and for additional rigor and completeness in the analysis. The specification model can also be analyzed with respect to specific known hazards. Different model types facilitate different types of analyses. From the previous sections, it is observed that substantial techniques exist for determining the reachability of states when a model is expressed in a state space formulation. The goal of this section is to see how a deterministic finite automata can be converted into a state space formulation in order to determine reachability properties. Coupled with the notion of abstraction, the computational intensity of calculating the reachability of a given state becomes greatly reduced.

4.5.1 Motivation for System Reformulation

A state is said to be reachable from another state if there is a path from the first state to the second. In most systems, all desired states must be reachable from the initial state. If a state is unreachable, there are two possibilities:

1. The state has no function in the system and can be eliminated from the model
2. The state should be reachable and the model is incorrect and should be modified accordingly.

Theoretically, if the entire state space of a model were to be generated, it would be possible to identify all hazardous states, and all the paths involving these aforesaid states. Doing so would involve a forward search, as described in Section 4.1, which would have the drawbacks mentioned. Utilizing a backwards search from a hazardous state to see whether or not the initial state can be reached can also fall prey to similar difficulties. The number of backward paths for hazardous states is still enormous for real systems. In complex systems, complete reachability analysis is often impractical, but Leveson and Stolzy have shown that it is possible to devise algorithms that reduce the necessary state space search by focusing on a few properties. Their solution is to start from the hazardous state and only work far enough backwards along the path to determine how to change the model to make the hazardous state unreachable. Thus, only a subset of the actual reachablility set will need to be generated [73]. Hence, if the state machine model of a system with *n* states were reformulated as a state space model, determining the reduced reachability set would only involve at most *n* matrix multiplications of an *n* x *n* matrix.

4.5.2 A Simple Example: The Gambler's Ruin

Consider first a very simple discrete example upon which to apply the previous theory. The classical problem of the Gambler's Ruin involves gambling against a bank with capital A_1 in the following fashion:

> A coin is flipped, and if the outcome is heads, the bank pays one dollar to the player, but if the outcome is tails, the player pays one dollar to the bank.Consider the probability of flipping a head as p, and that the player has a capital A_2. The game terminates if either the bank or the player loses all of their capital.

The problem becomes:

Is it possible to break the bank?

This problem can be modeled as a simple deterministic finite automata (DFA), where there are two accepting states: when the capital is zero for either the player or the bank. Consider flipping a head to be an input of 0, and a tail to be an input of 1. The automaton becomes (Fig 4.1):

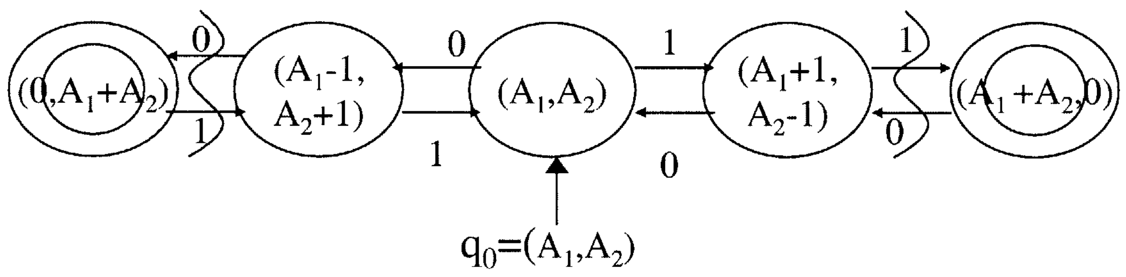


Figure 4.1: Finite Automaton of the Gambler's Ruin Problem

It is obvious from this automaton that the absorbing state of breaking the bank is reachable. However, there are A_1+A_2+1 states in the entire automaton. By unwinding the automaton and reformulating it into a state space notation, and then applying the concept of abstraction, the number of states can be greatly reduced, and computational resources can be saved.

In order to create a state space model of the automaton, first let us consider reframing the DFA as a Markov chain. For generality's sake, consider the probability of flipping a head as being p, and the probability of flipping a tail as being (1-p). The final states of breaking the bank become absorbing states, and the initial state becomes the only state in the initial state vector, with a probability of one. A similar diagram of the Markov chain becomes:

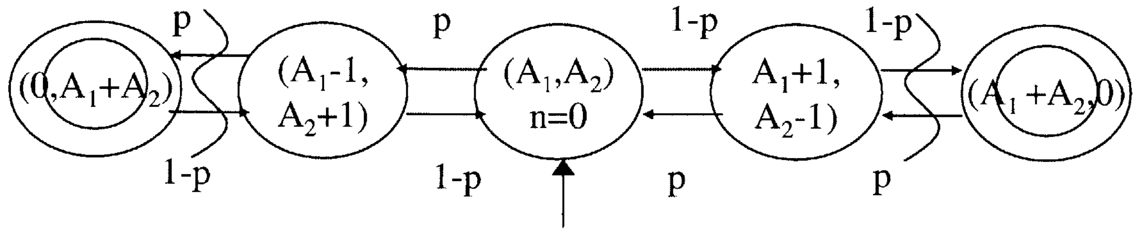


Figure 4.2: Markov Chain of the Gambler's Ruin Problem

From elementary mathematics, it is seen that the probability of breaking the bank is:

$$P(\text{Breaking Bank}) = \frac{\left(\frac{1-p}{p}\right)^{A_2} - 1}{\left(\frac{1-p}{p}\right)^{A_1+A_2} - 1}, \qquad p \neq 1-p \tag{4.30}$$

$$P(\text{Breaking Bank}) = \frac{A_1}{A_1 + A_2}, \qquad p = 1-p \tag{4.31}$$

Again, it is obvious that it is theoretically possible to break the bank, and thus the final state is reachable. The Markov chain also has A_1+A_2+1 states. It seems that little progress has been made, but in reality, a fundamental step has been taken. For, in the creation of the Markov chain, the one-step transition matrix **P** has been formulated. If we consider x_1 to be the absorbing state of breaking the bank (i.e. the bank has zero dollars and the player has A_1+A_2 dollars), and $x_{A_1+A_2+1}$ to be the absorbing state of the player going broke, filling in the requisite number of states in-between, while taking the probability of flipping a head as being one-half, the transition matrix **P** would look something like this:

$$\begin{bmatrix} 1 & \frac{1}{2} & 0 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & \frac{1}{2} & 0 & 0 & \cdots & 0 & 0 \\ 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 & \cdots & 0 & 0 \\ 0 & 0 & \frac{1}{2} & 0 & \frac{1}{2} & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & \cdots & \frac{1}{2} & 0 \\ 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \cdots & \frac{1}{2} & 1 \end{bmatrix}$$

Figure 4.3: Transition Matrix **P**

In reality, what we have arrived at is the state transition matrix A in the state space formulation of the problem. Now, formulating the state space equivalent model (as described in Section 3.2) to the finite automaton becomes trivial. Consider A_1+A_2+1 state variables x_i at the kth coin flip. The state variable $x_i(k)$ represents the likelihood of the player having i-1 dollars at the kth flip. The discrete time state space formulation now looks like:

$$\begin{bmatrix} x_1(k+1) \\ x_2(k+1) \\ x_3(k+1) \\ x_4(k+1) \\ \vdots \\ x_{A_1+A_2-1}(k+1) \\ x_{A_1+A_2}(k+1) \\ x_{A_1+A_2+1}(k+1) \end{bmatrix} = \begin{bmatrix} 1 & \frac{1}{2} & 0 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & \frac{1}{2} & 0 & 0 & \cdots & 0 & 0 \\ 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 & \cdots & 0 & 0 \\ 0 & 0 & \frac{1}{2} & 0 & \frac{1}{2} & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & \cdots & \frac{1}{2} & 0 \\ 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \cdots & \frac{1}{2} & 1 \end{bmatrix} \begin{bmatrix} x_1(k) \\ x_2(k) \\ x_3(k) \\ x_4(k) \\ \vdots \\ x_{A_1+A_2-1}(k) \\ x_{A_1+A_2}(k) \\ x_{A_1+A_2+1}(k) \end{bmatrix}$$

Figure 4.4: State Space Formulation of the Gambler's Ruin Problem

Now, this system is still very large, possessing A_1+A_2+1 state variables, so that calculating the range space condition would seem to be very computationally intensive. Thus, to avoid this, we must now use the concept of abstraction, in order to reduce the number of state variables.

4.5.3 Abstraction

In order to formulate the Gambler's ruin problem in state space format, one might begin by creating A_1+A_2+1 state variables in order to mimic the states in the finite automaton. However, it is much cleverer to consider approaching the problem more abstractly. Instead of considering each combination of (Banker's_Capital, Player's_Capital) in Fig 4.1 as a distinct state, and writing out the entire state space, let us view the problem in terms of winning and losing. Each time the player flips a head, s/he wins a dollar, and each time s/ he flips a tail, a dollar is lost. If we return to the notion that the probability of flipping a head is *p*, and that of flipping a tail is (1-*p*), and let $u(i)$ represent the probability of the player breaking the bank with *i* dollars, then, for $1 < i < A_1 + A_2 - 1$:

$$u(i) = pu(i+1) + (1-p)u(i-1) \tag{4.32}$$

isolating for $u(i+1)$ gives

$$u(i+1) = \frac{1}{p}u(i) - \frac{1-p}{p}u(i-1) \tag{4.33}$$

which implies

$$u(i+2) = \frac{1}{p}u(i+1) - \frac{1-p}{p}u(i) \tag{4.34}$$

Note that the boundary conditions are expressed as:

$$u(0) \equiv 0 \tag{4.35}$$

$$u(A_1 + A_2) = 1 \tag{4.36}$$

that is, if the player has zero dollars, s/he has lost, and if the player has A_1+A_2 dollars, s/he has won.

Then let $x_1(i) = u(i+1), x_2(i) = u(i)$ be the state variables in order to get:

$$\begin{bmatrix} x_1(i+1) \\ x_2(i+1) \end{bmatrix} = \begin{bmatrix} \frac{1}{p} & \frac{-1(1-p)}{p} \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x_1(i) \\ x_2(i) \end{bmatrix} \tag{4.37}$$

or

$$\vec{x}(i+1) = A\vec{x}(i) \tag{4.38}$$

which is in standard state space format. By inspection, it can be seen that the matrix A is full rank (for $p \neq 1$), as is the matrix A^2. Now, applying the reachability conditions of equation (4.7) we determine that the entire state space is reachable. Hence, we can conclude that the state of the player breaking the bank is reachable. This is accomplished by doing a very simple matrix multiplication on a very small matrix. Hence, the act of determining reachability becomes very reasonable.

As an aside, using the boundary conditions in equations (4.35-6) we can solve for the probability the player breaks the bank given an initial capital of A_2 dollars:

$$\vec{x}(i) = A^i x(0), \qquad \vec{x}(0) = \begin{bmatrix} pu(2) \\ 0 \end{bmatrix} \tag{4.39}$$

so with initial player capital of A_2 dollars,

$$\vec{x}(A_2) = A^{A_2} \begin{bmatrix} \frac{1}{p}u(2) \\ 0 \end{bmatrix} \tag{4.40}$$

where the probability of the player breaking the bank is given by $x_2(A_2)$.

So, it is easily observed that the probability of breaking the bank is not zero, and thus it is confirmed that this final state is reachable. Note that this approach is identical to formulating the problem using A_1+A_2+1 state variables.

From the perspective of the bank, the state in which the player breaks the bank is hazardous. Thus, using the technique of Leveson and Stolzy [73] outlined in Section 4.5.1, the bank would begin the analysis from that hazard, and attempt to find the first state from which this disaster could be averted, which would occur in the state where the bank has one dollar, and the player has A_1+A_2-1 dollars. The issue of backwards reachability becomes simple, because from our formulation we are carrying the predecessor state continuously with us in the state vector $\vec{x}$. Thus, in order to determine what the reachable predecessor states were, we need only look to the second element of the state vector. Hence, given this state (or family of states), we can determine their usefulness in escaping the hazard. It is obvious that the predecessor state $(1,A_1+A_2-1)$ is important, because it leads directly to the hazardous state and to another non-hazardous state $(2, A_1+A_2-1)$. Hence, guarding conditions must be put around the critical state of $(1, A_1+A_2+1)$ by the bank in order to limit the potentially hazardous situation. In reality, of course, the bank would determine a margin of safety, say \$100, and declare that to be the hazardous state, as no bank wishes to have its capital diminish to \$1. So a family of hazardous states would be formed (all those states in which the bank has capital less than \$100), leading to the creation of a safety envelope for the problem.

This technique of generating backwards states from the hazardous state in order to determine the first escape path is developed using the Leveson-Stolzy algorithm [73], which was originally designed to work on Petri Nets. A Hazard Automaton Reduction Algorithm is developed from the Leveson-Stolzy algorithm by implementing modifications which allow the concepts of Leveson-Stolzy to be efficiently applied to automata and state machines. The formal definition of the Hazard Automaton Reduction Algorithm along with several of its properties and their proofs are detailed in depth in the following chapter.

CHAPTER 5

When asked what it was like to set about proving something, the mathematician likened proving a theorem to seeing the peak of a mountain and trying to climb to the top. One establishes a base camp and begins scaling the mountain's sheer face, encountering obstacles at every turn, often retracing one's steps and struggling every foot of the journey. Finally when the top is reached, one stands examining the peak, taking in the view of the surrounding countryside and then noting the automobile road up the other side!

Robert J. Kleinhenz

A State Machine Hazard Analysis and Backwards Reachability

Whereas *system reliability* deals with the problems of ensuring that a system, including all hardware and software subsystems, performs a required task or mission for a specified time in a specified environment, *system safety* is concerned only with ensuring that a mishap does not occur in the process. Usually there are many possible system failures that have relatively little "cost" associated with them. Others have such drastic consequences that an attempt must be made to avoid them at all costs, perhaps even at the cost of attaining some or all the goals of the system.

5.1 Motivation

While software itself cannot be unsafe, it can issue commands to a system it controls that place the system in a unsafe state. Furthermore, the controlling software should be able to detect when factors beyond the control of the computer place the system in a hazardous state and to take steps to eliminate the hazard, or, if that is not possible, initiate procedures to minimize the hazard.

A mishap is an unplanned event or series of events that results in death, injury, illness, or damage to or loss of property or equipment. Mishaps can be classified according to their severity from catastrophic to negligible.

A hazard is a set of conditions within a state from which there is a path to a mishap. Hazards can be categorized by the aggregate probability of the occurrence of the individual conditions that make up the hazard and by the seriousness of the resulting mishap. Together, these constitute a measure of the *risk* of the situation. Risk is formally defined as the hazard level combined with the likelihood of the hazard leading to an accident (sometimes called danger) and hazard exposure or duration (sometimes called latency).

The first step in safety analysis is to identify the system hazards and assess their severity and probability (i.e. risk). Often early in the design of a system, the probabilities are unknown and the analysis is done considering only severity. For simplicity, the states of the system will be divided into two groups: high risk and low risk. High risk states are states that lead to catastrophic or unacceptable losses (hazards). It is important to note that in many, if not most, realistic systems it is impossible to completely eliminate risk: the goal is to design a system with "acceptable" risk.

The overall goal in designing a safety-critical system is to eliminate hazards from the design or, if that is not possible, to minimize risk by altering the design so that there is a very low probability of the hazard occurring. To show that a system is safe, or low risk, it is necessary to first ensure that given that the specifications are correctly implemented and no failures occur, the operation of the system will not result in a mishap. Second, the risk of faults or failures leading to a mishap must be eliminated or minimized by using specialized procedures. If it is not possible to eliminate completely the possibility of a hazard occurring, then in order to reduce risk the exposure time of the hazardous conditions must be minimized.

In order to determine whether or not a system can actually "reach" any high risk states, all possible states that a system can reach from the initial state due to a legal set of transition can be generated. This is called a forward reachability graph. However, generating the entire reachability graph may well be impractical due to the size of the graph for a complex system.

5.2 Informally Defining the Hazard Elimination Algorithm

One way to do a safety hazard analysis is to work backwards from the hazardous state to determine if it is reachable. This approach is useful when the goal of the analysis is to prove only that the system cannot reach certain hazardous states, which is often a requirement for safety-critical systems. Fault tree analysis is a similar technique used for the same purpose. The backward approach is itself practical only if one considers a relatively small number of high risk states. It must be noted that the concern here is with system safety, not with the correctness of the system. A system is "safe" if it is free from mishaps even if it does not accomplish its mission or functional objectives.

Now, if a deterministic finite automaton is considered, as defined in Section 3.1, the transition relation between a pair of states is defined as being a function. The transition function of the DFA can be used to determine if the hazardous state is reachable by using the hazardous state as the initial state and determining whether the original initial state is reachable. It is possible for the backward reachability graph to be as large as the original graph. Thus, a solution must not require the entire backward reachability graph to be generated. One possible solution defines and uses a type of state called a *critical* state.

Consider separating the states in a finite automaton into two disjoint sets:

1. States from which it is possible to reach only low risk states
2. States from which it is possible to reach high risk states and possibly also low risk states

A critical state is a low risk state from which it is possible to reach both a hazardous state and a low risk state. Thus, if a hazardous state is reachable, there must be a critical state on the path from the initial state to the hazardous state (including the possibility that the critical state is the initial state, as long as the initial state is low risk). Otherwise, the design needs to be completely redone since all executions result in hazardous states.

To ensure that a particular hazardous state can never be reached, it is possible to simply work backwards to the *first* critical state and to use design techniques to ensure the bad path is never taken. This technique is conservative, because in order to reduce the large amount of computing to produce the entire reachability graph, hazardous states may be eliminated that in reality may not have been reachable. However, it does no harm to eliminate the possibility of a mishap that would not have occurred. Moreover, eliminating a nonexistent path may have the effect of eliminating or lessening the possibility of mishaps caused by extraneous faults and failures.

The algorithm starts with the set of hazardous states. For each member of this set, the immediately prior state or states are generated. Each of these "one-step backward" states is then examined to see if it is a potential critical state and can be used to eliminate one path to the hazardous state. Finally, there exists only the need to look forward one step from each potentially critical state in order to label it as critical. This is because if the escape path used to eliminate the hazardous state in question leads eventually to another hazardous state, the hazardous part of escape path will be eliminated by a critical state that is a successor to the one being used to eliminate the original hazard in question.

5.3 Defining Hazard Automata

Let us define a *Hazard Automaton* A as being:

$$A = (Q, q_0, Q_L, Q_H, Z, \Sigma, \delta) \tag{5.1}$$

where Q represents the states of the system, q_0 the start state of the system,

$$Q_H \subseteq Q \tag{5.2}$$

and

$$Q_L \subseteq Q \tag{5.3}$$

and

$$Z \subseteq Q_H, \tag{5.4}$$

and

$$\delta: Q \times \Sigma \to Q \tag{5.5}$$

so that δ is the transition function that maps the present state to the next state where Σ is the input alphabet. Recall that the set of finite strings of elements of Σ is denoted by Σ^*.

Consider the states Q_H, Q_L to be the set of high risk states and low risk states, respectively, that are pre-determined by the engineer or system designer. These states obey the following properties:

$$Q = Q_H \cup Q_L \tag{5.6}$$

$$Q_H \cap Q_L = \varnothing \tag{5.7}$$

that is, Q_H and Q_L form a partition of Q. Furthermore, we assume:

$$\text{if } q \in Q_L \text{ then } \exists(\sigma \in \Sigma) \text{ such that } \delta(q, \sigma) \in Q_L \tag{5.8}$$

$$Z \subseteq Q_H \tag{5.9}$$

The initial state of the system must all be an element of the set Q_L. That is:

$$q_0 \in Q_L \tag{5.10}$$

Let us next define the notion of predecessor states, successor states and reachable states.

5.3.1 Predecessor, Successor and Reachable States

Predecessor, successor and reachable states can be defined quite easily in a Hazard Automaton A. Given a state q, the predecessor states of q are all of the states from which there exists a legal transition under a single given input that takes the predecessor state to the state q. Similarly, the successor states of q are all of the states for which there exists a legal transition under a single given input that takes the state q to the successor state. The reachable states of q are all of the states for which there exist a sequence of legal transitions, under a given input string that takes the state q to the reachable state. More formally:

Define the set of *successor* states S_q for a state q as:

$$S_q = \cup_{\sigma \in \Sigma}\{\delta(q, \sigma)\} \tag{5.11}$$

Define the set of *predecessor* states P_q for a state q by:

$$q' \in P_q \Leftrightarrow q \in S_{q'} \tag{5.12}$$

Define the *reachability function* δ^* such that:

$$\begin{aligned} \delta^*(q, \sigma w) &= \delta(\delta^*(q, w), \sigma) \\ \delta^*(q, \lambda) &= q \end{aligned} \tag{5.13}$$

where

$$\delta^*: Q \times \Sigma^* \rightarrow Q \tag{5.14}$$

for $\sigma \in \Sigma$, $w = w_1 w_2 \ldots w_n$ being a string over the alphabet of Σ, and λ being the empty input string. Thus, the set of *reachable* states R_q from the state q can be defined as:

$$q' \in R_q \Leftrightarrow \exists w \in \Sigma^*[\delta^*(q, w) = q']\,. \tag{5.15}$$

All states in R_q can be considered to be descended from, or have the common *ancestor* state q. Similarly, the set of *ancestor* states A_q can be defined as:

$$q' \in A_q \Leftrightarrow q \in R_{q'} \tag{5.16}$$

5.3.2 Critical States

Recall that the Hazard Automaton A possesses three sets of states, Q_H, Q_L and Z. The goal in a hazard analysis is to identify and remove hazards. This is equivalent to rendering all of the states in Z unreachable. The algorithm outlined in the next section attempts to do so by utilizing the notion of a *critical* state.

Define the set of critical states C such that:

$$q \in C \Leftrightarrow \begin{cases} q \in Q_L \wedge \\ \exists q' \in Q_H \text{ such that } q' \in S_q \wedge \\ \exists z \in Z \text{ such that } z \in R_q \end{cases} \tag{5.17}$$

5.4 Hazard Automaton Reduction Algorithm

The Hazard Automaton Reduction Algorithm (HARA) takes a hazard automaton $A = (Q, q_0, Q_L, Q_H, Z, \Sigma, \delta)$ as an input, along with a single identified hazardous state $z \in Z$ and produces as an output a "reduced" Hazard Automaton $A' = (Q', q_0', Q_L', Q_H', Z', \Sigma', \delta')$ with the hazardous state rendered unreachable[3]. A hazard automata $A' = (Q', q_0', Q_L', Q_H', Z', \Sigma', \delta')$ is a reduction of the hazard automata $A = (Q, q_0, Q_L, Q_H, Z, \Sigma, \delta)$ if:

$$Q' = Q \tag{5.18}$$

$$q_0' = q_0 \tag{5.19}$$

$$Q_L' = Q_L \tag{5.20}$$

$$Q_H' = Q_H \tag{5.21}$$

$$Z' = Z \tag{5.22}$$

$$\Sigma' = \Sigma \tag{5.23}$$

$$\delta' \subseteq \delta \tag{5.24}$$

3. HARA returns a "reduced" HA if it is possible to remove the hazard from the original HA without damaging the functionality of the design.

HARA disables the transitions in δ that lead to the hazardous state z. Repeated application of the Hazard Automaton Reduction Algorithm can be used to render all identified hazards in the automaton unreachable, as long as there exists at least a single path of low risk states from the initial state to a final state.

Given the inputs of a Hazard Automaton $A = (Q, q_0, Q_L, Q_H, Z, \Sigma, \delta)$ and a hazardous state $z \in Z$ to HARA, and the output of a reduced Hazard Automaton $A' = (Q', q_0', Q_L', Q_H', Z', \Sigma', \delta')$ by HARA, the correctness condtions on HARA are as follows:

$$Q' = Q \tag{5.25}$$

$$q_0' = q_0 \tag{5.26}$$

$$Q_L' = Q_L \tag{5.27}$$

$$Q_H' = Q_H \tag{5.28}$$

$$Z' = Z \tag{5.29}$$

$$\Sigma' = \Sigma \tag{5.30}$$

$$\delta - \delta' : \{(q, \sigma, q') | (q \in Q_L \wedge q' \in Q_H \text{ such that } q' \in S_q \wedge z \in R_{q'})\} \tag{5.31}$$

This last condition can also be stated in the form that:

$$\neg\exists w | \delta'^*(q_0, w) = z \tag{5.32}$$

indicating that the hazardous state z is no longer reachable in the reduced Hazard Automaton A'.

Consider the Hazard Automaton:

$$A = (Q, q_0, Q_L, Q_H, Z, \Sigma, \delta) \tag{5.33}$$

The following describes the details of the algorithm to identify and render the hazardous state of the Hazard Automaton unreachable[4]:

A'=**HARA**($A = (Q, q_0, Q_L, Q_H, Z, \Sigma, \delta)$,z)

4. The DFA has a transition enabled under all inputs, and a transition exists for all inputs.

```
% Initialize variables and Reduced Hazard Automata
States_to_Process = {z}
States_Processed = $\phi$
Ancestor_States   = $\phi$
% These are the ancestor states of the hazardous state z in the automata $A'$
Predecessor_States = $\phi$
% These are the predecessor states of the state $q$ in the automata $A'$
$\delta' = \delta$
$A' = (Q, q_0, Q_L, Q_H, Z, \Sigma, \delta')$
%Render Hazard $z$ unreachable in Automata
do while  States_to_Process -States_Processed $\neq \phi$
   choose $q \in$ States_to_Process
%Find Predecessor States of $q$ with respect to the automata $A'$: $P_{q, A'}$
    Predecessor_States = $P_{q, A'}$
%Eliminate each path from the predecessor state to the state $q$, consider each element of $P_q$ case by case
   begin case
      Case 1: $P_{q, A'} = \phi \wedge q \neq q_0$
%State $q$ is unreachable from any other state.
         States_to_Process = States_to_Process - $q$
         States_Processed = States_Processed + $q$
      Case 2: $P_{q, A'} \neq \phi \wedge (P_{q, A'} \cap \text{Ancestor\_States}) = P_{q, A'}$
%No new Predecessor states found; State is in an unreachable cycle
         States_to_Process = States_to_Process - $q$
         States_Processed = States_Processed + $q$
      Case 3: $P_{q, A'} \neq \phi \wedge (P_{q, A'} \cap \text{Ancestor\_States}) \neq P_{q, A'}$
%New Predecessor states found.
         Ancestor_States = Ancestor_States $\cup P_{q, A'}$
         States_to_Process = States_to_Process - $q$
         States_Processed = States_Processed + $q$
         for all $a \in$ (Ancestor_States-States_Processed)
            choose $a \in$ Ancestor_States-States_Processed
            if $a \in Q_L$
% $a$ is Critical State because it is a low risk state, has low risk sucessor (by definition of low risk state) and
%has the hazard $z$ in its reachability graph.
%Disable hazardous transition by removing from $\delta$
                  $\delta' = \delta' - (a, \sigma, q)$  where $((q \in Q_H) \wedge (z \in R_q))$
            else
%Go one more step backwards
                States_to_Process = States_to_Process  $\cup \{a\}$  - States_Processed
            endif
         endfor
      Case 4: $P_q = \phi \wedge q = q_0$
% State $q$ is initial state which is defined as a low risk state.  This case should have been caught in Case 3,
%and the initial state should have been used as a critical state.  An error has occurred.
         Terminate Algorithm and Return Error Message
      end case
   end while
Terminate algorithm and Return $A' = (Q, q_0, Q_L, Q_H, Z, \Sigma, \delta')$
end
```

The Hazard Automaton Reduction Algorithm takes as input the Hazard Automaton A and a single identified hazardous state z. It returns as an output the Reduced Hazard Automaton A', in which the hazard z is no longer reachable. The first section of the algorithm initializes some useful variables, and creates the Reduced Hazard Automaton A'. A *while* loop is then set up to terminate when the hazard z is no longer reachable in A'. The set of predecessor states of the hazard z with respect to the automaton A' (also referred to as $P_{q,A'}$) is generated. If there are new predecessor states generated, then the algorithm goes to Case 3, where the attempt is made to remove the reachability of the hazard z by disabling the transitions from the predecessor states to the hazardous state z. Case 3 attempts to accomplish this by considering each predecessor state individually, which is accomplished in the *for* loop. Each predecessor state is identified as being either high risk or low risk. If the predecessor state is low risk, then the transition between the low risk predecessor state to the hazard is removed, causing the transition function δ' in the Reduced Hazard Automaton A' to be modified. All further actions in the algorithm are taken with respect to this newly modified A'. If all of the predecessor states to the hazard z are low risk, then the *for* loop in Case 3 will eliminate all transitions to the hazardous state, and the algorithm will terminate at the *while* condition and return the Reduced Hazard Automaton $A' = (Q, q_0, Q_L, Q_H, Z, \Sigma, \delta')$.

However, some of the predecessor states of z may be high risk. These high risk predecessor states must be treated as hazardous states, and are added to the set of States_To_Process. The algorithm then reencounters the *while* condition and continues on as if all of the states in States_To_Process are hazardous, and the reachable paths to each one must be eliminated.

The algorithm is guaranteed to terminate due to the fact that, for any hazard automaton that posesses a path of low risk states from the initial state to the final state, a critical state must exist along the path from the inital state to the final state (if the hazard is reachable). This is clear because the initial state itself is a low risk state and has a low risk sucessor (due to the fact that there is a low risk path from the initial state to the final state), which means that if there is no other low risk predecessor to the hazard other than the initial state, the initial state itself will become the critical state for the hazard. If the hazard is not reachable from the initial state, then it will be identified by either Case1 or Case 2 of the algorithm. Case 1 deals with the situation where a path to the hazard is composed of only high risk states, and is not reachable from the initial state. Case 2 deals with the situation where the path to the hazard is trapped in a cycle of high risk states which is not reachable from the initial state. These paths are then removed from consideration in the algorithm, and do not result in a non-termination situation. Since these paths were never reachable to begin with, they do not impact the overall reachability of the hazard in question from the initial state. Case 4 should never be encountered, and is included for completeness' sake.

Hence, it can be concluded that the Hazard Automaton Reduction Algorithm can take the Hazard Automaton $A = (Q, q_0, Q_L, Q_H, Z, \Sigma, \delta)$ along with the hazardous state $z \in Z$ and produce a reduced automaton $A' = (Q, q_0, Q_L, Q_H, Z, \Sigma, \delta')$, where the hazardous state z has been rendered unreachable. All of the hazardous transitions from the low risk critical states leading to the high risk states that eventually lead to the hazard have been disabled.

5.5 Functionality of the Hazard Automaton Reduction Algorithm

Now, given the algorithm above, it is desirable to show that, if a Hazard Automaton has a path of low risk states from the initial state to a final state, then the Hazard Automaton Reduction Algorithm is capable of removing all of the hazards in the automaton. Given this sort of automaton, the Hazard Automaton Reduction Algorithm, when applied repeatedly, will produce a reduced automaton with the critical transitions excised. Hence, it can be said that the Hazard Automaton Reduction Algorithm can eliminate all hazardous behaviour in an automaton, if that automaton possesses a path of low risk states. More formally, this theorem can be stated:

Theorem 1:

Given a Hazard Automaton A capable of exhibiting a path of low risk states from the initial state to a final state, the Hazard Automaton Reduction Algorithm can eliminate all identified hazardous behaviour of the automaton. All identified hazardous behaviour of A being eliminated is equivalent to saying that all identified hazards are no longer reachable. This second framing of the statement shall be proved by contradiction.

Proof:

Suppose not.

This means that:

$$\exists z \in Z | \delta'^*(q_0, w) = z \tag{5.34}$$

The entire argument by contradiction hinges on the fact that there is a critical state along the path between each hazardous state and the initial state for any hazard automaton A that is capable of exhibiting non-hazardous behaviour. Without a critical state on the path between the hazardous and initial state in the original automaton A, the reachability of the hazardous state cannot be eliminated. If no critical state exists on the path between q_0 and z, then there is no low risk state on the path between q_0 and z that possesses any low risk successors. Thus, the initial state q_0 has no low risk successors, and thus can lead only to hazardous behaviour. This is in contradiction to the assertion that A posseses a path of low risk states starting from the initial state and ending in a final state (a state that has no outgoing transitions).

The only way for there to be no critical state in the set of ancestor states of the hazard would be if no initial state was an ancestor state of the hazard. If there is no path existing between the hazard and the initial state in the original automaton A, then the hazard is already unreachable:

$$\neg\exists\delta^*(q_0, w) = z, z \in Z \tag{5.35}$$

and since:

$$\delta' \subseteq \delta \tag{5.36}$$

we have a contradiction of (5.34).

□

Hence, we can say that HARA renders all identified hazards unreachable in the new automaton A', which is equivalent to saying that all identified hazardous behaviour in the reduced automaton A' is eliminated.

5.6 Optimality of the Hazard Automaton Reduction Algorithm

As well as eliminating all identified hazardous behaviour in an automaton, the HARA also ensures that no reachable, strictly desirable behaviour is eliminated. This is equivalent to saying that no sequence of purely non-hazardous, low risk states is eliminated. Thus, for an automaton A which is capable of exhibiting low risk behaviour, not only are all of the hazards removed, but no desirable low-risk behaviour sequences are removed in A. Thus, in some fashion, HARA can be construed as being optimal, in the sense that it removes all of the hazards whilst removing the minimum amount of non-hazardous behaviour traces at the same time[5]. More formally:

5. A sequence of desirable, low risk behaviour is defined as being a path or sequence of states, all of which are low risk. That is, this sequence of purely low-risk states does not have a hazardous predecessor along the path from initial state to final state.

Theorem 2:

Given a Hazard Automaton A capable of exhibiting a path of low risk states from the initial state to a final state, the Hazard Automaton Reduction Algorithm does not eliminate any sequence of reachable, desirable, non-hazardous low risk behaviour starting from the initial state.

Proof:

By contradiction. Suppose not.

Then,

$$\exists(q_0\sigma_1 q_1\ldots\sigma_i q_L),\quad \forall i \in N, \sigma_i \in \Sigma, q_i \in Q_L \tag{5.37}$$

in the original Hazard Automaton A and $q_L \in Q_L$. However, in A':

$$\neg\exists(\delta'^*(q_0, w) = q_L), w \in \Sigma^* \tag{5.38}$$

This means that, at some point, a transition must have been disabled in the original automaton, upstream of the low risk state, rendering it unreachable from the initial state in the new automaton A'. If there is no path from q_0 to q_L, then we have a contradiction to (5.37), as q_L was unreachable to begin with.

So there must be a critical state somewhere along the path from q_0 to q_L in the original automaton A in order for a transition to be disabled, thereby cutting off the reachability of the low risk state q_L. Thus,

$$\begin{aligned}\delta^*(q_0, w) &= \delta^*(\delta^*(q_0, w'), w'') \\ &= \delta^*(q_C, w'') \\ &= q_L\end{aligned} \tag{5.39}$$

where $q_C \in C, q_C \cap R_{q_0} \neq \phi$.

In order for the path to q_L to have been eliminated, a hazard must exist somewhere along the reachable path from q_0 to q_L and beyond. Now, if we recall from the definition of a critical state, only a path of high risk states must exist between the critical state and the hazardous state. The critical state is the first low risk state (with a low risk predecessor) encountered along the path from the hazard to the initial state. So, each critical state eliminates only the path of high risk states (and all of their successors) leading to the hazard. So, we must have, in A:

$$\exists(q_C \sigma_i q_i \dots q_{i+n} \sigma_{n+1} z) \tag{5.40}$$

where

$$\forall i \in N, \sigma_i \in \Sigma, q_C \in Q_L, z \in Z, q_i \in Q_H \tag{5.41}$$

There are two possible ways that the path of low risk states to q_L could have been eliminated.

1. The path of low risk states leading from q_C to q_L is a successor of one of the high risk states $q_i \in Q_H$ leading to the hazardous state z, or is a successor of z itself.
2. The hazard lies after the path of low risk states to q_L (z is successor to q_L).

5.6.1 Case 1

If the path of low risk states from q_C to q_L is a successor of one of the high risk states $q_i \in Q_H$ or of the hazard z itself, then the low risk state q_L has a hazardous or high risk predecessor along the direct path from the critical state to the low risk state in the original automaton A. Thus, the behaviour exhibited by the sequence of states, from the low risk critical state to the low risk eliminated state evinces a hazardous or high risk state. This is a contradiction to the assertion that a sequence of desirable, low risk, non-hazardous behaviour beginning in the initial state q_0 and terminating in the final state q_L is eliminated (5.37). After all, if in order to get to the low risk state q_L one has to pass through a high risk state $q_i \in Q_H$, the sequence of states that takes you from one to the other is not purely low risk.

5.6.2 Case 2

The hazardous state z lies after the state q_L. By definition of a low risk state:

$$\exists q' | \delta(q_L, \sigma) = q', q' \in Q_L, \sigma \in \Sigma \tag{5.42}$$

And now we are saying:

$$\exists w | \delta^*(q_L, w) = z \tag{5.43}$$

then we have by definition that q_L is a critical state. Since $q_L = \delta^*(q_C, w')$, the state q_L is encountered before the critical state q_C in HARA, making q_L the first low risk state with a low risk successor encountered due to the hazard z. Thus, the critical state q_L would be used to eliminate the hazard z in HARA, and hence could not itself be eliminated in the process[6]. Thus we have a contradiction, as q_C is no longer the first low risk state encountered in the backwards path search due to the hazard z.

□

5.7 Hybrid Extension of the Hazard Automaton Reduction Algorithm

Now, for the case of a hybrid automaton, we consider the LSVW [84] model:

$$A = (W, X, Q, \Theta, E, H, D, T) \tag{5.44}$$

as defined in Section 3.3.1. The question becomes whether or not we can reduce the hybrid case into the form of the discrete case, in which case the proceeding two proofs for functionality and optimality would still hold. The added difficulty of the hybrid case is that a hazardous state can occur either through a discrete transition or through continuous time evolution.

However, this problem can be circumvented. Let us create an augmented LSVW automaton $HA = (W, X', Q', \Theta', E, H, D', T)$ which considers the matter of risk. First augment the internal state variables X of the hybrid automaton H to X', by adding a single internal variable, called *Risk*. This is a variable which can possess one of three values: Low, High or Hazard. Depending on the state of $V = W \cup X$ at any given point in the continuous tra-

6. Note that the critical state is no longer critical due to hazardous state z as z would have been eliminated using ..

jectories T, or before and after a discrete transition, the variable *Risk* is assigned a value by an external engineer, depending on whether the state is low risk, high risk, or hazardous. So, for the augmented LSVW A', we have that:

$$X' = X \cup \{\text{Risk}\} \tag{5.45}$$

$$Q' \subseteq val(X') \tag{5.46}$$

$$\Theta' \equiv val(X') \tag{5.47}$$

$$D' = val(X') \times \{E \cup H\} \times val(X') \tag{5.48}$$

T' is the set of valuations for $V' = W \cup X'$ that obey suffix, prefix and concatenation closure in A' (5.49)

Once this assignment has been made, a hybrid Hazard Automaton HA can be created from the LSVW automaton A'. The hybrid Hazard Automaton can be defined as:

$$HA = (W, X', Q', \Theta', E, H', D'', T'') \tag{5.50}$$

where X', Q', Θ' are defined as in Equations (5.45-5.47) and:

$$H' = H \cup \{\varepsilon\} \tag{5.51}$$

$$D'' \subseteq Q' \times \{E \cup H'\}' \times Q' \tag{5.52}$$

T'' is the set of valuations for $V = W \cup X'$ that obey suffix, prefix and concatenation closure in HA (5.53)

A discrete ε-transition can be inserted into the transition map D' of the hazard automaton HA each time the state variable *Risk* changes in order to create the transition map D'. This acts to augment the discrete transition relation D' to include dummy transitions that do not affect the valuation V' of the automaton, but signal a change in the risk behaviour of the automaton.

With the augmented state set $V' = W \cup X'$ and the augmented transition relation D'', we can assert that a hazardous state can only occur as the result of a discrete transition. Thus, a coarse discretization of the continuous system with respect to risk has been achieved. The *Risk* variable acts to abstract away the behaviour of the continuous trajectories, and to create a superstate that possesses a commonality due to risk behaviour. Hence, a trajectory with risk behaviour "Low" can be seen as a superstate consisting of the infinite number of valuations within the trajectory. It can move to another superstate with *Risk* behaviour High via a discrete transition only. Thus, if we encapsulate the trajectories into superstates based on risk, the application of the Hazard Automaton Reduction Algorithm occurs just as it did in the discrete case.

5.7.1 Functionality Proof Sketch

Recall that the definition of functionality in the discrete case only guarantees that all hazards are removed from the automaton. No promises are made with respect to maintaining the operability of the automaton. Because the hybrid Hazard Automaton is discretized with respect to the *Risk* variable, it follows directly from the discrete proof that all of the hazardous superstates can be removed. For each hazardous superstate, if the predecessor superstate is achieved by an actual transition in the original D, then the calculation of successor superstates and an evaluation of their risk level determine whether or not the predecessor superstate is a critical superstate, and so on. If the predecessor superstate is achieved solely by an ε-transition, then this predecessor superstate is treated as having only the one hazardous superstate as a successor, and a further one-step backwards calculation must be performed.

Now, given the algorithm above, it is desirable to show that, if a hybrid Hazard Automaton HA is capable of exhibiting non-hazardous behaviour, the Hazard Automaton Reduction Algorithm is capable of removing all of the hazards in the hybrid Hazard Automaton. Given this sort of automaton, the Hazard Automaton Reduction Algorithm will produce a Reduced hybrid Hazard Automaton HA' with the the discrete transitions leading to eventual hazardous trajectories excised. Hence, it can be said that the Hazard Automaton Reduction Algorithm can eliminate all undesirable behaviour in a hybrid Hazard Automaton, if that hybrid Hazard Automaton is capable of exhibiting non-hazardous behaviour in its present design form.

5.7.2 Optimality

As with the previous subsection, the same argument applies with respect to the optimality of the hybrid Hazard Automaton. As well as eliminating all identified hazards in a Hazard Automaton, the Hazard Automaton Reduction Algorithm also ensures that no reachable, strictly desirable behaviour is eliminated. This is equivalent to saying that no sequence of purely non-hazardous, low risk behaviour is eliminated. Thus, for a hybrid Hazard Automaton HA capable of exhibiting low risk behaviour, not only are all of the hazards removed, but no desirable low-risk behaviour sequences are removed in HA'. Thus, in some fashion, the Hazard Automaton Reduction Algorithm can be construed as being optimal, in the sense that it removes all of the hazards whilst removing the minimum amount of non-hazardous behaviour at the same time[7].

5.7.3 Simulation Relation

The key to the hybrid proofs for functionality and optimality lies in the simulation relation between the original hybrid LSVW automaton which has a risk variable as a part of its internal variables X and the hybrid Hazard Automaton which has the extra epsilon transitions between continuous trajectory segments of different risk designations. If the hybrid automaton model of Lynch et al. [84] is considered for the purposes of the proof, the the augmented LSVW automaton which has *Risk* as a part of its internal variable set X can be denoted by:

$$A' = (W, X, Q, \Theta, E, H, D, T) \tag{5.54}$$

and the hybrid Hazard Automaton with the augmented transition map can be denoted by:

7. A sequence of desirable, low risk behaviour is defined as being a path or sequence of trajectories, all of which are low risk. That is, this sequence of purely low-risk trajectories does not have a hazardous predecessor along the path from initial state to final state

$$HA = (W, X, Q, \Theta, E, H', D', T) \tag{5.55}$$

where:

$$H' = H \cup \{\varepsilon\} \tag{5.56}$$

$$D' \subseteq Q \times \{E \cup H'\} \times Q \tag{5.57}$$

T is the set of valuations for $V = W \cup X$ that obey suffix, prefix and concatenation closure in HA (5.58)

such that ε is the dummy action which precipitates the discrete epsilon transition between continuous trajectory segments of different risk. Note that the states of the system for each automaton are identical, it is only the transition map that has been augmented.

Theorem 3:

A simulation relation R exists between the augmented LSVW hybrid automaton [84] A' and the hybrid Hazard Automaton HA. That is, HA implements A'.

Proof:

The mapping between the two hybrid automata is essentially the identity mapping. The only difference between the two automata exists in the addition of the "dummy" epsilon action and the "dummy" transition to the transition map of the hybrid Hazard Automaton. However, this epsilon transition can be mapped onto a continuous trajectory without problem, due to several properties of trajectories in the hybrid automaton model. The existence of point trajectories, and the fact that trajectories obey prefix, suffix and concatenation closure aid in creating a map from A' to HA. The three properties the simulation relation must obey are checked below.

5.7.3.1 Equivalence of Start States

Obviously, since the states Q of both A' and HA are identical, the start states Θ of A' and HA are identical.

5.7.3.2 Equivalence of Discrete Steps

The discrete transition map D' of HA contains all of the discrete transitions in the map D of A'. That is, $D \subseteq D'$. So, if α is an execution fragment of A' consisting of one discrete action surrounded by two point trajectories, with the first state in of the execution fragment being $x_{A'}$, then there is obviously a corresponding execution fragment β of HA which has the first state of the execution fragment β as $x_{HA} = x_{A'}$. Similarly the last states in the executions α, β are identical. Since the external actions of both A' and HA are identical, then $trace(\alpha) = trace(\beta)$.

5.7.3.3 Equivalence of Trajectories

It needs to be proved for α an execution fragment of A' consisting of one trajectory, with the first state of α being $x_{A'}$, there exists a closed execution fragment β of HA with the first state of β being x_{HA}, $trace(\beta) = trace(\alpha)$, and the last state of α mapping to the last state of β via the relation *R*.

The trajectories of A' and HA are identical, except for the existence of discrete epsilon transitions in HA. We must show that the introduction of the discrete epsilon transition in HA does not disrupt the correspondence of trajectories in A' via the relation *R*.

Consider the execution fragment α in A' which corresponds to the moment at which a discrete epsilon transition is taken in HA. The discrete epsilon transition is instantaneous, so there is no continuous evolution of the trajectory in A' taking place as it occurs. The execution fragment α consists of two points: the point $x_{A'}$ before the transition occurs and the point after the transition occurs. The corresponding trajectory β in HA consists of the discrete epsilon transition surrounded by two point trajectories. The first point in the trajectory x_{HA} of β is equivalent to the point $x_{A'}$ in α. A similar correspondence exists for the final points in the trajectories. The leftmost endpoint of any trajectory can have its external variables manipulated, due to the existence of point trajectories and because all trajectories are defined over a right-open interval. Given that the external actions and variables for A' and HA are identical, and that the ε action is internal, the leftmost endpoint of β can be manipulated if necessary in order to ensure $trace(\alpha) = trace(\beta)$.

□

Thus, a simulation relation exists between A' and HA. From Lynch et al. [84] we have that:

$$traces_{HA} \subseteq traces_{A'} \tag{5.59}$$

The coarsely discretized automaton HA exhibits the same external behaviour as A'. If we consider aggregate superstates based on risk, derived from trajectory fragments, the proofs for functionality and optimality of the Hazard Automaton Reduction Algorithm in hybrid automata resemble their discrete counterparts. If we classify trajectories based on their risk designation, we have two sets: T_L, T_H corresponding to low risk trajectories and high risk trajectories. That is:

$$\begin{aligned} &\tau_i \text{ projected on } X \in Q_L \rightarrow \tau_i \in T_L \\ &\tau_i \text{ projected on } X \in Q_H \rightarrow \tau_i \in T_H \end{aligned} \tag{5.60}$$

The set of all hazardous trajectories Z is a subset of the set of high risk trajectories. If we consider an execution fragment α, where $\alpha = \tau_0 a_1 \tau_1 a_2 \tau_2 a_3 \tau_3 a_4 \tau_4 \ldots$ is an action-trajectory sequence, we see that trajectories can be considered to have successors and predecessors. That is, in the execution fragment α, τ_2 has the successor trajectory τ_3 and the predecessor trajectory τ_1. It also has the ancestor trajectory τ_0 and the descendent trajectory τ_4. Again, the sets of predecessor, successor, ancestor and descendent trajectories can be denoted by T_P, T_S, T_A, T_D respectively. We add the condition that a low risk trajectory must have at least one low risk successor trajectory. The first state in the trajectory τ_i is denoted by $firstq\tau_i$ and the final state in the trajectory is denoted by $lastq\tau_i$. So, the relation:

$$firstq\tau_{i+1} = \delta(lastq\tau_i, a) \text{ where } a \in \{E \cup H'\} \tag{5.61}$$

holds.

A critical trajectory is the first low risk trajectory which precedes a hazardous trajectory, and has an immediate low risk successor trajectory. That is, $\tau_j = \tau_C$ is a critical trajectory if there exists the execution fragments $\alpha, \alpha', \alpha'' \in frags_{HA}$ such that:

$$\alpha = \tau_i a_{i+1} \tau_{i+1} \ldots \tau_j \quad \text{where } \tau_j \in T_L \tag{5.62}$$

and

$$\begin{array}{c} \alpha\alpha' \in frags_{HA} \text{ where} \\ \alpha' = a'_{j+1} \tau'_{j+1} \ldots \tau'_k \quad \text{and } \tau'_k \in Z \end{array} \tag{5.63}$$

and

$$\begin{array}{c} \alpha\alpha'' \in frags_{HA} \text{ where} \\ \alpha'' = a''_{j+1} \tau''_{j+1} \text{ and } \tau''_{j+1} \in T_L \end{array} \tag{5.64}$$

The set of critical trajectories is denoted by C.

The hybrid form of the Hazard Automaton Reduction Algorithm takes the hybrid Hazard Automaton $HA = (W, X, Q, \Theta, E, H, D, T)$ and produces the Reduced hybrid Hazard Automaton $HA' = (W, X, Q, \Theta, E, H, D', T)$ where:

$$D' = D - (lastq\tau_c, a, firstq\tau_H) \tag{5.65}$$

where:

$$lastq\tau_c \in C, firstq\tau_H \in T_H \tag{5.66}$$

and

$$\exists(\alpha \in frags_{HA}) | \tau_i a_i \ldots \tau_z, \text{ where } firstq\tau_i = firstq\tau_H \text{ and } \tau_z \in Z \tag{5.67}$$

5.7.4 Proof of Functionality

Now, given the Hazard Automaton Reduction Algorithm, it is desirable to show that, if a hybrid hazard automata is capable of exhibiting non-hazardous behaviour, the Hazard Automaton Reduction Algorithm is capable of removing all of the hazards in the automaton. Given this sort of automaton, the Hazard Automaton Reduction Algorithm will produce a reduced hybrid hazard automaton with the critical transitions excised. Hence, it can be said that the Hazard Automaton Reduction Algorithm can eliminate all undesirable behaviour in a hybrid hazard automaton, if that hybrid hazard automaton is capable of exhibiting non-hazardous behaviour in its present design form. More formally, this theorem can be stated:

Theorem 4:

Given a hybrid Hazard Automaton HA capable of exhibiting low risk behaviour, the Hazard Automaton Reduction Algorithm can eliminate all identified hazardous behaviour of the hybrid hazard automaton. All identified hazardous behaviour of HA being eliminated is equivalent to saying that all identified hazardous trajectories are no longer reachable. This second framing of the statement shall be proved by contradiction.

Proof:

Suppose not.

This means that there exists some execution α with the trajectory $\tau_i = \tau_H$ where:

$$\exists \tau_i \in Z | \alpha = \tau_0 a_1 \tau_1 \ldots a_i \tau_i \tag{5.68}$$

The entire argument by contradiction hinges on the fact that there is a critical trajectory along the path between each hazardous trajectory and the initial trajectory for any automaton that is capable of exhibiting non-hazardous behaviour. If no critical trajectory exists on the path between τ_0 and τ_H, then there are no low risk states on the path between τ_0 and τ_H that possesses any low risk successors. Thus, the initial state $\theta_0 = first q \tau_0$ has no low risk successors, and thus can lead only to hazardous behaviour. This is in contradiction to the assertion that HA is capable of exhibiting low risk behaviour.

So, given that there must exist a critical trajectory τ_c along the path from θ_0 to τ_H there exists the execution α_H such that:

$$\begin{aligned} \alpha_H &= \tau_0 a_1 \tau_1 \ldots a_j \tau_j a_{j+1} \ldots \tau_H \\ \alpha_H &= \alpha\alpha' \end{aligned} \tag{5.69}$$

where

$$\begin{aligned} \alpha &= \tau_i a_{i+1} \tau_{i+1} \ldots \tau_j \quad \text{and } \tau_j \in T_L \\ \alpha' &= a'_{j+1} \tau'_{j+1} \ldots \tau'_k \quad \text{and } \tau'_k \in Z \end{aligned} \tag{5.70}$$

and $\tau_j = \tau_C \in C$. This means that:

$$\begin{aligned} &\exists \alpha'' \in frags_{HA} \text{ where} \\ &\alpha\alpha'' \in execs(HA) \\ &\alpha'' = a''_{j+1} \tau''_{j+1} \text{ and } \tau''_{j+1} \in T_L \end{aligned} \tag{5.71}$$

which means there must have been, in the original automaton HA:

$$\exists lastq\alpha | firstq\alpha' = \delta(lastq\alpha, a) \text{ where } a \in \{E \cup H'\} \tag{5.72}$$

However, if τ_C is the first critical trajectory encountered in the execution α_H in a backwards reachable fashion from hazardous trajectory τ_H on the path to the initial trajectory τ_0, then the transition in (5.69) should have been disabled by HARA. If τ_C is not the first critical state encountered, that is, there is a prior critical trajectory encountered in a backwards reachable fashion from the hazardous trajectory τ_H on the path to the initial trajectory, then the hazardous trajectory should already have been rendered unreachable by HARA using this prior critical state, and the execution α_H should no longer be hazardous. Thus, there is a contradiction.

□

Hence, we can say that HARA renders all identified hazards unreachable in the new automaton HA', which is equivalent to saying that all identified hazardous behaviour is eliminated.

5.7.5 Proof of Optimality

As well as eliminating all identified hazards in an automaton, HARA also ensures that no reachable, strictly desirable behaviour is eliminated. This is equivalent to saying that no sequence of purely non-hazardous, low risk behaviour is eliminated. Thus, for an automaton HA which is capable of exhibiting low risk behaviour, not only are all of the hazards removed, but no desirable low-risk behaviour sequences are removed in HA'. Thus, in some fashion, HARA can be construed as being optimal, in the sense that it removes all of the hazards whilst removing the minimum amount of non-hazardous behaviour at the same time[8].

Theorem 4:

Given a hybrid hazard automaton HA capable of exhibiting low risk behaviour, HARA does not eliminate any execution of purely low risk behaviour.

Proof:

By contradiction. Suppose not.

Then,

$$[\exists\alpha_L | (\alpha_L \in execs_{HA})] \wedge [\neg(\exists\alpha_L) \in execs_{HA'}] \tag{5.73}$$

How would this come to pass? This would mean that in the original automaton HA

$$\{\exists\alpha_L | \alpha_L = \tau_0 a_1 \ldots \tau_L\} \text{ such that } (\forall i)(\tau_i \in T_L) \tag{5.74}$$

but the corresponding execution no longer exists in HA'.

This is clearly impossible as the only way for any execution fragment to be eliminated would be if a transition were removed from the transition map D''. Transitions are removed by HARA clearly only between low risk trajectories and high risk trajectories.

8. A sequence of desirable, low risk behaviour is defined as being a sequence of action-trajectory pairs, for which all trajectories are low risk.

The purely low risk execution α_L has no change in risk behaviour, and thus no transitions would need to be removed. Thus, we have an obvious contradiction.

$\square$

Hence, we have proved that HARA is both functional and optimal for both regular and hybrid hazard automata.

In the next two chapters, the Hazard Automaton Reduction Algorithm will be used in conjunction with backwards reachability controls techniques to remove hazardous behaviour from two aeronautical systems. The first example is the Altitude Switch, which is discrete in nature. The second example is the Medium Term Conflict Detection algorithm for aircraft conflicts, which is a hybrid system.

CHAPTER 6

How can it be that mathematics, being after all a product of human thought independent of experience, is so admirably adapted to the objects of reality?

Albert Einstein (1879 - 1955)

Altitude Switch Example

In the next two chapters, a backwards reachability hazard analysis will be performed on two examples: an aircraft altitude switch and an aircraft Medium Term Conflict Detection algorithm. The altitude switch is a discrete system, in that it involves only discrete transitions in order to change states. The altitude switch is converted from a SpecTRM-RL model into a state space model, and then the Hazard Automaton Reduction Algorithm is applied in conjunction with the controls reachability techniques outlined in Chapter 4. A hazardous state is then analyzed in the context of the algorithm. The backwards reachable path of the hazardous state is calculated until a critical state is reached, and then a constraint is postulated in order to eliminate this hazardous path.

6.1 The Altitude Switch

The altitude switch (ASW) is a reusable component that turns power on to a device of interest (DOI) when the aircraft descends below a threshold altitude (2,000 feet) above ground level. The ASW receives altitude information from an analog radio altimeter and from two digital radio altimeters, with the altitude taken as the lowest valid altitude seen. If the altitude cannot be determined for more than two seconds, the ASW indicates a fault by failing to strobe a watchdog timer. A fault is also indicated if internal failures are detected in the ASW. The detection of a fault turns on an indicator lamp within the cockpit.

The ASW receives a status indication from the DOI indicating whether the DOI is powered on. If the DOI does not indicate that it is powered on within two seconds after power is applied, a fault is indicated by failing to strobe the watchdog timer. The ASW does not apply power to the DOI if the DOI is already powered on. If the DOI is powered off after the aircraft descends below the altitude threshold, the ASW does not reapply power to the DOI unless the aircraft again descends below the threshold altitude.

The ASW also accepts an inhibit signal that prevents it from turning on power to the DOI or indicating a fault. All other ASW functions are unaffected by the inhibit signal. The ASW also accepts a reset signal that returns it to its initial state [96].

The ASW interfaces with the following external devices, as currently implemented: two digital altimeters, one analog altimeter, the watchdog timer, the cockpit interface and the DOI (See Fig. 6.1).

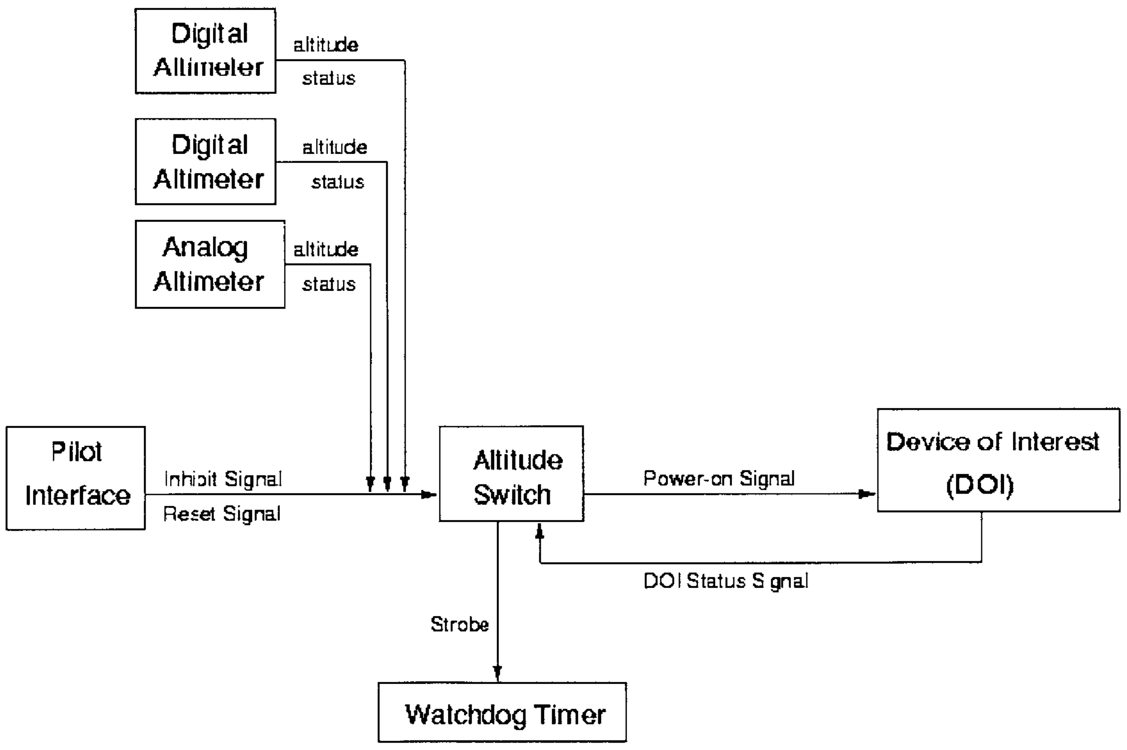


Figure 6.1: Altitude Switch Component Diagram

The DOI can be any aircraft component that can receive an electrical signal to control its operation and provide information about its status. The watchdog timer is used to determine the failure of the altitude switch or the inability of the altitude switch to ascertain the aircraft altitude within a certain period of time. Input messages to the ASW contain the altitude and status from the analog radio altimeter and the altitude and status from the two digital radio altimeters, along with the inhibit and reset signals from the cockpit interface, and the DOI status from the DOI. Output messages include the Power-on signal to the DOI and the strobe signal to the watchdog timer.

Safety analysis of the ASW depends on the DOI. If the DOI is non-safety critical, for example, a dimmer switch for passenger reading lights, then there are no real safety implications due to the ASW. However, if the DOI is the landing gear, then the ASW becomes safety critical. For the purposes of this dissertation, it shall be assumed that the DOI is a safety critical device, which must be activated once the altitude threshold has been breached.

6.2 SpecTRM-RL Model of the Altitude Switch

A state machine diagram of the ASW in the SpecTRM-RL modeling language is shown in Figure 6.2 (see next page). There are state variables for the statuses of each of the three altimeters, which have three possible values: Unknown, Valid or Invalid. There is a state variable for the status of the device of interest, which can assume the values Unknown, On, Off or FaultDetected. The state variable Altitude has four possible values, Unknown, Below Threshold, AtOrAboveThreshold and CannotBeDetermined. The initial state of all state variables upon startup is Unknown.

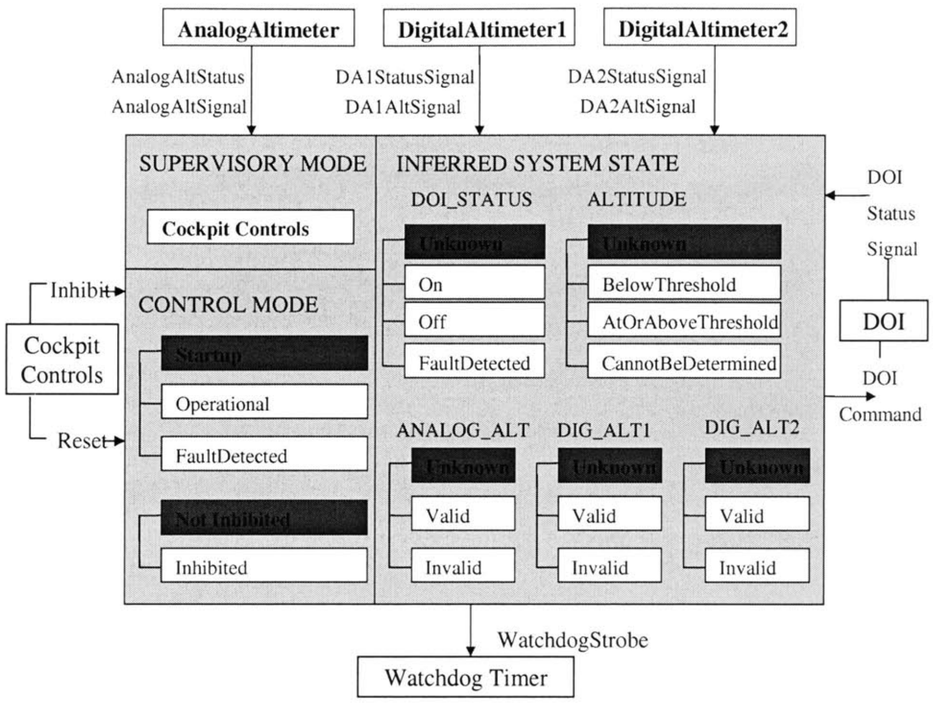


Figure 6.2: SpecTRM-RL Model of Altitude Switch upon Startup

There are three control modes in which the ASW can be operated: Startup, Operational, and FaultDetected. The ASW can also transition to the Inhibited control mode if the pilot presses the inhibit button on the cockpit interface. The ASW begins in Startup mode upon initialization, then transitions into Operational mode if no faults are detected. The ASW transitions into FaultDetected mode if there is a failure of the DOI to turn on, or if the altitude cannot be determined, or if there is an internal fault, such as remaining in Startup for more that 3 seconds. The default control mode of the ASW is Startup. The truth tables governing all of the transitions for the Altitude Switch Model in SpecTRM-RL are given in Appendix A.

6.3 Analysis of Hazardous Situation

Consider the obviously hazardous state of the system whereby the state variable Altitude has the value BelowThreshold and state variable DOI_Status is Off (see Figure 6.3). In addition, assume that the rest of the ASW is behaving normally, that is, the system is in

Operating control mode, and the inhibit command has not been invoked. This means that the aircraft has passed below the minimum altitude of 2,000 feet but the landing gear has failed to deploy. The question becomes, is it possible to reach this state?

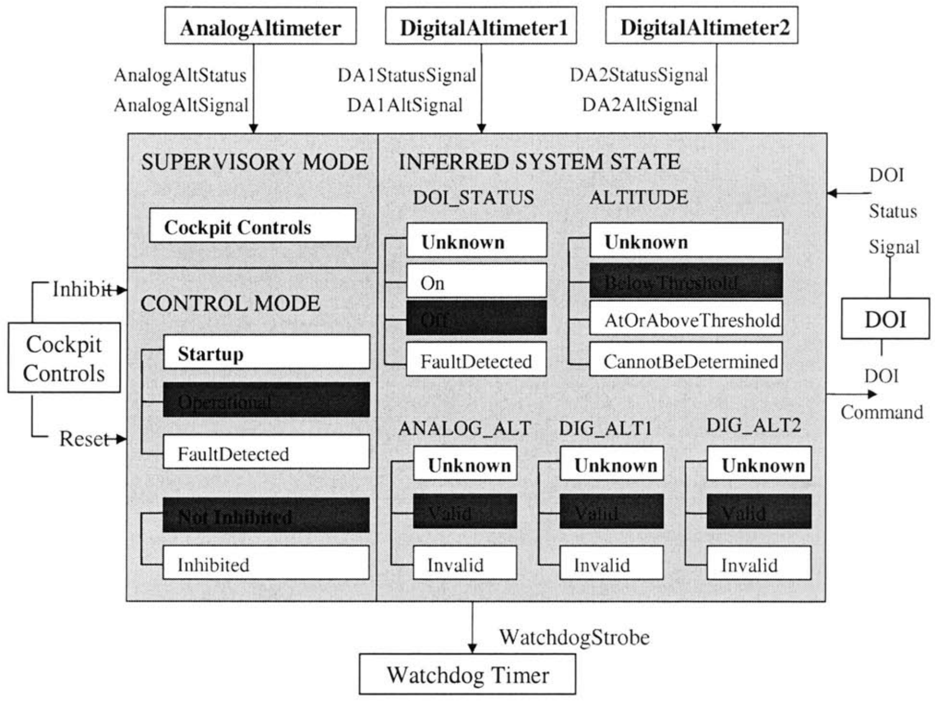


Figure 6.3: Hazardous State of Altitude Switch Model

Ideally, to answer this question, the state machine model of the Altitude Switch must be converted into a state space model, as per Section 4.5, and the state transition matrix *A* and input matrix *B* must be constructed. Consider a single component of the altitude switch model, for instance, the DOI. Recall that the state variable DOI_Status has four possible values: Unknown, On, Off and FaultDetected. The latent variables x_i corresponding to the state values must be created: x_1 corresponding to the value Unknown, x_2 corresponding to On, x_3 corresponding to Off and x_4 corresponding to FaultDetected. If the value of the latent variable is non-zero, then it is possible for the DOI_Status to assume the state machine value corresponding to the latent variable. For instance, given an initial state x(0), after k transitions, if the latent variables x_1,x_2,x_3 and x_4 are, respectively, 1,0,1 and 1, this means that at the kth step, the value of DOI_Status can be Unknown, Off or FaultDetected. The valuation On is not reachable in k steps from the initial state x(0) as $x_2(k)=0$, and thus the state variable cannot assume that value.

Examining the logic of the ASW upon startup, the value of DOI_Status is initialized to Unknown. The ASW then receives an input from the DOI. If no input is received, the DOI_Status remains in Unknown until an input is received from the DOI regarding its status. This input received can either be On or Off. The DOI must continually send feedback to the ASW as to its status. If more than two seconds pass since the last signal was received from the DOI regarding its status, the variable DOI_Status will transition to Unknown, unless the present value is FaultDetected.

Consider the input values received by the ASW from the DOI to be either On or Off. Depending on which value the input takes, the DOI_Status value transitions from Unknown to On or Off just after startup. The DOI_Status can transition to FaultDetected only under two scenarios. The first situation occurs as follows:

1. The ASW sends a signal to the DOI commanding it to turn on
2. Feedback is received from the DOI within two seconds stating that the DOI has not turned on (i.e. DOI status is still off),
3. A fault is then detected, and the variable DOI_Status transitions to Fault Detected.

The second circumstances occur as follows:

1. The ASW sends a signal to the DOI commanding it to turn on
2. Two seconds elapse and no feedback has been received from the DOI regarding its status
3. A fault is then detected and the variable DOI_Status transitions to FaultDetected.

If three input variables, u_1, u_2 and u_3 are created, corresponding respectively to the DOI_Status_Signal values of On, Off and Unknown, the behavior of the state variable DOI_Status can be represented in control state space as:

$$\begin{bmatrix} x_1(k+1) \\ x_2(k+1) \\ x_3(k+1) \\ x_4(k+1) \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{3}} & 0 & 0 & 0 \\ \frac{1}{\sqrt{3}} & 0 & 0 & 0 \\ \frac{1}{\sqrt{3}} & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1(k) \\ x_2(k) \\ x_3(k) \\ x_4(k) \end{bmatrix} + \begin{bmatrix} 0 & 0 & \frac{1}{\sqrt{2}} \\ 1 & 0 & 0 \\ 0 & \frac{1}{\sqrt{2}} & 0 \\ 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} u_1(k) \\ u_2(k) \\ u_3(k) \end{bmatrix} \tag{6.1}$$

where the columns of A and B have been normalized, and the presence of input is indicated by $u_i(k) = 1$.

So, for instance, if we start in the Off configuration of the DOI_Status (i.e. $x_3=1$, $x_1=x_2=x_4=0$), and then receive an input of On (i.e. $u_1=1$ and $u_2=u_3=0$), the final configuration becomes $x_1=x_3=x_4=0$ and $x_2=1$. Thus, the DOI_Status has changed from Off to On from the k^{th} step to the $k+1^{th}$ step. Using equations (4.6) and (4.7) from Chapter 4 the reachable space of the DOI after any number of steps can be determined.

However, the components of the ASW are highly coupled. The status of the DOI changes from On to Off when the value of the Altitude state variable becomes BelowThreshold. The value of the Altitude variable can become BelowThreshold only if all three altimeter statuses are valid, and all three altimeters input a value below the threshold of 2000 feet. The altimeter statuses are valid only if the ASW has received inputs within 2 seconds of the last input. It becomes immediately apparent that the entire state transition matrix must be constructed simultaneously, not on a component by component basis. However, the block-like nature of each component is preserved in the structure of the A matrix, and with the appropriate choice of latent variables x_i. This leads to a structure of Jordan chains in the A matrix, which is a direct indicator of the modal behaviour of the system. Each Jordan chain represents a mode which can be excited independently, given a judicious choice of input (See Appendix B for full A and B matrices).

6.4 Finding the Critical State

In a manner similar to that outlined above, the overall twenty-two row by twenty-two column state transition matrix A can be constructed, as well as the input matrix B (See Appendix B). Once these matrices are known, the entire state space can be quantified by performing twenty-two matrix multiplications using equation (4.6). However, performing twenty-two matrix multiplications is still a laborious task. Instead of exploring the entire state space, we wish to simply explore the reachability of a hazardous state until a critical state is reached. Exploring the reachability graph to a critical state would greatly reduce the number of matrix multiplications required, since it becomes unnecessary to reach the initial state and generate the entire graph. The technique would be to encode the hazardous state as an initial state, propagate the matrix multiplication backwards by one step, and then translate the latent variables back into their state variables to recover all of the predecessor states. Then the risk level of each decoded state can be determined. If the state is revealed to be a low risk state, then only one simple forward multiplication of the matrix is necessary in order to determine if it has low risk successors. Essentially, the backwards reachability of a hazardous state x(k) is determined by calculating:

$$x(k) = A^k x(0) + [A^{k-1}B|A^{k-2}B|\ldots|B]\begin{bmatrix} u(0) \\ u(1) \\ \ldots \\ u(k-1) \end{bmatrix} \qquad (6.2)$$

If the columns are calculated sequentially from right to left, and each column is unwound into its states, which are then tested to determine their criticality, the computational intensity of the algorithm is mitigated. Note that this calculation is performed under all inputs u(i), so that a subspace vector is generated in each multiplication with the input.

A critical state can be found for the hazard depicted in Figure 6.3, in which the altitude dips below the threshold value but the DOI does not turn on. This critical state occurs when the altitude is at or above the threshold value, and the DOI-Status is Off (see Figure 6.4). This is a critical state because it is a low risk state, but has a high risk descendant, the aforementioned hazard. To control the reachability of the hazardous state from the critical state, one need only ensure that the logical specification of the ASW has a predicate which couples the change in the value of the Altitude variable to an enforced change in the value of the DOI_Status variable. If the valuation for the Altitude variable changes from AtOrAboveThreshold to BelowThreshold, then the value of the DOI_Status variable must change either to On, or the control mode of the ASW must change from Operational to FaultDetected.

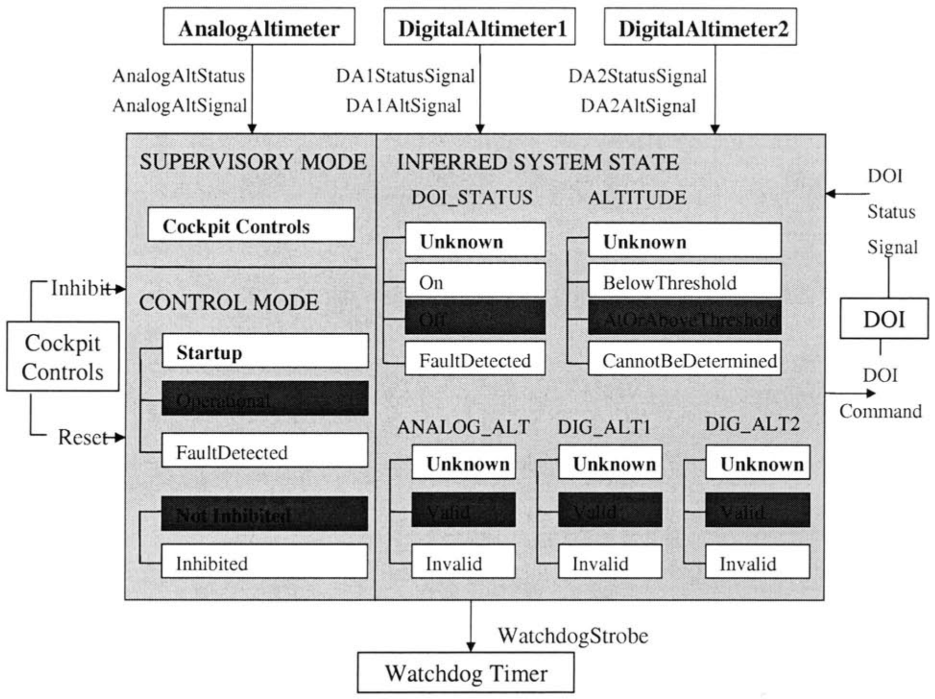


Figure 6.4: A Critical State Corresponding to Previous Hazard

The change due to the imposed coupling constraints should become apparent in the structure of the state transition matrix *A*. Recall that each state machine value is a latent variable x_i. The latent variable corresponding to the value BelowThreshold of the state variable Altitude should have its transition coupled with the transition of the latent variable corresponding to the On value of the state variable DOI_Status. Otherwise, the latent variables that represent the control modes of the ASW should be forced to change if the DOI_Status does not transition. That is, the state transition matrix *A* in the control formulation of the ASW should not possess a vector in its subspace that has the latent variables for the Altitude, DOI and/or control modes decoupled. Under no circumstances can Altitude be changed without changing either DOI_Status or control modes.

These coupling constraints will change the reachability matrix seen in equation (6.2), acting to make some of the columns linearly dependent of each other, thereby illustrating that some of the state space is no longer reachable.

The technique of searching backwards in a control state space is a generic method, which has the potential to scale up for very large systems. There are already a great many matrix manipulation packages available, some of which are adept at handling very large matrices (Matrix X, Matlab etc.). Control theorists have many tools that automatically check for the reachability of continuous controls state spaces, and these tools can be adapted to check for the reachability of state machines. The fact that there are so many established control tools is useful in the sense that a great deal of the development work has been done, and commercial off-the-shelf technology can be employed once an interface has been written to accurately translate state machine specifications into control theory state space representations.

6.5 Comparison with Other Methods of Hazard Analysis

Commonly used methods for software hazard elimination in industry are still the veteran techniques of simulation and testing. Although provably effective in the very early stages, when the design is still infested with many hazards, the effectiveness of testing and simulation drops as the design becomes cleaner, and requires an alarming amount of time to discover increasingly more subtle bugs. These techniques are dependent on the human who is creating the test cases or simulation scenarios. If that person never considers a certain given aspect of the environment of the system, then large amounts of the program may never be tested or simulated. A serious problem is that we are never sure when the techniques have reached their limits and have no estimate of how many hazards may still lurk in the design. As the complexity of designs increase, it is possible for these methods to completely collapse due to their inability to scale up properly [25].

An alternative to simulation and testing is the approach of formal verification. Formal verification conducts an exhaustive exploration of all possible behaviors of the system. Hence, when a design is pronounced correct by a formal verification method, it implies that all behaviors have been explored, and questions of adequate coverage become irrelevant. Several approaches to formal verification exist. There is considerable research on the subject of theorem proving, term rewriters and proof checkers for verification. However, these techniques can be computationally intensive. Additionally, an extensive background in logic and theorem proving is required in order to efficiently use a theorem prover [104].

Alternatively, model checking is an approach to verification in which a desired behavioral property is checked over a given system model through exhaustive enumeration of all the states reachable by the system. Model checking is fully automated, and its application requires no user supervision. Anyone qualified to run a simulation of the model is equally qualified to run the model checker. A model checker will provide a counterexample that demonstrates the behavior that violates the property being checked [90]. The main disadvantage of model checking is that state explosion can occur if the system being verified has many components that make transitions in parallel. In this case the number of global system states may grow exponentially with the number of processes. Because of this problem, some researchers in formal verification believe that model checking may not be practical for very large complex systems.

The most successful technique, to date, for dealing with these software model checking problems is based on partial order reduction [45], whereby concurrently executed events appear arbitrarily ordered with respect to one another. As a result, the number of states that are needed for model checking is reduced. The methods of compositional reasoning [27], induction [17], symmetry [24] and abstraction [7] have also been used to try to reduce the state explosion problem in model checking with varying success.

Although symbolic representations and partial order reduction has greatly increased the size of systems that can be verified, many realistic systems are still too large to be handled. Thus, it is important to find techniques that avoid the need to explore the entire state space of the model. The Hazard Automaton Reduction Algorithm allows hazards to be controlled or eliminated without generating the entire reachability graph. This vastly reduces the state explosion problem, making the problem of hazard elimination more tractable. When employed with state space control theory reachability results, the problem reduces to performing a finite number of matrix multiplications. Hence, an approach based on the Hazard Automaton Reduction Algorithm coupled with control theory results seems to be a promising method by which to control or eliminate hazards in very large complex systems.

For smaller models, the Hazard Automaton Reduction Algorithm could be coupled with model checking techniques to remove hazards. However, when the hazards become farther removed from their critical states, as sometimes happens in large models, the beneficial effects of Hazard Automaton Reduction Algorithm are greatly diminished and the state explosion problem emerges once again. This is not the case if the state space controls technique is employed, as all of the cost goes into the creation of the large transition matrix. Each successive step backwards involves only one *n* by *n* matrix manipulation, so computational cost does not increase drastically with the distance between the hazard and the critical state.

The benefits are even greater for hybrid models. SpecTRM-RL has been extended to include models of a continuous nature. A hybrid model of a Medium Term Conflict Detection algorithm for aircraft is next analyzed using Hazard Automaton Reduction Algorithm and controls reachability techniques. For hybrid models, the Hazard Automaton Reduction Algorithm cannot be coupled with model checking techniques because there are no termination guarantees in backwards reachability using model checking.

CHAPTER 7

Knowledge must come through action; you can have no test which is not fanciful, save by trial.

Sophocles (495 BC - 406 BC), Trachiniae

Medium Term Conflict Detection Example

Medium Term Conflict Detection (MTCD) is a conflict detection algorithm which will be used to support Air Traffic Controllers (ATCOs) in their task of monitoring and separating aircraft. MTCD can be modelled as having both continuous and discrete parts, and falls under the rubric of hybrid modelling. In this chapter, some background regarding the purpose and function of MTCD will be given, as well as highlighting certain artifacts of the algorithm which make it difficult to uniquely determine whether a conflict has occurred. The evolution of a hazard in the hybrid SpecTRM-RL model of MTCD, and its ultimate elimination using the Hazard Automaton Reduction Algorithm will be detailed.

7.1 MTCD Background

In today's air traffic control (ATC) environment, controllers monitor flights by scanning flight progress strips and radar displays, in order to predict future air situations. The controllers are monitoring one aircraft and relating its movements to the total air situation at all moments in the near future through the sector airspace. Controllers are also responsible for resolving any conflicts that occur. Currently, the majority of flights use fixed point routings along fixed air traffic service (ATS) routes with limited capacity. One possible way to increase the capacity of an airspace is to allow random point routings, i.e., routings that do not follow the fixed ATS routes. Increasing air traffic and increasing use of random point routings impose an even greater workload on controllers, especially in high density areas. Automated support can help to keep the workload of controllers within acceptable and safe limits.

The Medium Term Conflict Detection function will assist controllers in monitoring the air situation continuously and provide conflict data to the controllers through the human machine interface (HMI). Controllers monitor this operational data on situation displays. Controllers also remain responsible for the assessment of conflicts, as well as reacting to them. MTCD must provide controllers with enough time to assess, and, if necessary, resolve the conflict by deliberate action.

MTCD supports conflict detection for all flights for which a system trajectory is available. Since the trajectory data that MTCD receives is accurate to only a certain degree, MTCD creates an uncertainty area around the trajectory data, and uses this expanded volume for the purposes of conflict detection. MTCD begins conflict detection for a flight when it is a pre-defined time from entering the area of operation, and continues conflict detection until the flight leaves the area entirely.

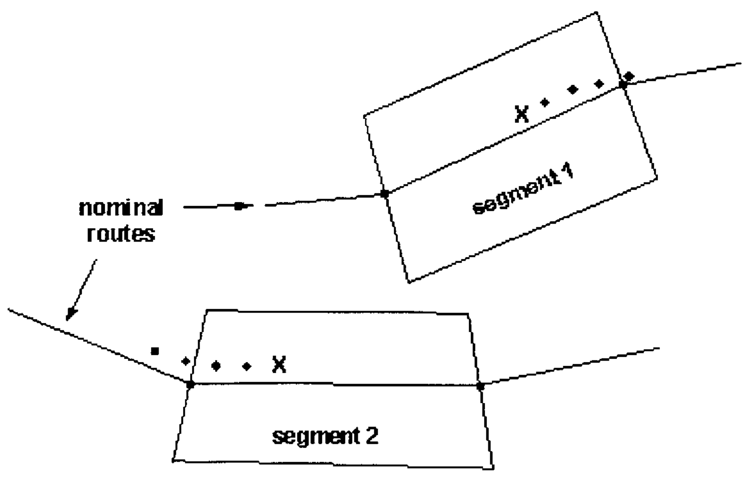


Figure 7.1: Aircraft Conflict (Buffer Violation)

MTCD detects four types of conflicts:

1. Aircraft Conflicts: Loss of separation between probable positions of two aircraft, based on system trajectories and uncertainty areas, the latter are introduced to take minor deviations into account
2. Nominal Route Overlaps: Loss of separation between system trajectories of two aircraft
3. Special Use Airspace Penetrations: Loss of the required distance between probable positions of an aircraft and a special use airspace.
4. Descent Below Lowest Usable Flight Level: Probable positions of an aircraft within an airspace is below the minimum altitude proscribed for that airspace.

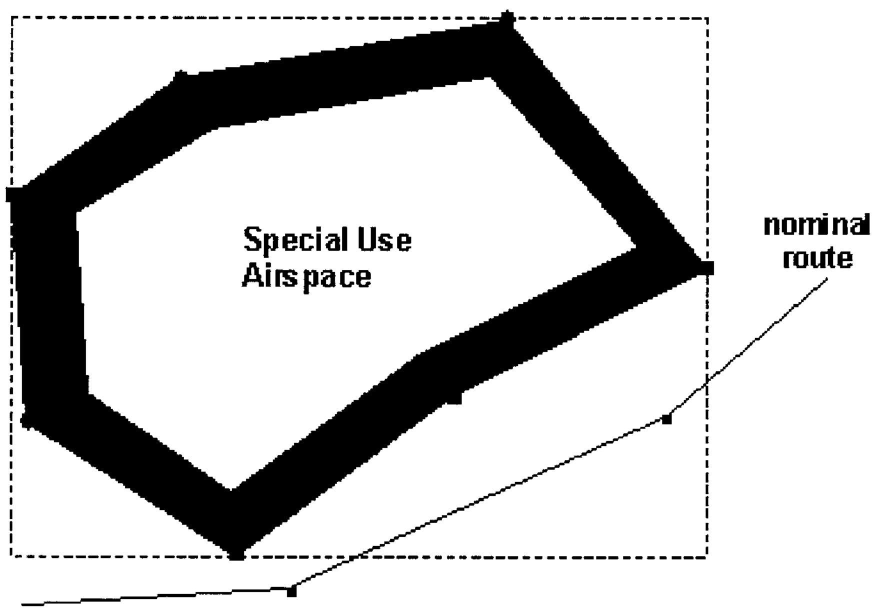


Figure 7.2: Special Use Airspace Penetration

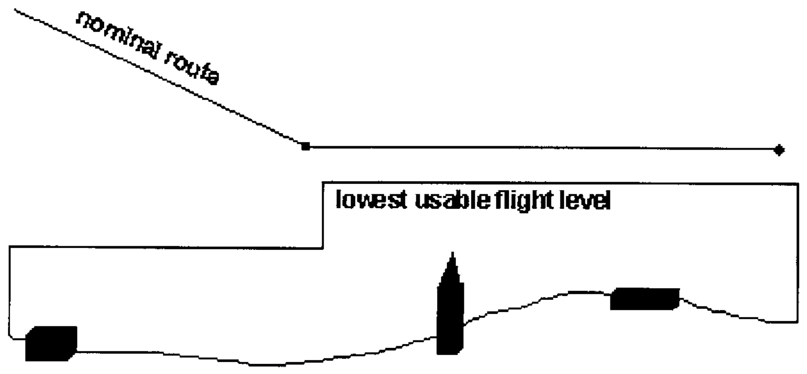


Figure 7.3: Descent Below Lowest Usable Flight Level

MTCD is a planning tool with a typical detection horizon of zero to twenty minutes for aircraft conflicts, twenty to sixty minutes for nominal route overlaps, and zero to sixty minutes for special use airspace penetrations and descents below lowest usable flight level. MTCD is not a conflict alert tool. Conflict alert, with a typical horizon of zero to two minutes is covered by Eurocontrol by a separate function, called Safety Nets. MTCD covers all phases of flight. In arrival and departure phases, different separation criteria apply. MTCD should allow for different separation distances between individual flights that have been sequenced for arrival or departure.

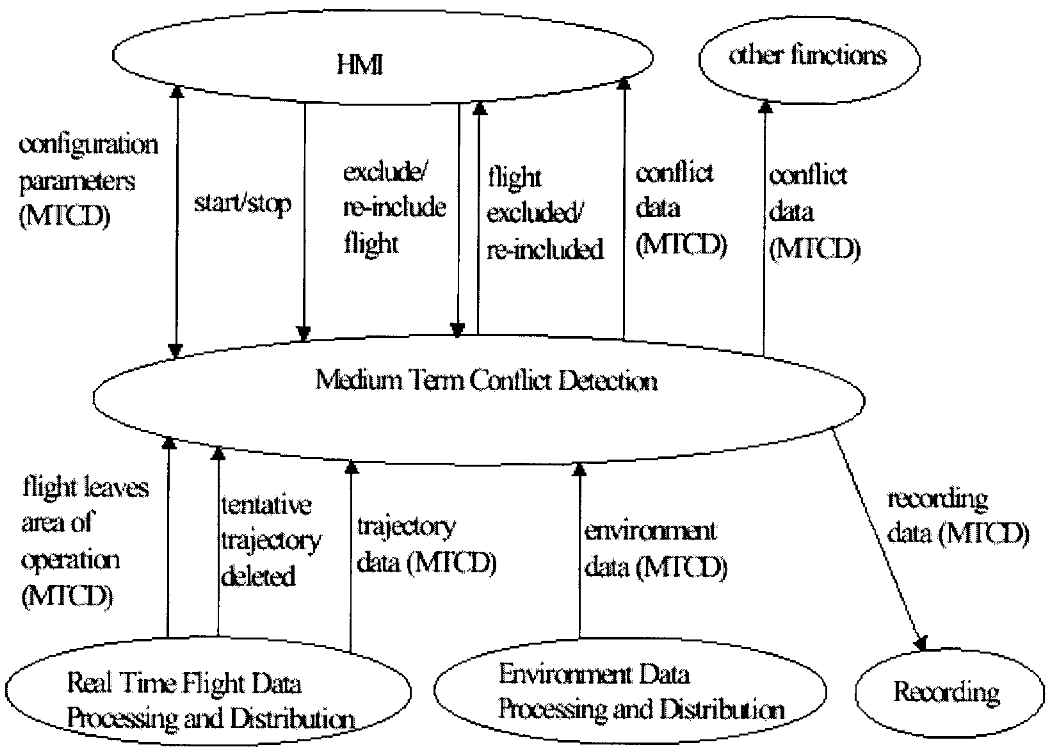

HMI
other functions
configuration parameters (MTCD)
start/stop
exclude/ re-include flight
flight excluded/ re-included
conflict data (MTCD)
conflict data (MTCD)
Medium Term Conflict Detection
flight leaves area of operation (MTCD)
tentative trajectory deleted
trajectory data (MTCD)
environment data (MTCD)
recording data (MTCD)
Real Time Flight Data Processing and Distribution
Environment Data Processing and Distribution
Recording

Figure 7.4: MTCD and its Input/Output Environment

MTCD calculations are based on system trajectories of flights, flight plan data and aircraft data. This data is provided by the Real-Time-Flight Data Processing and Distribution function. Trajectories can be either system trajectories or tentative trajectories. To be able to end existing conflicts, Real-Time Flight Data Processing and Distribution must inform MTCD when a flight leaves the area of operation, or when a tentative trajectory has been deleted. In addition to trajectory data, MTCD requires environment data, which is provided by the Environment Data Processing and Distribution function.

7.2 Modelling Concerns

In principle, MTCD is quite simple. The traffic and its evolution is specified by a set of trajectories so all it needs to do is examine these trajectories in pairs and report whenever such trajectories come too close.

Complications occur because of the:

1. Model uncertainties in aircraft behaviour
2. Introduction of filtering mechanisms so that high traffic situations can be handled

The following sections discuss how these concerns were addressed and also consider how a number of fundamental design choices were made.

7.2.1 Evolutive versus Analytical Approaches

Conflicts can be detected through either analytical or evolutive means. An evolutive algorithm steps through the traffic development at regular intervals (every 10 seconds) and at every snapshot looks for aircraft that are too close to one another. In contrast, the analytical approach computes the relative dynamics of trajectories and determines the precise start and end time of conflicts. The analytical approach is more accurate and in general faster than the evolutive approach. The evolutive approach is simpler and more extensible than the analytical approach. It is extensible in the sense that if the detection of other problems or situations that might occur aside from conflicts needed to be accomplished, then an evolutive approach might be the only way in order to achieve these additional goals.

MTCD uses the analytical approach. It should be noted that the evolutive approach can be used to check results obtained analytically.

7.2.2 Uncertainty Modeling

At a specified future time, the position of an aircraft is not known for sure. The approach used in MTCD is to construct a buffer shape that notionally surrounds an aircraft's future position. The approach is called geometric because, although the buffer sizes may embody uncertainties, their ultimate values are determined through an optimization process that involves controller assessment of the MTCD tool.

The current philosophy is that each trajectory segment has uncertainty information that is derived from the airspace in which the segment is contained. The trajectory segments should not span more than one airspace. The European Air Traffic Control Harmonization Integration Program (EATCHIP) technical program (TP) drafting group has decided that the trajectory will be augmented by the uncertainty information (i.e. x, y, z variances at each point), thereby simplifying MTCD's task. In the approach taken, horizontal and vertical uncertainties are treated separately.

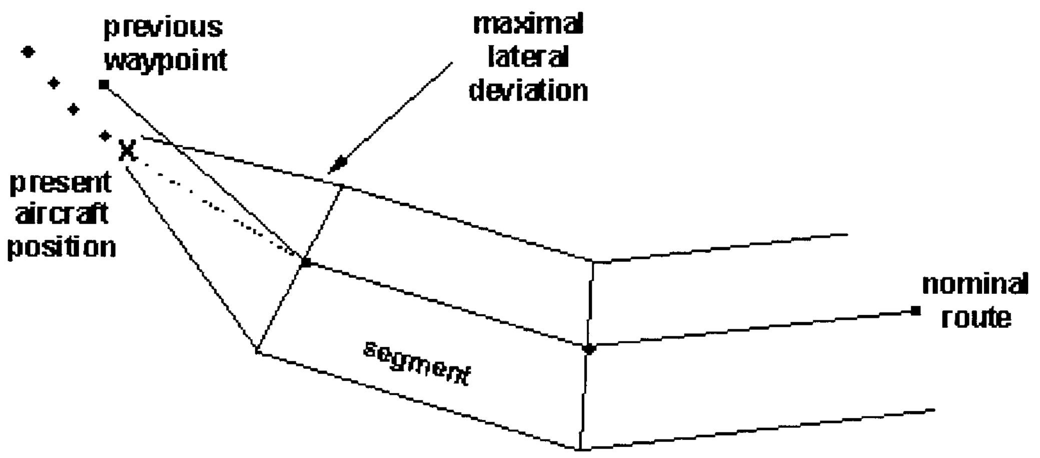


Figure 7.5: Trajectory and Buffer

Horizontal uncertainties are embodied in a buffer shape that is centered on the aircraft's nominal position. The shape is a rectangle aligned to the aircraft's path with the corners rounded by a circle with diameter equal to the long axis of the rectangle. The dimensions of the buffer shape vary with predictive time in a linear fashion for a given trajectory segment. Conflicts occur when such shapes overlap and this aspect can be easily demonstrated to controllers. In some ways an ellipse would have been more elegant as a buffer shape, but a satisfactory well conditioned algorithm that predicts the times of overlaps of expanding ellipses has not been found.

Note that in addition to including aircraft position uncertainty, the buffer shape includes an allowance for half the separation standard. In this way it is only needed to determine when buffers overlap in order to detect a conflict, rather than determining when buffers come within a certain distance (e.g. the separation standard) of each other.

Vertical uncertainty modelling is a fairly simple affair. If a particular altitude separation threshold is infringed, then a vertical conflict occurs. This altitude separation threshold will depend on the respective aircraft's behaviour (whether the craft is level, climbing, descending, above or below 2950 feet etc.), and the uncertainty is directly dependent on that behaviour.

7.2.3 Filters

In terms of computer processing, conflict detection is potentially the most demanding of automated ATC functions. It involves comparison between pairs of trajectories. This effort will increase as the square of the number of flights. Thus, for 200 flights there will be 19 900 comparisons. Determining the exact details of a conflict depends on uncertainty models, which in turn depend on segment containment in airspace volumes. The amount of processing is large. Now, in the 200 flight scenario there may only be 100 contemporary conflicts. Thus, 19800 of the comparison checks would lead to a “no conflict” assessment. Filters constitute a quick test that easily eliminates most of the non-conflicts from consideration.

The filter compares two trajectories for a given time interval to be checked and indicates whether or not a conflict is possible. If a conflict is possible, the procedure indicates a filtered time interval in which any conflict must occur. The filtered time interval is determined by the earliest and latest timeslices in which conflicts may occur, clipped if necessary by the time interval checked. For MTCD it may be necessary to perform conflict checks for up to 60 minutes in advance of the current time, in order to detect nominal route overlaps.

7.3 Continuous Model of the Aircraft

The basis for analysis, computation or simulation of the unsteady motions of an aeroplane is the mathematical model of the vehicle and its subsystems. An aeroplane in flight is a very complicated dynamic system. It consists of an aggregate of elastic bodies so connected that both rigid and elastic relative motions can occur. The external forces that act on an aeroplane are also complicated functions of its shape and its motion. It seems clear that realistic analyses of engineering precision are not likely to be accomplished with a very simple mathematical model.

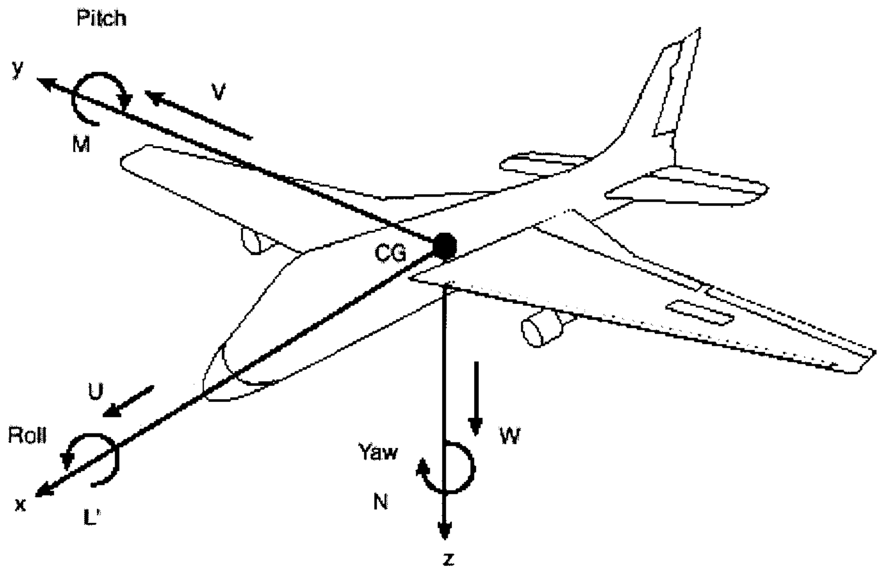


Figure 7.6: Linear and Angular Position and Velocity of Aircraft

To begin formulating the model, first treat the vehicle as a single rigid body with six degrees of freedom. This body is free to move in the atmosphere under the actions of gravity and aerodynamic forces. It is primarily the nature and complexity of the aerodynamic forces that distinguish flight vehicles from other dynamic systems. If the gyroscopic effect of spinning rotors is next factored in, along with a discussion of structural distortion, the complexity of the model increases. If the Earth is treated as being flat and stationary in inertial space, the model is simplified enormously. This assumption is quite acceptable for most aeroplane flights.

In the interest of completeness, the rigid-body equations are derived from first principles. The velocities and accelerations are relative to an inertial frame of reference. The position and orientation of the airplane are given relative to the Earth-fixed axis frame (F^E), and the center of gravity of the aeroplane (c_g) has co-ordinates (x,y,z). The orientation or the aeroplane is given by a series of three consecutive rotations (ψ,θ,ϕ), the *Euler Angles*, whose order is important (Fig 7.6). Any orthogonal axes whose origin is fixed at the center of gravity of the aeroplane are termed *body axes* (F^B). Since most aircraft are very nearly symmetrical, it is usual to assume exact symmetry, and to let C_{xz} be the plane of symmetry. Then, C_x points forward, C_z downward and C_y to the right. In this case, the two products of inertia, I_{xy} and I_{xz} are zero. The directions of C_x and C_z are chosen to coincide with the principal axes of the vehicle, so that the remaining product of inertia I_{zx} vanishes.

Denote the aerodynamic force exerted on the aircraft body in the body-referential axis frame as $A_B = \begin{bmatrix} X & Y & Z \end{bmatrix}$. Let $G_B = \begin{bmatrix} L & M & N \end{bmatrix}$ represent the total aerodynamic moment of the aircraft in the body-referential axis frame. Symbolize the body-referenced linear velocity and angular velocity of the aeroplane as (u, v, w) and (p, q, r). Represent the relative angular momentum of aircraft rotors as $h' = \begin{bmatrix} h'_x & h'_y & h'_z \end{bmatrix}$. The wind speed in the Earth-reference frame is given by $W = \begin{bmatrix} W_x & W_y & W_z \end{bmatrix}$. Indicating the mass of the aeroplane as *m*, the gravitational constant as *g*, and the inertia of the aeroplane as:

$$I = \begin{bmatrix} I_x & -I_{xy} & -I_{xz} \\ -I_{yx} & I_y & -I_{yz} \\ -I_{zx} & -I_{zy} & I_z \end{bmatrix}$$

the equations of motion become [40]:

$$X - mg\sin(\theta) = m(\dot{u}^E + qw^E - rv^E) \tag{7.1}$$

$$Y + mg\cos(\theta)\sin(\phi) = m(\dot{v}^E + ru^E - pw^E) \tag{7.2}$$

$$Z + mg\cos(\theta)\cos(\phi) = m(\dot{w}^E + pv^E - qu^E) \tag{7.3}$$

$$L = I_x\dot{p} - I_{zx}\dot{r} + qr(I_z - I_y) - I_{zx}pq + ph_z' + rh'_y \tag{7.4}$$

$$M = I_y\dot{q} + rp(I_x - I_z) + I_{zx}(p^2 - r^2) + rh'_x - ph'_z \tag{7.5}$$

$$N = I_z\dot{r} - I_{zx}\dot{p} + pq(I_y - I_x) + I_{zx}qr + ph'_y - qh'_x \tag{7.6}$$

$$p = \dot{\phi} - \dot{\psi}\sin(\theta) \tag{7.7}$$

$$q = \dot{\theta}\cos(\phi) + \dot{\psi}\cos(\theta)\sin(\phi) \tag{7.8}$$

$$r = \dot{\psi}\cos(\theta)\cos(\phi) - \dot{\theta}\sin(\phi) \tag{7.9}$$

$$\dot{\phi} = p + q(\sin(\theta) + r\cos(\varphi))\tan(\theta) \tag{7.10}$$

$$\dot{\theta} = q\cos(\phi) - r\sin(\phi) \tag{7.11}$$

$$\dot{\psi} = (q\sin(\phi) + r\cos(\phi))\sec(\theta) \tag{7.12}$$

$$\begin{aligned} \dot{x}_E = {} & u^E\cos(\theta)\cos(\psi) + v^E(\sin(\phi)\sin(\theta)\cos(\psi) - \cos(\varphi)\sin(\psi)) + \\ & w^E(\cos(\phi)\sin(\theta)\sin((\psi) - \sin(\phi)\cos(\psi))) \end{aligned} \tag{7.13}$$

$$\begin{aligned} \dot{y}_E = {} & u^E\cos(\theta)\sin(\psi) + v^E(\sin(\phi)\sin(\theta)\sin(\psi) + \cos(\phi)\cos(\psi)) + \\ & w^E(\cos(\phi)\sin(\theta)\sin(\psi) - \sin(\phi)\cos(\psi)) \end{aligned} \tag{7.14}$$

$$\dot{z}^E = -u^E\sin(\theta) + v^E\sin(\phi)\cos(\theta) + w^E\cos(\phi)\cos(\theta) \tag{7.15}$$

$$u^E = u + W_x \tag{7.16}$$

$$v^E = v + W_y \tag{7.17}$$

$$w^E = w + W_z \tag{7.18}$$

There are several assumptions contained in the above equations. The aeroplane is assumed to be a rigid body, which may have attached to it any number of spinning rotors. The C_{xz} plane is regarded to be a plane of mirror symmetry. The axes of any spinning rotors are assumed to be fixed in direction relative to the body axes, and the rotors have constant angular speed relative to the body axes.

The equations above consist of fifteen coupled nonlinear ordinary differential equations in the independent variable t, and three algebraic equations. It is clear that the aerodynamic forces and moments depend in some manner on three things:

1. The relative motion of the aeroplane with respect to the air
2. The control variables that fix the angles of any movable surfaces
3. The settings of any propulsion controls that determine the thrust vector

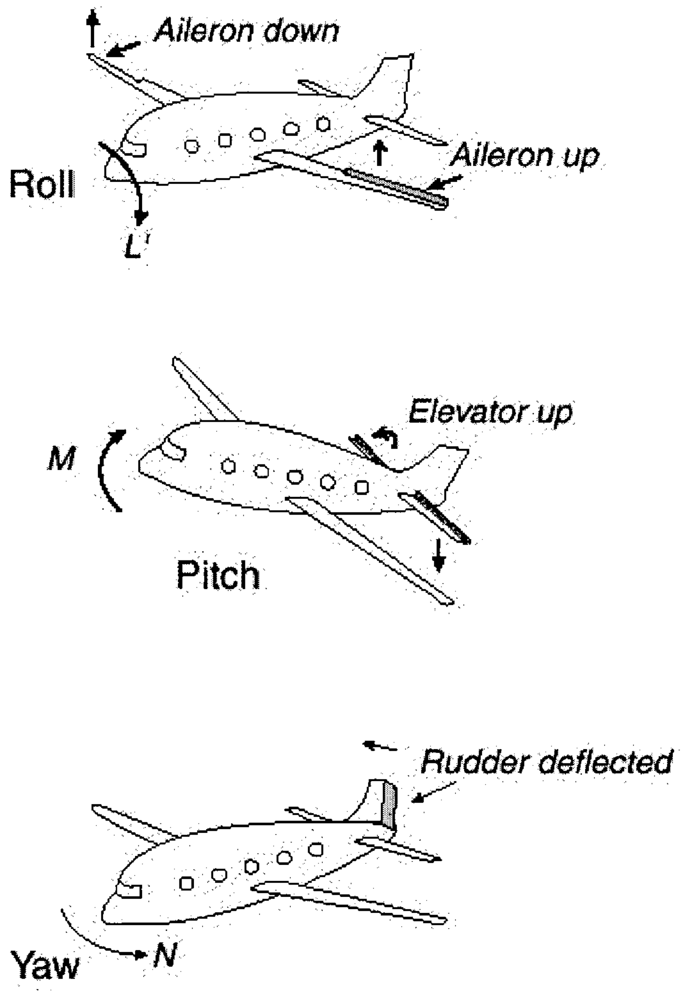


Figure 7.7: Control Angles: Aileron, Elevator and Rudder

Thus it is universally assumed that the six aerodynamic forces and moments are functions of the six linearly and angular velocities (u,v,w,p,q,r) and of a *control* vector $\mathbf{c}$ = $(\delta_a, \delta_e, \delta_r, \delta_t)$, of which the first three are the aileron, elevator and rudder angles, and the last is the throttle control. The control variables, from a mathematical standpoint, are arbitrary functions of time. The wind vector would be supplied by the environmental data.

The true implicit variables of the system become:

1. Centre of Gravity Position: x_E, y_E, z_E
2. Attitude: ψ, θ, ϕ
3. Velocity: u^E, v^E, w^E
4. Angular Velocity: p, q, r

Of the fifteen differential equations (7.1-7.15), three are dependent (7.10-7.12), which leaves twelve independent differential equations. Thus, the number of independent equations equals the number of independent variables, and the system is mathematically complete.

The equations of motion must be linearized in order to fit them into a linear state space description. The equations are linearized by using small-disturbance theory. It is assumed that the motion of the airplane consists of small deviations from a reference condition of steady flight. The reference value of all the variables are denoted by the subscript zero, and the small perturbations are denoted by the prefix Δ. When the reference value is zero, the Δ is omitted. All disturbance quantities are assumed to be small, and their squares and products are negligible compared to first order quantities. The reference flight condition is assumed to be symmetric and with no angular velocity, so $v_0 = p_0 = q_0 = r_0 = \phi_0 = 0$. If the stability axes are selected as the body axes for the aeroplane, then $w_0 = 0$, with u_0 being the reference flight speed, and θ_0 the reference angle of climb. Furthermore, the effects of spinning rotors are deemed negligible and the wind velocity is assumed to be zero.

Using the mathematical notation for differentials where:

$$L_\alpha = \left.\frac{\partial L}{\partial \alpha}\right|_{\alpha = 0} \tag{7.19}$$

and realizing that in symmetrical flight, the side force Y, the rolling moment L, and the yawing moment N will be zero, it can be concluded that v, p, r, ϕ, ψ, y_E are all zero. The derivatives of the asymmetrical or lateral forces and moments Y,L,N with respect to the symmetric or longitudinal motion variables u,w,q are zero. The derivatives of the symmetric forces and moments with respect to the asymmetric motion variables may be neglected in all calculations. All derivatives with respect to rates of change of motion variables may be neglected, except for $Z_{\dot{w}}$ and $M_{\dot{w}}$. The derivative X_q is negligibly small, and the density of the atmosphere is assumed not to vary with altitude. The linear forces and moments become:

$$\Delta X = X_u \Delta u + X_w w + \Delta X_c \tag{7.20}$$

$$\Delta Y = Y_v v + Y_p p + Y_r r + \Delta Y_c \tag{7.21}$$

$$\Delta Z = Z_u \Delta u + Z_w w + Z_{\dot{w}} w + Z_q q + \Delta Z_c \tag{7.22}$$

$$\Delta L = L_v v + L_p p + L_r r + \Delta L_c \tag{7.23}$$

$$\Delta M = M_u \Delta u + M_w w + M_{\dot{w}} \dot{w} + M_q q + \Delta M_c \tag{7.24}$$

$$\Delta N = N_v v + N_p p + N_r r + \Delta N_c \tag{7.25}$$

In the preceding equations, the terms on the right with the subscript c are control forces and moments that result from the control vector **c**. Using equations (7.20-5) and small disturbance theory, the linear equations of motion become:

$$\begin{bmatrix}\Delta\dot{u}\\ \dot{w}\\ \dot{q}\\ \Delta\dot{\theta}\end{bmatrix} = \begin{bmatrix} \frac{X_u}{m} & \frac{X_w}{m} & 0 & -g\cos(\theta_0) \\ \frac{Z_u}{m-Z_{\dot{w}}} & \frac{Z_w}{m-Z_{\dot{w}}} & \frac{Z_q+mu_0}{m-Z_{\dot{w}}} & \frac{-mg\sin(\theta_0)}{m-Z_{\dot{w}}} \\ \frac{1}{I_y}\left[M_u+\frac{M_{\dot{w}}Z_u}{m-Z_{\dot{w}}}\right] & \frac{1}{I_y}\left[M_w+\frac{M_{\dot{w}}Z_w}{m-Z_{\dot{w}}}\right] & \frac{1}{I_y}\left[M_q+\frac{M_{\dot{w}}(Z_q+mu_0)}{m-Z_{\dot{w}}}\right] & \frac{-M_{\dot{w}}mg\sin(\theta_0)}{I_y(m-Z_{\dot{w}})} \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix}\Delta u\\ w\\ q\\ \Delta\theta\end{bmatrix} + \begin{bmatrix} \frac{\Delta X_c}{m} \\ \frac{\Delta Z_c}{m-Z_{\dot{w}}} \\ \frac{\Delta M_c}{I_y}+\frac{M_{\dot{w}}}{I_y}+\frac{\Delta Z_c}{m-Z_{\dot{w}}} \\ 0 \end{bmatrix} \tag{7.26}$$

$$\Delta\dot{x}_E = \Delta u\cos(\theta_0)+w\sin(\theta_0)-u_0\Delta\theta\sin(\theta_0) \tag{7.27}$$

$$\Delta\dot{z}_E = -\Delta u\sin(\theta_0)+w\cos(\theta_0)-u_0\Delta\theta\cos(\theta_0) \tag{7.28}$$

$$\begin{bmatrix}\dot{v}\\ \dot{p}\\ \dot{r}\\ \dot{\phi}\end{bmatrix} = \begin{bmatrix} \frac{Y_v}{m} & \frac{Y_p}{m} & \left(\frac{Y_r}{m}-u_0\right) & g\cos(\theta) \\ \left(\frac{L_v}{I_x'}+I_{zx}'N_v\right) & \left(\frac{L_p}{I_x'}+I_{zx}'N_p\right) & \left(\frac{L_r}{I_x'}+I_{zx}'N_r\right) & 0 \\ \left(I_{zx}'L_v+\frac{N_v}{I_z'}\right) & \left(I_{zx}'L_p+\frac{N_p}{I_z'}\right) & \left(I_{zx}'L_r+\frac{N_r}{I_z'}\right) & 0 \\ 0 & 1 & \tan(\theta_0) & 0 \end{bmatrix} \begin{bmatrix}v\\ p\\ r\\ \phi\end{bmatrix} + \begin{bmatrix} \frac{\Delta Y_c}{m} \\ \frac{\Delta L_c}{I_x'}+I_{zx}'N_c \\ I_{zx}'\Delta L_c+\frac{\Delta N_c}{I_z'} \\ 0 \end{bmatrix} \tag{7.29}$$

$$\dot{\psi} = r\sec(\theta_0) \tag{7.30}$$

$$\Delta\dot{y}_E = u_0\psi\cos(\theta_0)+v \tag{7.31}$$

with $I_x' = \frac{(I_xI_z-I_{zx}^2)}{I_z}$, $I_z' = \frac{(I_xI_z-I_{zx}^2)}{I_x}$ and $I_z' = \frac{I_{zx}}{(I_xI_z-I_{xz}^2)}$.

As a consequence of the simplifying assumptions made in their derivation, the preceding equations can be divided into two categories: longitudinal (7.26-8) and lateral (7.29-31). Suppose that ϕ, $v, p, r, \Delta Y_c, \Delta L_c$ and ΔN_c are zero. Then equations (7.29-31) are satisfied. The remaining equations (7.26-8) form a complete set of six homogeneous variables $\Delta u, w, q, \Delta\theta, \Delta x_E, \Delta y_E$. Modes of motion are possible in which only these variables differ from zero. Such motions are called longitudinal or symmetric motion, and the corresponding equations and variables are named longitudinal. Conversely, if the longitudinal variables are set to zero, the remaining six equations (7.26-8) form a complete set for the determination of the variables ϕ, ψ, v, p, r, y_E. These variables are known as the lateral variables. The existence of pure longitudinal motions depends on two assumptions: the existence of a plane of symmetry and the absence of rotor gyroscopic effects. The existence of pure lateral motion depends on three conditions: the linearization of the equations, the absence of rotor gyroscopic effects and the ability to neglect all aerodynamic cross coupling.

The equations (7.26-7.31) are in state space format $\dot{x} = Ax + Bc$. The state vector for the longitudinal and lateral systems are, respectively:

$$x = \begin{bmatrix}\Delta u & w & q & r\end{bmatrix}^T \tag{7.32}$$

$$x = \begin{bmatrix}v & p & r & \phi\end{bmatrix}^T \tag{7.33}$$

and the A matrices can be read from equations (7.26) and (7.29). The control vector can also be broken down into longitudinal and lateral modes respectively:

$$c = \begin{bmatrix}\delta_e & \delta_t\end{bmatrix}^T \tag{7.34}$$

$$c = \begin{bmatrix}\delta_a & \delta_e\end{bmatrix} \tag{7.35}$$

where the elevator angle and throttle are used for longitudinal control, and the aileron and rudder angles are used for lateral control. To calculate the B matrices, control theory transfer function response techniques must be employed. For the case of the longitudinal response, from equation (7.26):

$$B\Delta c = \begin{bmatrix} \dfrac{\Delta X_c}{m} \\ \dfrac{\Delta Z_c}{m - Z_{\dot{w}}} \\ \dfrac{\Delta M_c}{I_y} + \dfrac{M_{\dot{w}}}{I_y} + \dfrac{\Delta Z_c}{m - Z_{\dot{w}}} \\ 0 \end{bmatrix} \tag{7.36}$$

Assume that the incremental aerodynamic forces and moment that result from control actuation can be given by a set of control derivatives in the form:

$$\begin{bmatrix} \Delta X_c \\ \Delta Z_c \\ \Delta M_c \end{bmatrix} = \begin{bmatrix} X_{\delta_e} & X_{\delta_t} \\ Z_{\delta_e} & Z_{\delta_t} \\ M_{\delta_e} & M_{\delta_t} \end{bmatrix} \begin{bmatrix} \Delta\delta_e \\ \Delta\delta_t \end{bmatrix} \tag{7.37}$$

From equation (7.37) and (7.36) the B matrix becomes:

$$B = \begin{bmatrix} \dfrac{X_{\delta_e}}{m} & \dfrac{X_{\delta_t}}{m} \\ \dfrac{Z_{\delta_e}}{(m - Z_{\dot{w}})} & \dfrac{Z_{\delta_t}}{(m - Z_{\dot{w}})} \\ \dfrac{M_{\delta_e}}{I_y} + \dfrac{M_{\dot{w}} Z_{\delta_e}}{I_y(m - Z_{\dot{w}})} & \dfrac{M_{\delta_t}}{I_y} + \dfrac{M_{\dot{w}} Z_{\delta_t}}{I_y(m - Z_{\dot{w}})} \\ 0 & 0 \end{bmatrix} \tag{7.38}$$

for the longitudinal equations.

Similarly, for the lateral equations, assume the aerodynamics associated with the two lateral controls are given by the set of control derivatives:

$$\begin{bmatrix} \Delta Y_c \\ \Delta L_c \\ \Delta N_c \end{bmatrix} = \begin{bmatrix} Y_{\delta_a} & Y_{\delta_r} \\ L_{\delta_a} & L_{\delta_r} \\ N_{\delta_a} & N_{\delta_r} \end{bmatrix} \begin{bmatrix} \delta_a \\ \delta_r \end{bmatrix} \tag{7.39}$$

so B is given by:

$$B = \begin{bmatrix} \frac{Y_{\delta_a}}{m} & \frac{Y_{\delta_r}}{m} \\ \frac{L_{\delta_a}}{I_x'} + I_{zx}'N_{\delta_a} & \frac{L_{\delta_r}}{I_x'} + I_{zx}'N_{\delta_r} \\ I_{zx}'L_{\delta_a} + \frac{N_{\delta_a}}{I_z'} & I_{zx}'L_{\delta_r} + \frac{N_{\delta_r}}{I_z'} \\ 0 & 0 \end{bmatrix} \tag{7.40}$$

The complete state space description of the linearized model of aircraft dynamics has now been elucidated. These are the continuous equations used in the SpecTRM-RL model to calculate the potential trajectories of the aircraft between data updates. At the instant of data update, the values of the variables are the values that are read in from the Real Time Flight Data Processing and Distribution system. These values are then taken as the nominal values for the equations above. During the following time interval, before the next update, the above equations are propagated forward to determine the trajectory of the aircraft. At the next update, the trajectory is rectified, and the process begins again.

7.4 Hybrid Model of Medium Term Conflict Detection Algorithm

The dynamic model can be used in conjunction with the flight data, to calculate the continuous trajectory of the aeroplane given the correct initial states and inputs. With these calculated aeroplane trajectories, the algorithm determines whether or not the trajectories will conflict (either with the trajectory of another aeroplane or with a lowest flight level or restricted airspace). MTCD also factors in uncertainty in the modeling technique and flight data, thereby creating buffers around the trajectories, as well as providing a numbering and updating convention to account for conflicts and trajectories, enabling their creation and removal when the plane enters and exits the appropriate sector. An overview of the SpecTRM-RL model of MTCD is seen below (Fig. 7.8).

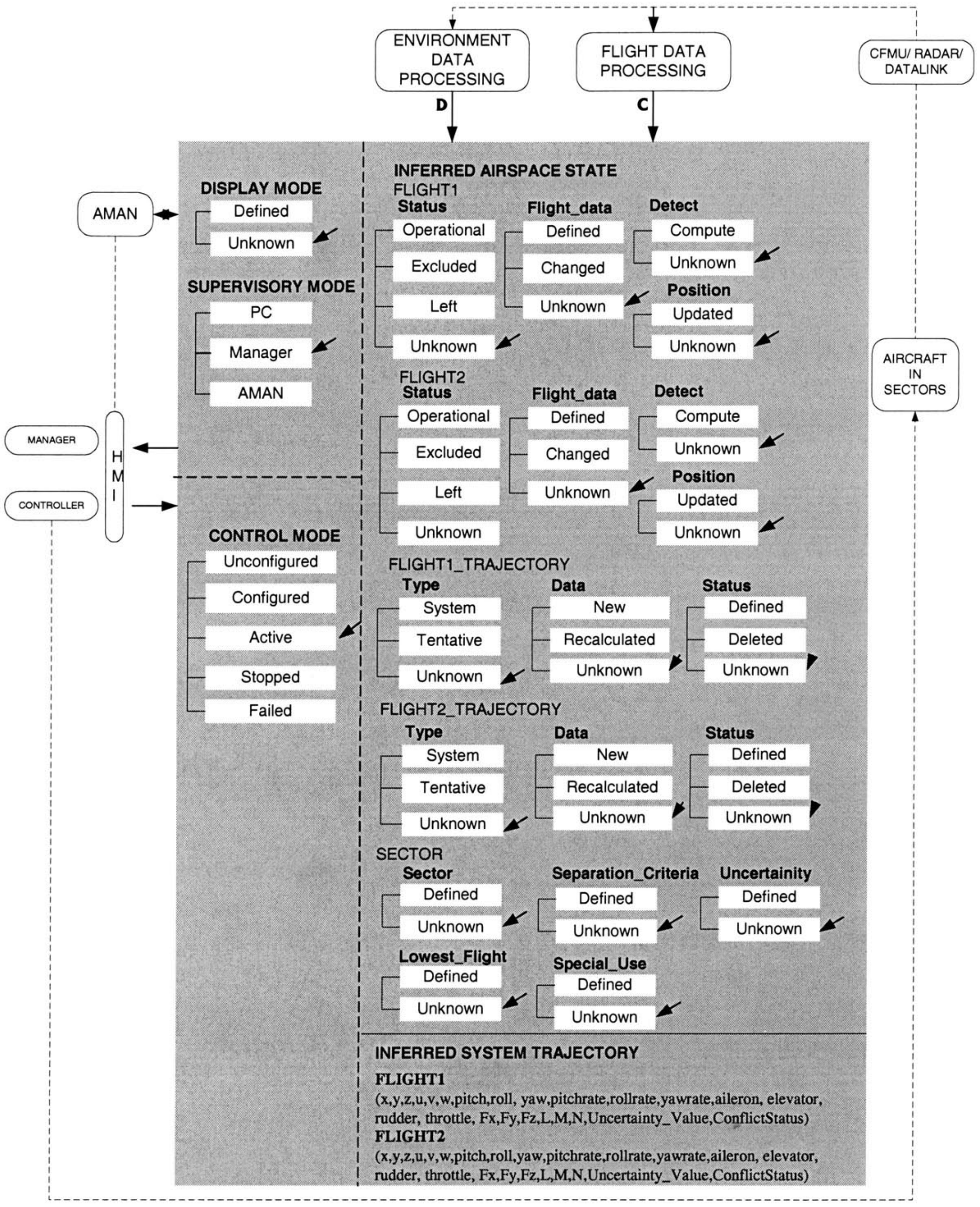
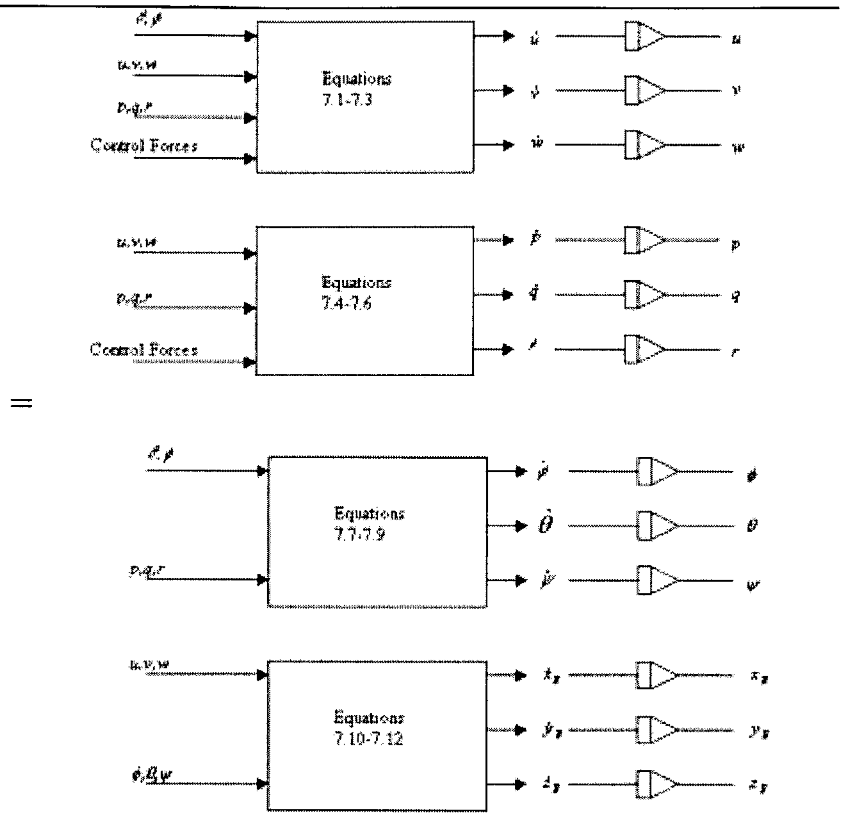


Figure 7.8: SpecTRM-RL Model of MTCD

The SpecTRM-RL model of MTCD incorporates the notion of hybrid trajectories by creating a new component to the generic model form, entitled Inferred System Trajectory.



Value(aileron, elevator, rudder, throttle, Fx, Fy, Fz, L, M, N, Uncertainty_Value,Conflict_Status)

Control Mode = Active	T
Flight1 Status = Unknown	F
Flight1 Flight_Data = Unknown	F
Flight1 Position = Updated	T
Flight1 Detect = Compute	T
Flight1_Trajectory Type = Unknown	F
Flight1_Trajectory Data = Recalculated	T
Flight1_Trajectory Status = Defined	T
Sector = Defined	T
Uncertainty = Defined	T
Time-Time(Last_Update)<UpdateRate	T

Figure 7.9: Trajectory Element in Hybrid SpecTRM-RL

The inferred trajectories are a model of the continuous processes or plants including state variables and measured or manipulated process variables as reflected by the inputs and outputs to the controller. In the case of MTCD, each aircraft has its own unique trajectory.

The continuous input variables included in the trajectory are the position (x,y,z) and velocity (u,v,w) of the center of mass of the aircraft, angular position (pitch, roll, yaw) and velocity (pitch rate, roll rate, yaw rate) of the aircraft. The control variables included are the aileron, elevator and rudder angles, and throttle control. The forces and moments acting on the aeroplane are also included in the trajectory, as is the uncertainty in the modeled data and the conflict status (Unknown, Buffer Violation, Descent Below Altitude, Special Airspace Violation, Nominal Route Overlap, No Conflict). Evaluated over time these variables act to form the continuous trajectory of the system for each aircraft. Combining all of these aircraft trajectories yields the system trajectory of the entire airspace sector that MTCD is operating upon.

The logical specification for determining if a conflict has occurred can be expressed explicitly in the disjoint normal form of the truth tables in SpecTRM-RL. The four different types of conflicts are handled separately, with a different function performing the identification of each type of conflict. For example, in order to determine if an aircraft will infringe upon a restricted airspace, MTCD employs the Medium Term Area Proximity Warning (MTAPW) system. The MTAPW sub-algorithm of MTCD determines all potential special airspace violations for all identified restricted airspaces and all included trajectories. The logical flow of the algorithm is detailed below in Figs 7.10-11. The conditions in the two diagrams can easily be converted into logical predicates involving only the calculated trajectories and the specified special airspaces. Similar logical diagrams can be drawn for the conflict situations of buffer overlap (aircraft conflict) and descent below lowest usable flight level. The full SpecTRM-RL description is contained in Appendix C.

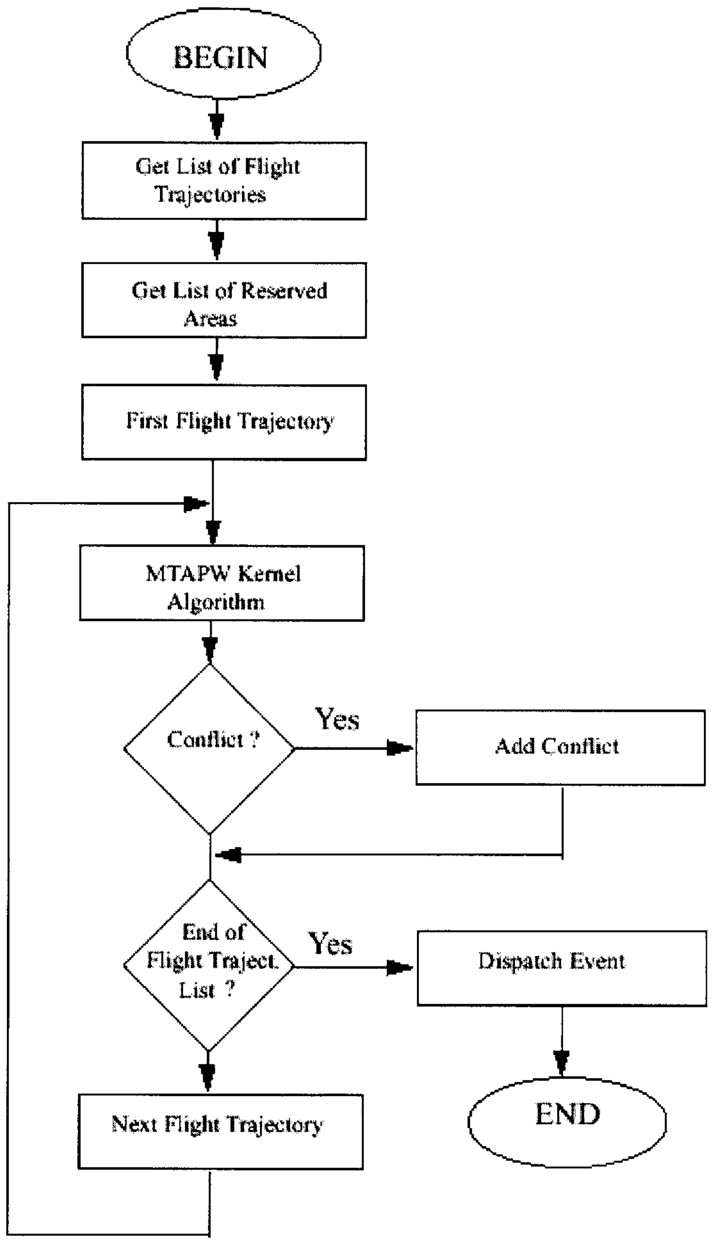


Figure 7.10: MTAPW Shell

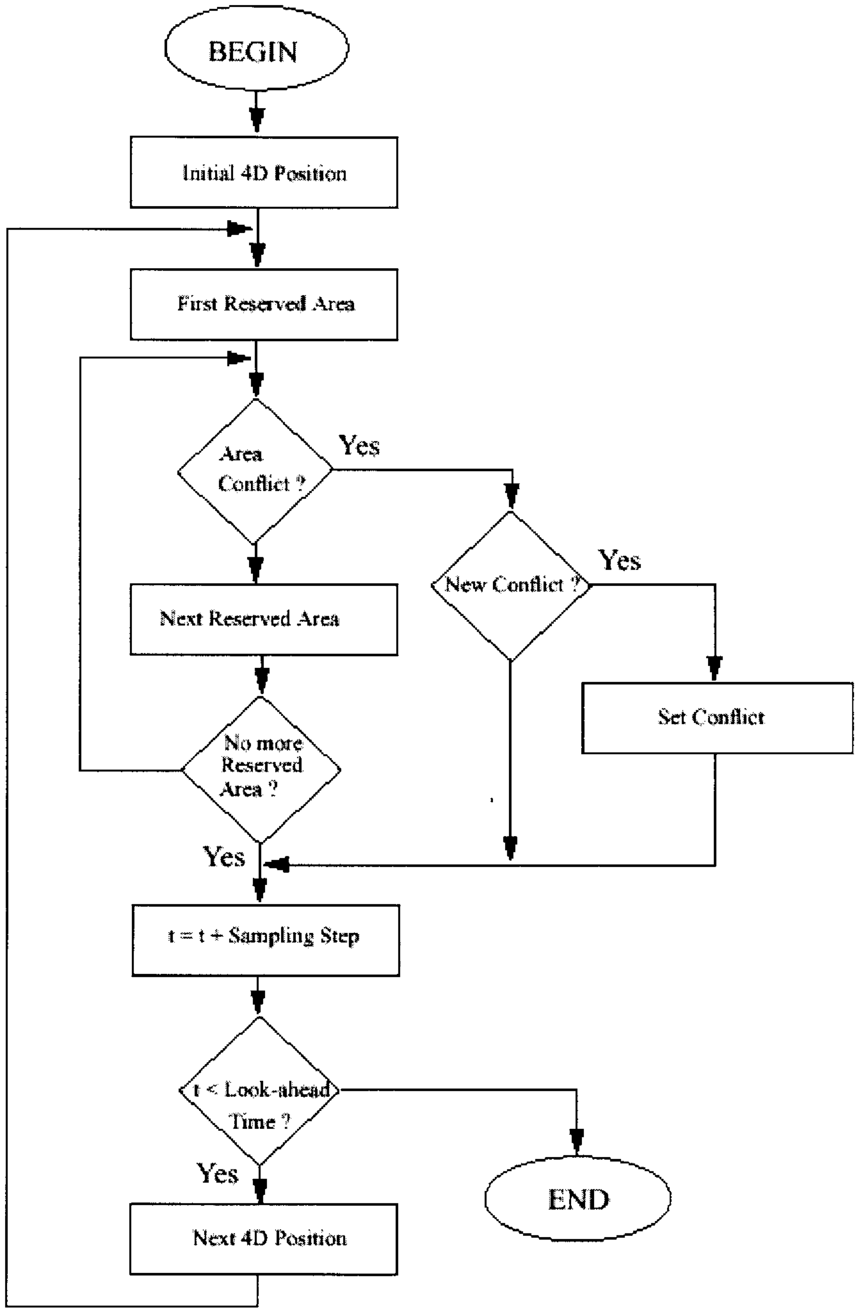


Figure 7.11: MTAPW Kernel

The 4-D position described above refers to the spatial and temporal coordinates of the aircraft. The sampling step refers to the update rate of the algorithm, which is dependent on the number of aircraft in the sector. The look-ahead time refers to the 20-60 minute time horizon on conflict detection for special airspace violations.

7.5 Hazard Automaton Reduction Algorithm Applied to MTCD

Consider the case of an aircraft conflict between a plane and a restricted airspace, which can clearly be defined as the hazardous state that occurs when the trajectory buffer of the plane overlaps the buffer zone of the special use airspace. MTCD checks for this condition by modeling the separation distances and uncertainties in a buffer about the nominal trajectory of the plane, and then tests to see if the plane buffer overlaps the specified special airspace buffer. Since no reasonable algorithm for calculating the overlap in varying elliptical buffers is known, MTCD uses first the box test, and then the circle test, to determine if a conflict has arisen. If both the box and circle tests are positive, then a conflict is detected by MTCD, and the controller must take appropriate action.

In the present model of the two-aircraft system, more information is necessary in order to apply the Hazard Automaton Reduction Algorithm. Begin by recalling that the Hazard Automaton Reduction Algorithm only considers the risk of the states of the system. Create an internal variable for MTCD called Risk that is a function of the state of the aircraft trajectories in the system, as well as various inputs. The variable Risk is a discrete variable, and can possess only three values {Low,High,Hazard}. Next, consider augmenting the set of discrete transitions δ. Each time the Risk variable changes value, a discrete transition is needed. If there is already a discrete transition enabled at this point, then no further work need be done. If there is no discrete transition, a "dummy" transition must be inserted. An internal "dummy" trigger predicate P_t' is created, which is only used in conjunction with the internal "dummy" transition. This "dummy" trigger enables the "dummy" transition, causing the system state to change by a discrete action (not simply through a continuous evolution). The final state of the trajectory prior to the "dummy" transition is identical to the first state of the trajectory following the "dummy" transition except for the value of the

Risk variable. This ensures that the Risk value of the system can only change via discrete transitions, and not through a continuous evolution of variables.

The Risk variable is used to group multiple aircraft trajectory configurations into families related by their apparent risk and to artificially create "dummy" discrete transitions which act to coarsely discretize the system. The Hazard Automaton Reduction Algorithm can be applied to this partially discretized system. It remains only to find the critical configurations of MTCD corresponding to the hazard of loss of separation between an aircraft and a restricted airspace. Recall that a hazard occurs when the buffer of the plane infringes upon the buffer of the airspace. The critical state would occur at the last point where it is possible for the plane to recover and assume a trajectory that would not infringe upon the buffer of the airspace. If the MTCD algorithm has not detected that a conflict is imminent past this critical point, then the conflict detection scheme is flawed.

Starting with the hazardous point in the trajectory where the two buffers overlap but no conflict is detected, the state can be propagated backwards until the point at which discrete input-enabling is restored. This point can be found because all of the aircraft trajectories can be uniquely described using their 18 differential equations that define their state within the system at any given instant. For n aircraft, the backwards reachable space of the differential equations can be determined by examining the entire reachable range of the matrix corresponding to the A and B matrices in equations (7.26,7.29,7.38,7.40) defined by the aircraft differential equations. The initial hazardous condition is encoded as x(0), and the range of the backwards trajectories is calculated by multiplying out each column in the reachability matrix using the initial conditions as the nominal flight state for the A and B matrices in the differential equations. After the entire reachability matrix has been computed, the state space description for $\mathfrak{R}_n$ is converted into its equivalent reachable region in the map of hybrid trajectories. The boundaries of the region can then be checked to see if they are input enabled. If not, then all enabled discrete transitions are taken, and a new set of nominal flight values is used to calculate the new A and B matrices. The reachability set of the new dynamic equations is calculated, and the region is checked to see if input enabling is restored. If not, another round of discrete transitions must be taken and the process begins again. Thus, the full reachability set of the model need never be generated, the graph need only be generated until it intersects the critical region where input enabling has been restored.

The point at which input-enabling is restored occurs at the last possible moment when a controller command to the aircraft in question would be able to avert a potential conflict. The situation is critical, but not hazardous, if a potential conflict has not been detected and action can still be taken by the controller in order to avert the conflict. The controller still has enough time to plan a resolution to the conflict if it were detected. Therefore, the conflict is still avoidable even though it has not yet been detected. However, once the controller is no longer able to plan in order to avoid an upcoming conflict, the situation has become hazardous due to the lack of detection of the potential conflict.

The system designers can use the information from the hazard analysis to redesign the system to be safer or to assist in making trade-offs between alternative designs. For example, to mitigate the identified hazard of a potential conflict going unidentified by MTCD, the controller might be given a warning every time a plane's trajectory is about to become inflexible as well as an advisory about how to route other planes accordingly. This design effectively eliminates the hazardous paths out of the identified critical states. It has drawbacks, however, because many of these warnings would be unnecessary. Resolving such trade-offs and perhaps generating better solutions, is the job of the design engineer.

CHAPTER 8

It is unwise to be too sure of one's own wisdom. It is healthy to be reminded that the strongest might weaken and the wisest might err.

Mohindas Gandhi (1869 - 1948)

Conclusions

8.1 State Explosion and Scalability

The Hazard Automaton Reduction Algorithm can reduce the depth of searches needed to be employed upon complex state spaces in order to eliminate and control hazards. Instead of testing to see if a hazardous state is reachable from an initial state, the Hazard Automaton Reduction Algorithm enables the user to search back just sufficiently far in the reachability graph to determine a state from which the hazard can be successfully controlled or eliminated. Thus, the state explosion problem can be avoided in many instances, if the critical state is close enough to the hazardous state.

Techniques from modern state space controls theory are particularly suited for determining the reachability of a state. It becomes necessary to perform only a finite number of matrix multiplications to determine whether or not a state is reachable from another state. A state machine model can be converted into a controls state space formulation and a state transition matrix can be created. In addition, a simple rank calculation can be performed at the start of the process to determine whether or not the entire state space is reachable at all, thereby eliminating from consideration all hazards that are not realistic combinations of state values, and mitigating the state explosion problem.

The Hazard Automaton Reduction Algorithm allows hazards to be controlled or eliminated without generating the entire reachability graph. This vastly reduces the state explosion problem, making the problem of hazard elimination more tractable. When employed with state space control theory reachability results, the problem reduces to performing a finite number of matrix multiplications. Hence, an approach based on the Hazard Automaton Reduction Algorithm coupled with control theory results is a promising method by which to control or eliminate hazards in very large complex systems.

For smaller models, the Hazard Automaton Reduction Algorithm could be coupled with model checking techniques to remove hazards. However, when the hazards become farther removed from their critical states, as sometimes happens in large models, the beneficial effects of Hazard Automaton Reduction Algorithm are greatly diminished and the state explosion problem emerges once again. This is not the case if the state space controls technique is employed, as all of the cost goes into the creation of the large transition matrix. Each successive step backwards involves only one n by n matrix manipulation, so computational cost does not increase drastically with the distance between the hazard and the critical state.

The benefits are even greater for hybrid models. For hybrid models, the Hazard Automaton Reduction Algorithm could not be coupled with model checking techniques because there are no termination guarantees in backwards reachability using model checking.

8.2 Hybrid Systems and Hazard Elimination

At present, many linear hybrid model checkers possess the same approach to verification [25]. The main function the model checker performs is the verification of safety properties: given an initial region and an unsafe region, the model checker verifies whether the system starting with the initial region ends up within the unsafe region. The verification is done by forward or backward reachability analysis. The verification procedure is not necessarily decidable, i.e., the computation could go on with no guarantee of termination. The lack of termination can result from the model checker using the "Fixed Point" iteration method in the following fashion [3,25,91]:

> For a state assertion ϕ, let Pre(ϕ) be a state assertion that is true for a state q if and only if there exists an f-state q' such that (q, q') is either a jump or a flow of the system. If the state assertion ϕ_1=Pre(*unsafe*) can be computed, then all states that will enter the unsafe region by trajectories of length 1 are characterized. The backwards reachability analysis is carried out by successively applying the Pre operator to the current reachable region, starting with the unsafe region. The computation stops whenever the reachable region intersects the initial region, or there is no new reachable region discovered.

It is possible to continue indefinitely using this approach, as it is very difficult to quantify the point at which no new reachable region is discovered. Determining the equality of the present reachability set to the previous reachability set involves a computation that can be influenced by the slightest numerical imprecision. Thus, there can be no guarantee that the terminating condition will ever be reached.

Reachability analysis is not necessarily decidable even for linear hybrid automata. In fact, it is not decidable except for some special cases. It is decidable for timed automata (systems which have only clocks that run with an identical rate) or simple multi-rate systems. And even for the problems that are decidable, most of them are PSPACE-hard [3,25]. Hence, carrying out a full backwards reachability analysis in a hybrid automaton is very difficult. The notion of propagating back a limited number of steps until a critical region is reached is very appealing for hybrid automata because the full reachability of the system need never be explored.

A formulation in the SpecTRM-RL language for modeling hybrid systems was presented. The notion of backwards reachability was outlined, in terms of how it could be used to create constraints that would avoid hazardous states in a process control system. A control theory technique was postulated to calculate the backwards reachable region of a continuous set of differential equations. These techniques were illustrated by applying them to a Medium Term Conflict Detection algorithm under development for aircraft collision avoidance. The hazardous condition of a missed detection was investigated, and a critical region was found. Constraints were then suggested in order to remove the hazardous situation from the forward reachability flow of the critical region. The issue of non-termination was never encountered due to the close proximity of the critical region to the hazardous region.

The MTCD model can scale up, simply by adding more aircraft trajectory inputs to the MTCD interface. Of course, the more aircraft added, the more computationally complex the algorithm becomes, eventually swamping out the effects of the Hazard Automaton Reduction Algorithm due to the increased coupling of the continuous state variables. Additionally, the Hazard Automaton Reduction Algorithm is conservative, which leads to unreachable hazards being designed out of the system. Future efforts must focus on being able to manage the computational complexity of the hybrid version of the Hazard Automaton Reduction Algorithm.

8.3 A Final Word

The coupled method of the Hazard Automaton Reduction Algorithm and state space control reachability guarantees a solution to the control or elimination of hazards in a purely discrete system design. This strong statement can be made because after a finite number of matrix multiplications, either the critical state is reached and the hazard can be controlled or eliminated, or it can be concluded that no such state exists, and therefore the entire design is hazardous. Thus, after the design has been analyzed using this method, either all the hazards are controlled or the entire system must be redesigned, as there is no possible solution given the current design.

For hybrid systems, the statement is not as strong. If the critical state cannot be found, it is entirely possible that it exists on a boundary or region that has yet to be explored. However, this difficulty in finding the critical region is most likely indicative of a sub-optimal design. This statement can be made because it is usually best to have control over a flow of low risk behaviour as near as possible to hazardous behaviour, so that the path leading to hazardous behaviour can be diverted as late as possible. This would translate, in some sense, to using the minimum magnitude of input in order to divert the control from the path toward the hazardous behaviour onto the path of low risk behaviour. Thus, system redesign should be re-considered if a critical state is not found corresponding to any hazard.

The secret to creativity is knowing how to hide your sources.

Albert Einstein (1879 - 1955)

References

[1] Air Force Space Division, *System Safety Handbook for the Acquisition Manager*, SDP 127-1, January 12, 1987.

[2] Alur, R, Courcoubetis, C., Halbwachs, N., Henzinger, T.A., Ho, P. H., Nicollin, X., Olivero, A., Sifakis, J., Youvine, S., The Algorithmic Analysis of Hybrid Systems, 1994.

[3] Alur, R., Costas, Courcoubetis, Henzinger, Thomas A. and Ho, Pei-Hsin. Hybrid automata: an algorithmic approach to the specification and verification of hybrid systems. In Hybrid Systems I, Lecture Notes in Computer Science 736, Springer-Verlag, 1993, pp. 209-229.

[4] Alur, R., Henzinger, Thomas A. and Ho, Pei-Hsin. Automatic symbolic verification of embedded systems. IEEE Transactions on Software Engineering 22:181-201, 1996. A preliminary version appeared in the Proceedings of the 14th Annual IEEE Real-time Systems Symposium (RTSS 1993), pp. 2-11.

[5] Anderson, John D., Introduction to Flight, McGraw-Hill Inc., 1989.

[6] Behrmann, G., Larsen, Kim G., Pearson, Justin, Weise, Carsten and Yi, Wang. Efficient Timed Reachability Analysis Using Clock Difference Diagrams. Computer Aided Verification 1999.

[7] Bensalem, S., Bouajjani, A., Loiseaux, C. and Sifakis, J. Property Preserving Simulations. Workshop on Computer Aided Verification. Fourth International Workshop. CAV'92. Proceedings LNCS 663. Springer 1992. pp. 260-273.

[8] Bjorner, Nikolaj, Browne, Anca and Manna, Zoher. Automatic Generation of Invariants and Intermediate Assertions. TCS special issue dedicated to CP '95. June 3 1996.

[9] Bjorner, Nikolaj S., Manna, Zoher, and Sipma, Henny B. Deductive Verification of Real-time Systems Using STeP. Submitted to Elsevier Science.

[10] Bjorner, N. Browne, Anca, Chang, Eddie, Colon, Michael, Kapur, Arjun... STeP: Deductive-Algorithmic Verification of Reactive and Real-time Systems. International Conference on Computer Aided Verification, pp. 415-418. vol. 1102of Lecture Notes in Computer Science, Springer-Verlag, July 1996.

[11] Bjorner, N. S., Browne, A., and Manna, Z. Automatic Generation of Invariants and Intermediate Assertions. Theoretical Computer Science 173, 1 (Feb. 1997), 49-87. International vonference on Principles and Practice of Constraint Programming. Vol

976. Springer-Verlag. 1995.

[12] Bjorner, N. S., Manna, Z. Sipma, H. B., and Uribe, T. E. Deductive Verification of Real-Time Systems Using STeP. 4th International AMAST Workshop on Real-Time Systems. Vol 1231. Spriger-Verlag. May 1997.

[13] Bjorner, N. S., Browne, Anca, Colon, Michael, Finkbeiner, Bernd, Manna, Zoher, Pichora, Marc, Sipma, Henny B., Uribe, Tomas E. STeP: The Stanford Temporal Prover Educational Release Version 1.4 User's Manual. July 1998.

[14] Bose, S., and Fisher, A.L., Automatic Verification of Synchronous Circuits Using Symbolic Logic Simulation and Temporal Logic. In *Proceedings of the IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, 1989.

[15] Branicky, Michael, Studies in Hybrid Systems, Sc.D. Thesis, Massachusetts Institute of Technology, June 1995

[16] Branicky, Michael, Dolginova, Ekaterina and Lynch, Nancy. A Toolbox for Proving and Maintaining Hybrid Specifications. In Panos J. Antsaklis, editor, Hybrid Systems IV (HS'96, Cornell University, Ithaca, NY, October 12-16, 1996), volume 1273 of Lecture Notes in Computer Science. Springer-Verlag 1996.

[17] Browne, M.C., Clarke, E.M. and Dill, D.L. Automatic Circuit Verification using Temporal Logic: Two New Examples. Formal Aspects of VLSI Design. Elsevier 1986.

[18] Bryant, R.E., Graph-based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers* C-35(8):677-691.

[19] Bryant, R.E. and Seger, C-J., Formal Verification of Digial Circuits using Symbolic Ternary System Models., *Workshop on Computer-Aided Verification, 2nd International Conference, CAV'90. Proceedings, LNCS* 531, Springer 1990, pp. 33-43.

[20] Burch, J.R. et al, Sequential Circuit Verification Using Symbolic Model Checking. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pp. 46-51. IEEE, 1990.

[21] Burstall, R.M., Program Proving as Hand Simulation with a Little Induction, In *IFIP Conbress 74*, pp. 308-312. North Holland, 1972.

[22] Clarke, E.M., Emerson, E.A., and Sistla, A.P., Automatic Verification of Finite-State Concurrent Systems Using Temporal Lobic Specifications. *ACM Transactions on Programming Languages and Systems.*, 8(2): pp224-263.

[24] Clarke, E.M., Filkorn, T. and Jha, S. Exploiting Symmetry in Temporal Logic Model Checking. Proceedings of the 5th Workshop on Computer Aided Verification. June/July 1993. pp. 450-462.

[25] Clarke, E. M., Grumberg, O. and Peled, D. Model Checking. MIT Press 2001.

[26]] Clarke, E.M., Grumberg, O. et al., Parametrized Networks. In *Proceedings of the 6th International Conference on Concurrency Theory, LNCS* 962. Springer 1995. pp. 395-407.

[27] Clarke, E.M., Long, D.E. and McMillan, K.L. A Language for Compositional Specification and Verification of Finite State Hardware Controllers. Proceedings of the 9th International Symposium of Computer Hardware Description Languages and Their Applications. North Holland 1989. pp. 281-295.

[28] Cleaveland, R.W., Parrow, J., and Steffen, B., The concurrency Workbench. *Proceedings of the 1989 International Workshop on Automatic Verification Methods for Finite State Systems, LNCS* 407. Springer, 1989., pp. 24-37.

[29] Cross, A, Fault Trees and Event Trees. In A.E. Green, Editor, *High Risk Safety Technology*, pp. 49-65, John Wiley and Sons, New York, 1982.

[30] Coudert, O., Berthet, C., and Madres, J.C., Verification of Synchronous Sequential Machines Based on Symbolic Execution, *Proceedings of the 1989 International Workshop on Automatic Verification Methods for Finite State Systems, LNCS* 407, Springer, 1989, pp. 365-373.

[31] Coudert, O., Madres, J.C., and Berthet, C., Verifying Temporal Properties of Sequential Machines Without Building their State Diagrams, *Workshop on Computer-Aided Verification. 2nd International Conference, CAV'90. Proceedings, LNCS* 531, Springer 1990, pp. 23-32.

[32] Cox, S.J. and N.R.S. Tait, *Reliability, Safety & Risk Management: An Integrated Approach*, Butterworth-Heinemann Ltd., Oxford, England, 1991, pp. 1-24.

[33] Dahleh, M. Introduction to Linear State Space Control Theory. Course Notes. Massachusetts Institute of Technolgy. Fall 1999.

[34] Dang, T. Maler, O., Reachability Analysis via Face Lifting, in T.A. Henzinger and S. Sastry (Eds), Hybrid Systems: Computation and Control, LNCS 1386, 96-109, Springer, 1998.

[35] Daniel, Peter, SAFECOMP '97, Proceedings of the 16th International Conference on Computer Safety, Reliability and Security, York, 7-10 September 1997, York, Springer-Verlag, Berlin.

[36] Daniels, B.K., Achieving Safety and Reliability with Computer Systems, Proceedings of the Safety and Reliability Society Symposium, 1987, Altrincham, Manchester, UK, 11-12 November 1987. Elsevier Applied Science, London.

[37] Daniels, J.T. and Holden, P.L., Quantification of Risk In *Loss Prevention and Safety Promotion in the Process Industries*, pp. 1-12, Queen's College, Cambridge, UK. Pergamon Press, September 1983.

[38] Duke, B. W., Program Manager's Handbook for System Safety and Military Standard 882B. *Hazard Prevention*, pp. 15-21, March/April 1986.

[39] Emerson, E.A., *Branching Time Temporal Logic and the Design of Correct Concurrent Programs*. Ph.D Thesis, Harvard University, 1981.

[40] Etkin, Bernard and L.D. Reid, Dynamics of Flight: Stability and Control, John Wiley and Sons, 1996.

[41] European Air Traffic Control Harmonisation and Integration Programme, Functional

Specification for EATCHIP Phase III Medium Term Conflict Detection, 1997.

[42] Fortune, Joyce and Geoff Peters, *Learning From Failure—The Systems Approach*, John Wiley and Sons, Chichester, England, 1995, pp. 1-37.

[43] Garland, Stephen and Nancy A. Lynch and Mandana Vaziri. IOA: A Language for Specifying, Programming and Validating Distributed Systems. User and Reference Manual. Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, December 1997.

[44] Godefroid, P., Using Partial Orders to Improve Automatic Verification Methods., In *Proceedings of the 2nd Workshop on Computer Verification, LNCS* 531, pp.176-185. Springer 1990.

[45] Godefroid, P. and Pirottin, D. Refining Dependencies Improves Partial Order Verification Methods. In Proceedings of the 5th Conference on Computer Aided Verification. LNCS 697. Springer 1993. pp. 438-449.

[46] Gritzalis, Dimitris, Reliability, Quality and Safety of Software-Intensive Systems, International Conference on Reliability, Quality and Safety of Software-Intensive Systems (ENCRESS '97), 29-30 May 1997, Athens, Chapman & Hall, 1997.

[47] Grumberg, O. and Long, D.E., Model Checking and Modular Verification. *ACM Transactions on Programming Languages and Systems* 16:843-872.

[48] Hammer, W. *Handbook of System and Product Safety*, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1972.

[49] Hammer, W. *Product Safety Management and Engineering*, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1980.

[50] Harel, Z., and Kurshan, R.P., COSPAN, *Proceedings of the 1996 Workshop on Computer Aided Verification, LNCS* 1102, Springer, 1996, pp. 423-427.

[51] Heimdahl, M.P.E and Nancy Leveson. Completeness and Consistency Analysis of State-Based Requirements, Published in IEEE Transactions on Software Engineering (May 1996).

[52] Henzinger, T.A., Pei-Hsin Ho, and Howard Wong-Toi. A user guide to HyTech. Proceedings of the First International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 1995), Lecture Notes in Computer Science 1019, Springer-Verlag, 1995, pp. 41-71.

[53] Henzinger, T.A., Pei-Hsin Ho, and Howard Wong-Toi, HyTech: the next generation. Proceedings of the 16th Annual IEEE Real-time Systems Symposium (RTSS 1995), pp. 56-65.

[54] Henzinger, T.A., Peter W. Kopke, Anuj Puri, and Pravin Varaiya. What's decidable about hybrid automata? Journal of Computer and System Sciences 57:94—124, 1998. A preliminary version appeared in the Proceedings of the 27th Annual ACM Symposium on Theory of Computing (STOC 1995), pp. 373-382.

[55] Henzinger. T.A., The theory of hybrid automata. Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS 1996), pp. 278-292.

[56] Henzinger, T.A., Pei-Hsin Ho, and Howard Wong-Toi. HyTech: a model checker for hybrid systems. Software Tools for Technology Transfer 1:110-122, 1997. A preliminary version appeared in the Proceedings of the Ninth International Conference on Computer-aided Verification (CAV 1997), Lecture Notes in Computer Science 1254, Springer-Verlag, 1997, pp. 460-463.

[57] Hope, S, et. al., Methodologies for Hazard Analysis and Risk Assessment in the Petroleum Refining and Storage Industry. Hazard Prevention, Pp24-32, July/August 1983.

[58] Hughes, G.E. and Creswell., M.J., Intruduction to Modal Logic. Methuen, 1977.

[59] Jaffe, M.S., Leveson, N.G., Heimdahl, M.P.E, Melhart, B.E., "Software Requirements Analysis for Real-Time Process-Control Systems", IEEE Transactions on Software Engineering, Vol 17, No. 3, March 1991, pp. 241-257.

[60] Katz, S, and Peled, D., An efficient Verification Method for Parallel and Distributed Programs. In *Workshop on Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, LNCS* 354, pp. 489-507. Springer 1998.

[61] Knight, J. and N.G. Leveson., An Experimental Evaluation of the Assumption of Independence in Multi-Version Programming, IEEE Transactions on Software Engineering, Vol. SE-12, No. 1, January 1986, pp. 96-109.

[62] Kroger, F., LAR: A Logic of Algorithmic Reasoning. *Acta Informatica*, 8(3).

[63] Kurshan, R.P. et al. Static Partial Order Reduction. In *Tools and Algorithms for the Construction and Analysis of Systems, LNCS* 1384, pp.345-357. Springer, 1998.

[64] Kurzhanski, A.B. and Varaiya,P. Ellipsoidal techniques for reachability analysis. In N. Lynch and B. Krogh, editors, Hybrid Systems: Computation and Control (HSCC'00), LNCS 1790, pages 203--213. Springer-Verlag, 2000.

[65] Larsen, K.G., Paul Pettersson and Wang Yi., UPPAAL in a Nutshell. In Springer International Journal of Software Tools for Technology Transfer 1(1+2), 1997.

[66] Larsen, K.G., Paul Pettersson and Wang Yi.Compositional and Symbolic Model-Checking of Real-Time Systems. In Proceedings of the 16th IEEE Real-Time Systems Symposium, Pisa, Italy, 5-7 December, 1995.

[67] Larsen, K.G., Fredrik Larsson, Paul Pettersson and Wang Yi.Efficient Verification of Real-Time Systems: Compact Data Structure and State-Space Reduction. In Proceedings of the 18th IEEE Real-Time Systems Symposium, pages 14-24. San Francisco, California, USA, 3-5 December 1997.

[68] Lamport, Leslie, Specifying Concurrent Systems with TLA+, Leslie Lamport, Current Draft, 7 December, 2000.

[69] Lamport, Leslie, The Temporal Logic of Actions, ACM Transactions on Programming Languages and Systems, April 30, 1994.

[70] Lehtela, M., Computer-Aided Failure Mode and Effect Analysis of Electronic Circuits. *Microelectronic Reliability*, 30(4):761-773, 1990.

[71] Leveson, N.G. Safeware: System Safety and Computers. Addison-Wesley 1995.

[72] Leveson, N.G., Heimdahl, M.P.E. and Reese, J.D. Designing Specifications Languages for Process Control Systems. Presented at SIGSOFT FOSE '99, Foundations of Software Engineering. Toulouse. Sept. 1999.

[73] Leveson, N.G. and Stolzy, J.L. Safety Analysis Using Petri Nets. IEEE Transactions on Software Engineering. SE-13(3):386-397. March 1987.

[74] Leveson, N.G., Completeness in Formal Specification Language Design for Process Control Systems, Proceeedings of Formal Methods in Software Practice Conference, August 2000.

[75] Leveson, N.G., Heimdahl, M.P.E., Hildreth, H., and Reese, J.D., Requirements Specification for Process-Control Systems. Published in IEEE Transactions on Software Engineering (Sept. 1994)

[76] Leveson, N.G., L. Alfaro, C. Alvarado, M. Brown, E.B. Hunt, M. Jaffe, S. Joslyn, D. Pinnel, J. Reese, J. Samarziya, S. Sandys, A. Shaw, Z. Zabinsky., Demonstration of a Safety Analysis on a Complex System. Presented at the Software Engineering Laboratory Workshop, NASA Goddard, December 1997.

[77] Leveson, N.G., Maxime de Villepin, Mirna Daouk, John Bellingham, Jayakanth Srinivasan, Natasha Neogi, and Ed Bachelder (MIT) and Nadine Pilon and Geraldine Flynn (Eurocontrol). A Safety and Human-Centered Approach to Developing New Air Traffic Management Tools. Air Traffic Management 2001, Albuquerque NM, December 2001.

[78] Lichtenstein, O., and Pnueli, A., Checking that Finite State Concurrent Programs Satisfy their Linear Specification. In *Proceedings of the 12th Annual ACM Symposium on Principles of Programming Language*, pp. 97-107. ACM 1985.

[79] Livadas, Carlos and Nancy A. Lynch. Formal Verification of Safety-Critical Hybrid Systems. Proceedings of 1st International Workshop, Hybrid Systems: Computation and Control (HSCC'98, Berkeley, CA, April 1998), volume 1386 of Lecture Notes in Computer Science. Springer-Verlag 1998.

[80] Lutz, R., Targeting safety -related errors during software requirements analysis. In Proceedings of the First ACM SIGSOFT Symposium on the Foundations of Software Engineering, 1993.

[81] Lygeros, John and Nancy Lynch. On the Formal Verification of the TCAS Conflict Resolution Algorithms. Proceedings of the 36th IEEE Conference on Decision and Control, San Diego, CA, December 1997.

[82] Lygeros, John, Tomlin, Claire and Sastry, Shankar, "Controllers for reachability specifications for hybrid systems", Automatica, 1999.

[83] Lynch, N., Vaandrager, F., "Forward and Backward Simulations II. Timing Based Systems", Information and Compuation, Vol 128, No. 1, Academic Press, July 10, 1996.

[84] Lynch, Nancy, Roberto Segala, and Frits Vaandrager. Hybrid I/O Automata Revisited. In Maria Domenica Di Benedetto and Alberto Sangiovanni-Vincentelli, editors Hybrid Systems: Computation and Control. Fourth International Workshop

(HSCC'01, Rome, Italy, March 2001, volume 2034 of Lecture Notes in Computer Science, pages 403-417, 2001. Springer-Verlag.

[85] Lynch, Nancy, Roberto Segala, Frits Vaandrager, and H. B. Weinberg. Hybrid I/O automata. Submitted for journal publication.

[86] Lynch. N, A Three-Level Analysis of a Simple Acceleration Maneuver, with Uncertainties. Proceedings of the Third AMAST Workshop on Real-Time Systems, pages 1-22, Salt Lake City, Utah, March 1996.

[87] Lynch. N, Modelling and verification of automated transit systems, using timed automata, invariants and simulations. In R. Alur, T. Henzinger, and E. Sontag, editors, Hybrid Systems III: Verification and Control (DIMACS/SYCON Workshop on Verification and Control of Hybrid Systems, New Brunswick, New Jersey, October 1995), volume 1066 of Lecture Notes in Computer Science, pages 449-463. Springer-Verlag 1996.

[88] Lynch, N, Roberto Segala, Frits Vaandrager, and H. B. Weinberg. Hybrid I/O Automata. In R. Alur, T. Henzinger, and E. Sontag, editors, Hybrid Systems III: Verification and Control (DIMACS/SYCON Workshop on Verification and Control of Hybrid Systems, New Brunswick, New Jersey, October 1995), volume 1066 of Lecture Notes in Computer Science, pages 496-510. Springer-Verlag 1996.

[89] Malasky, Sol W., *System Safety: Planning/Engineering/Management*, Hayden Book Company Inc., Rochelle Park, New Jersey, USA., 1974, pp. 1-36.

[90] Manna, Z. and Pnueli, A., Temporal Verifications of Reactive Systems-Safety. Springer 1995.

[91] Manna, Z. and Sipma, H., Deductive Verification of Hybrid Systems Using STeP. Appeared in Hybrid Systems: Computation and Control, International Workshop, LCNS 1386, Springer-Verlag, Berkely, April 1998.

[92] Manna, Z., and Pnueli, A. Clocked Transition Systems. In Proc. Of the International Logic and Software Engineering Workshop, Beijeng, China. Aug. 1995.

[93] Manna, Z., and Pnueli, A. Clocked Transition Systems. Presented at the Workshop on Verification and Control of Hybrid Systems, New Brunswick, NJ. Oct. 1995.

[94] McCormick, Norman, J., Reliability and Risk Analysis. Academic Press, New York, 1981.

[95] McMillan, K., Using Unfolding to Avoid the State Explosion Problem in the Verification of Asynchronous Circuits, *Workshop on Computer-Aided Verification. Fourth International WOrkshop, CAV'92. Proceedings, LNCS* 663, Springer 1992., pp. 164-177.

[96] Miller, S. Modelling Software Requirements for Embedded Systems. Altitude Switch Specification. Rockwell Collins.

[97] Modugno, F., Leveson, N. G., Reese, J. D., Partridge, K. and Sandys, S.D. Integrated Safety Analysis of Requirements Specifications. IEEE 1997. pp. 148-159.

[98] Nielsen, D., Use of Cause-Consequence Charts in Practical Systems Analysis. In

Theoretical and Applied Aspects of System Reliability and Safety Assessment, pp. 849-880, SIAM, Philadelphia, 1975.

[99] Overman, W.T., *Verification of Concurrent Systems: Function and Timing.* PhD Thesis, University of California at Los Angeles, 1981.

[100] Peled, D., Combining Partial Order Reductions with On-the-Fly Model Checking, *Proceedings of the 1994 Workshop on Computer-Aided Verification Methods for Finite State Systems, LNCS* 818. Springer 1994, pp. 377-390.

[101] Pixley, C., Introduction to a Computation THeory and Implementation of Sequential Hardware Equivalence., *Workshop on Computer Aided Verification, 2nd International Conference, CAV'90. Proceedings, LNCS* 513, Springer 1990., pp. 54-64.

[102] Pnueli, A., The Temporal Logic of Programs, In *18th IEEE Symposium on Foundation on Computer Science*, pp. 46-57. IEEE Computer Society Press, 1977.

[103] Quielle, J.P. and Sifakis, J., Specification and Verification of concurrent Systems in CESAR. In *Proceedings of the 5th International Symposium on Programming*, pp.337-350.

[104] Rajan, S., Shankar, N., and Srivas, M.K. An Integration of Model Checking with Automated Proof Checking. Proceedings of the 1995 Workshop on Computer Aided Verification. LNCS 939. Springer 1995. pp. 84-97.

[105] Reese, John Damon, *Software Deviation Analysis*, PhD thesis, University of California, Irvine, CA, 1995.

[106] Rogers, William P., Introduction to System Safety Engineering, John Wiley and Sons, 1971.

[107] Shaw, Roger, Safety and Reliability of Software Based Systems, Twelfth Annual Centre for Software Reliability Workshop, Springer-Verlang London Limited, London, England, 1997.

[108] Sontag, Eduardo D., Mathematical Control Theory: Deterministic Finite Dimensional Systems.Second Edition, Springer, New York, 1998.

[109] Suokas, Juoko, The Role of Management in Accident Prevention, In *First International Congress on Industrial Engineering and Management*, Paris, June 11-13, 1986.

[110] Suokas, Juoko, The Role of Safety Analysis in Accident Prevention. *Accident Analysis and Prevention*, 20(1):67-85, 1988.

[111] Terry, J.G., *Engineering System Safety*, Mechanical Engineering Publications Ltd., Alden Press, Oxford, England, 1991, pp. 1-21.

[112] Tomlin, Claire, Lygeros, John and Sastry, Shankar, A Game Theoretic Approach to Controller Design for Hybrid Systems, *Proceedings of the IEEE, Volume 88, Number 7,* July 2000.

[113] Tomlin, Claire, Mitchell, Ian, and Ghosh, Ronojoy.,Safety Verification of Conflict Resolution Maneuvers, *IEEE Transactions on Intelligent Transportation Systems,*

Volume 2, Number 2, June 2001.

[114] Tomlin, Claire, Pappas, George J. and Sastry, Shankar., Conflict Resolution for Air Traffic Management: A Study in Multi-Agent Hybrid Systems, *IEEE Transactions on Automatic Control, Volume 43, Number 4,* April 1998.

[115] Thomson, J.R., *Engineering Safety Assessment: An Introduction*, Longman Scientific and Technical, Harlow, England, 1987, pp. 1-8.

[116] Valmari, A, A Stubborn Attack on State Explosion. In *Proceedings of the 16th International Colloquium on Automata, Languages and Programming, LNCS* 372. pp. 761-772. Springer, 1989.

[117] Vardi, M.Y., and Wolper, P., An Automata-Theoretic Approach to Automatic Program Verification. In *Proceedings of the 1st Annual Symposium on Logic in Computer Science.* IEEE Computer Society Press, 1986, pp.332-344.

[118] Weinberg, H.B. and Nancy Lynch. Correctness of Vehicle Control Systems - A Case Study. Proceedings of the 17th IEEE Real-Time Systems Symposium, pages 62-72, Washington, D. C., December, 1996.

[119] Yi, Wang, Paul Pettersson and Mats Daniels.Automatic Verification of Real-Time Communicating Systems by Constraint Solving. In Proceedings of the 7th International Conference on Formal Description Techniques, Berne, Switzerland, 4-7 October, 1994.

[120] Young, Thomas A., Mars Program Independent Assessment Team Report, March 14, 2000.

Appendix A

...the source of all great mathematics is the special case, the concrete example. It is frequent in mathematics that every instance of a concept of seemingly great generality is in essence the same as a small and concrete special case.

Paul R. Halmos, 1985

Level 3 SpecTRM-RL Model of Altitude Switch

State Value

DOI-Status

Obsolescence: 2 seconds

Exception-Handling: Goes into unknown state

Description:

Comments: There is nothing in the requirements that says what to do if a power-off message is sent and no status message is received from the DOI within 2 seconds. I decided it was safest to have this indicate a possible fault so the watchdog will time out and light the fault indicator lamp in the cockpit.

References:

Appears in: DOI-Power-On, Watchdog-Strobe

DEFINITION

= On

DOI-status-signal = On	T

= Off

DOI-status-signal = Off	T

= Unknown

Powerup	T		
Controls.Reset = T		T	
DOI-status-signal = obsolete			T

= Fault-Detected

Time >= (Time sent DOI-Power-On Message) + 2 seconds	T	T
DOI-status-signal = Off	T	
Time > Time received DOI-status-signal + 2 seconds		T

Column 1: Sent power on message but DOI did not turn on
Column 2: Sent power on message but never got feedback

State Value

Altitude

Obsolescence: 2 seconds

Exception-Handling: Because the altitude-status-signals change to obsolete after 2 seconds, altitude will change to Unknown if all input signals are lost for 2 seconds.

Description:

Comments:

References:

Appears in: DOI-Power-On

DEFINITION

= Unknown

Powerup	T		
Controls.Reset		T	
Analog-ALT = Unknown			T
Dig-Alt1 = Unknown			T
Dig-Alt2 = Unknown			T

= Below-threshold

Analog-Valid-and-Below	T		
Dig1-Valid-and-Below		T	
Dig2-Valid-and-Below			T

= At-or-above-threshold

Analog-Valid-and-Above	T	T	T	F	T	F	F
Dig1-Valid-and-Above	T	T	F	T	F	T	F
Dig2-Valid-and-Above	T	F	T	T	F	F	T

= Cannot-be-determined

Analog-Alt = Invalid	T
Dig-Alt1 = Invalid	T
Dig-Alt2 = Invalid	T

State Value

Analog-Alt

Obsolescence: 2 seconds

Exception-Handling: Will change to unknown when analog-alt-signal becomes obsolete (more than 2 seconds elapse since last message from Analog Altimeter)

Description:

Comments:

References:

Appears in: Altitude

DEFINITION

= Valid

Analog-Alt-Status = Valid	T

= Invalid

Analog-Alt-Status = Invalid	T

= Unknown

Analog-Alt-Status = Obsolete	T		
Powerup		T	
Controls.Reset = T			T

State Value

Dig-Alt1

Obsolescence: 2 seconds

Exception-Handling: Will change to unknown when DA1-Status-Signal becomes obsolete (more than two seconds elapse since last mesage from Digital Altimeter 1).

Description:

Comments:

References:

Appears in: Altitude

DEFINITION

= Valid

DA1-Status-Signal = Norm	T

= Invalid

DA1-Status-Signal = {Fail, NCD, Test}	T

= Unknown

DA1-Status-Signal = Obsolete	T		
Powerup		T	
Controls.Reset = T			T

State Value

Dig-Alt2

Obsolescence: 2 seconds

Exception-Handling: Will change to unknown when DA2-Status-Signal becomes obsolete (more than two seconds elapse since last mesage from Digital Altimeter 2).

Description:

Comments:

References:

Appears in: Altitude

DEFINITION

= Valid

DA2-Status-Signal = Norm	T

= Invalid

DA2-Status-Signal = {Fail, NCD, Test}	T

= Unknown

DA2-Status-Signal = Obsolete	T		
Powerup		T	
Controls.Reset = T			T

Input Value

DOI-Status-Signal

Source: DOI

Type: Enumerated

Possible Values (Expected Range): {On, Off}

Exception-Handling:

Arrival Rate (Load): ??

Min-Time-Between-Inputs:
Max-Time-Between-Inputs:

Obsolescence: 2 seconds

Exception-Handling: Assumes value Obsolete

Description:

Comments:

References:

Appears in: DOI-status

DEFINITION

= FIELD (Status in DOI-Status-Message)

Receive DOI-Status-Message FROM DOI	T

= PREV (DOI-Status-Signal)

Receive DOI-Status-Message FROM DOI	F
Time <= Time (DOI-Status-Message arrived) + 2 seconds	T

= Obsolete

Receive DOI-Status-Message FROM DOI	F	
Time > Time (DOI-Status-Message arrived) + 2 seconds	T	
Powerup		T

Input Value

Analog-Alt-Status

Source: Analog Altimeter

Type: Enumerated

Possible Values (Expected Range): {Invalid, Valid}

Exception-Handling:

Arrival Rate (Load): ??

Min-Time-Between-Inputs:
Max-Time-Between-Inputs:

Obsolescence: 2 seconds

Exception-Handling: Assumes value Obsolete

Description:

Comments:

References:

Appears in: Analog-Alt

DEFINITION

= FIELD (Status in Analog-Alt-Message)

Receive Analog-Alt-Message FROM Analog-Altimeter	T

= PREV (Analog-Alt-Status)

Receive Analog-Alt-Message FROM Analog-Altimeter	F
Time <= Time (Analog-Alt-Message arrived) + 2 seconds	T

= Obsolete

Receive Analog-Alt-Message FROM Analog-Altimeter	F	
Time > Time (Analog-Alt-Message arrived) + 2 seconds	T	
Startup		T

Input Value

Analog-Alt-Signal

Source: Analog Altimeter

Type: Enumerated

Possible Values (Expected Range): {Above, Below}

Exception-Handling:

Arrival Rate (Load): ??

Min-Time-Between-Inputs:
Max-Time-Between-Inputs:

Obsolescence: 2 seconds

Exception-Handling: Assumes value Obsolete

Description:

Comments:

References:

Appears in: Altitude

DEFINITION

= FIELD (Altitude in Analog-Alt-Message)

Receive Analog-Alt-Message FROM Analog-Altimeter	T

= PREV (Analog-Alt-Signal)

Receive Analog-Alt-Message FROM Analog-Altimeter	F
Time <= Time (Analog-Alt-Message arrived) + 2 seconds	T

= Obsolete

Receive Analog-Alt-Message FROM Analog-Altimeter	F	
Time > Time (Analog-Alt-Message arrived) + 2 seconds	T	
Powerup		T

Input Value

DA1-Status-Signal

Source: Digital Altimeter 1

Type: Enumerated

Possible Values (Expected Range): {Fail, NCD, Test, Norm}

Exception-Handling:

Arrival Rate (Load): ??

Min-Time-Between-Inputs:
Max-Time-Between-Inputs:

Obsolescence:2 seconds

Exception-Handling: Assumes value Obsolete

Description:

Comments: Four possible values can be sent signifying Failure Warning, No Computed Data, Functional Test, and Normal Operation.

References:

Appears in: Dig-Alt1

DEFINITION

= FIELD (Status in DA1-Message)

Receive DA1-Message FROM Digital-Altimeter-1	T

= PREV (DA1-Status-Signal)

Receive DA1-Message FROM Digital-Altimeter-1	F
Time <= Time (DA1-Message arrived) + 2 seconds	T

= Obsolete

Receive DA1-Message FROM Digital-Altimeter-1	F	
Time > Time (DA1-Message arrived) + 2 seconds	T	
Powerup		T

Input Value

DA1-Alt-Signal

Source: Digital Altimeter 1

Type: integer

Possible Values (Expected Range): -20..2500

Exception-Handling: Values below -20 are treated as -20 and values above 2500 as 2500

Units: ??

Granularity: ??

Arrival Rate (Load): ??

Min-Time-Between-Inputs:
Max-Time-Between-Inputs:

Obsolescence: 2 seconds

Exception-Handling:

Description:

Comments:

References:

Appears in: Altitude

DEFINITION

= FIELD (Altitude in DA1-Message)

Receive DA1-Message FROM Digital-altimeter-1	T

= PREV (DA1-Alt-Signal)

Receive DA1-Message FROM Digital-altimeter-1	F

= Obsolete

Receive DA1-Message FROM Digital-Altimeter-1	F	
Time > Time (DA1-Message arrived) + 2 seconds	T	
Powerup		T

Input Value

DA2-Status-Signal

Source: Digital Altimeter 1

Type: Enumerated

Possible Values (Expected Range): {Fail, NCD, Test, Norm}

Exception-Handling:

Arrival Rate (Load): ??

Min-Time-Between-Inputs:
Max-Time-Between-Inputs:

Obsolescence: 2 seconds

Exception-Handling: Assumes value Obsolete

Description:

Comments: Four possible values can be sent signifying Failure Warning, No Computed Data, Functional Test, and Normal Operation.

References:

Appears in: Dig-Alt2

DEFINITION

= FIELD (Status in DA2-Message)

Receive DA2-Message FROM Digital-Altimeter-2	T

= PREV (Dig2-Status-Signal)

Receive DA2-Message FROM Digital-Altimeter-2	F
Time <= Time (DA2-Message arrived) + 2 seconds	T

= Obsolete

Receive DA2-Message FROM Digital-Altimeter-2	F	
Time > Time (DA2-Message arrived) + 2 seconds	T	
Powerup		T

Input Value

DA2-Alt-Signal

Source: Digital Altimeter 2

Type: integer

Possible Values (Expected Range): -20..2500

Exception-Handling: Values below -20 are treated as -20 and values above 2500 as 2500

Units: ??

Granularity: ??

Arrival Rate (Load): ??

Min-Time-Between-Inputs:
Max-Time-Between-Inputs:

Obsolescence: 2 seconds

Exception-Handling:

Description:

Comments:

References:

Appears in: Altitude

DEFINITION

= FIELD (Altitude in DA2-Message)

Receive DA2-Message FROM Digital-altimeter-2	T

= PREV (DA2-Alt-Signal)

Receive DA2-Message FROM Digital-altimeter-2	F

= Obsolete

Receive DA2-Message FROM Digital-Altimeter-2	F	
Time > Time (DA2-Message arrived) + 2 seconds	T	
Powerup		T

$\boxed{\text{Control Input}}$

Inhibit

Source: Cockpit Inibit Button

Type: Enumerated

Possible Values (Expected Range): {on, off}

Arrival Rate (Load):

Min-Time-Between-Inputs:
Max-Time-Between-Inputs:

Obsolescence: None

Description:

Comments:

References:

Appears in: ASW

DEFINITION

= FIELD (Value in Inhibit-Message)

$\boxed{\text{Receive Inhibit-Message from Cockpit}}$ $\boxed{T}$

= PREV (Inhibit)

$\boxed{\text{Receive Inhibit-Message from Cockpit}}$ $\boxed{F}$

= Obsolete

$\boxed{\text{Powerup}}$ $\boxed{T}$

Control Input

Reset

Source: Cockpit Reset Button

Type: Signal

Possible Values (Expected Range): {High}

Arrival Rate (Load):

Min-Time-Between-Inputs:
Max-Time-Between-Inputs:

Obsolescence: Not applicable (lasts only one step)

Description:

Comments:

References:

Appears in: Analog-Alt, DOI-Status, Altitude, Analog.Alt, Dig-Alt1, Dig-Alt2, ASW

DEFINITION

= True

Receive Inhibit Signal	T

= False

Prev (Reset) = True	T	
Powerup		T

Macro

Analog-Valid-and-Below

Description:

Comments:

References:

Appears in: Altitude2

DEFINITION

Analog-alt = Valid	T
Analog-Alt-Signal = below	T

Macro

Analog-Valid-and-Above

Description:

Comments:

References:

Appears in: Altitude2

DEFINITION

Analog-alt = Valid	T
Analog-Alt-Signal = above	T

Macro

Dig1-Valid-and-Below

Description:

Comments:

References:

Appears in: Altitude2

DEFINITION

Dig1-alt = Valid	T
DA1-Alt-Signal < 2000_{THRES}	T

Macro

Dig1-Valid-and-Above

Description:

Comments:

References:

Appears in: Altitude2

DEFINITION

Dig-Alt1 = Valid	T
DA1-Alt-Signal >= 2000_{THRES}	T

Macro

Dig2-Valid-and-Below

Description:

Comments:

References:

Appears in: Altitude2

DEFINITION

Dig2-alt = Valid	T
DA2-Alt-Signal < 2000_{THRES}	T

Macro

Dig2-Valid-and-Above

Description:

Comments:

References:

Appears in: Altitude2

DEFINITION

Dig-Alt2 = Valid	T
DA2-Alt-Signal >= 2000_{THRES}	T

Output Command

DOI-Power-On

Destination: DOI

Acceptable Values: {high}

Initiation Delay: 0 milliseconds

Completion Deadline: 50 milliseconds

Exception-Handling: (What to do if cannot issue command within deadline time)

Feedback Information:

Variables: DOI-status-signal

Values: high (on)

Relationship: Should be on if ASW sent signal to turn on

Min. time (latency): 2 seconds

Max. time: 4 seconds

Exception Handling: DOI-Status changed to Fault-Detected

Reversed By: Turned off by some other component or components. Do not know which ones.

Comments: I am assuming that if we do not know if the DOI is on, it is better to turn it on again, i.e., that the reason for the restriction is simply hysteresis and not possible damage to the device.

This product in the family will turn on the DOE only when the aircraft descends below the threshold altitude. Only this page needs to change for a product in the family that is triggered by rising above the threshold.

References: ↑ ↓

CONTENTS

= discrete signal on line PWR set to high

TRIGGERING CONDITION

Operating Mode	Operational	T
	Not Inhibited	T
State Values	DOI-Status = On	F
	Altitude = Below-threshhold	T
	Prev(Altitude) = At-or-above-threshold	T

Output Command

DOI-Power-On

Destination: DOI

Acceptable Values: {high}

Initiation Delay: 0 milliseconds

Completion Deadline: 50 milliseconds

Exception-Handling: (What to do if cannot issue command within deadline time)

Feedback Information:

Variables: DOI-status-signal

Values: high (on)

Relationship: Should be on if ASW sent signal to turn on

Min. time (latency): 2 seconds

Max. time: 4 seconds

Exception Handling: DOI-Status changed to Fault-Detected

Reversed By: Turned off by some other component or components. Do not know which ones.

Comments: I am assuming that if we do not know if the DOI is on, it is better to turn it on again, i.e., that the reason for the restriction is simply hysteresis and not possible damage to the device.

This product in the family will turn on the DOE only when the aircraft descends below the threshold altitude. Only this page needs to change for a product in the family that is triggered by rising above the threshold.

References: ↑ ↓

CONTENTS

= discrete signal on line PWR set to high

TRIGGERING CONDITION

Operating Mode	Operational	T
	Not Inhibited	T
State Values	DOI-Status = On	F
	Altitude = Below-threshhold	T
	Prev(Altitude) = At-or-above-threshold	T

Output Command

Watchdog-Strobe

Destination: Watchdog Timer

Acceptable Values: high signal (on)

Min-Time-Between-Outputs: 0

Max-Time-Between-Outputs: 200_{PERIOD} msec

Exception-Handling:

Feedback Information: None

Reversed By: Not necessary

Comments:

References:

CONTENTS

= High signal on line WDT

TRIGGERING CONDITION

Operating Mode	Operational	T		
	Startup		T	
	Inhibited			T
State Values	Time <= (Time sent Watchdog Strobe) + 200 msec	T		T
	DOI-Status = Fault-detected	F		
	Time >= (Time entered Altitude.Cannot-be-determined) + 2_{DL} secs.	F		

Operating Mode

ASW

Description:

Comments: No information about how an internal fault is detected, what types detected, etc.

References:

Appears in: DOI-power-on, Watchdog-strobe

DEFINITION

= Startup

Powerup	T

= Operational

Controls.Reset = T	T			
Startup		T	T	T
Analog-Alt = Valid		T		
Dig-Alt1 = Valid			T	
Dig-Alt2 = Valid				T

= Internal-Fault-Detected

Internal-fault -detected	T	
Startup		T
Time >= Time entered Startup + 3 secs		T

Appendix B

omnia apud me mathematica fiunt.

René Descartes (1596-1650)

Matrices for State Space Description of the Altitude Switch

Appendix C

Does anyone believe that the difference between the Lebesgue and Riemann integrals can have physical significance, and that whether say, an airplane would or would not fly could depend on this difference? If such were claimed, I should not care to fly in that plane.

Richard W. Hamming, 1988

Level 3 of Hybrid SpecTRM-RL Model of MTCD