Runtime Verification for Stochastic Systems

by

Cristina M. Wilcox
cwilcox@mit.edu

Submitted to the Department of Aeronautical and Astronautical Engineering
in partial fulfillment of the requirements for the degree of

Masters of Science in Aerospace Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2010

Author ....
Department of Aeronautical and Astronautical Engineering
May 21, 2010

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

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

Runtime Verification for Stochastic Systems

by

Cristina M. Wilcox

cwilcox@mit.edu

Submitted to the Department of Aeronautical and Astronautical Engineering
on May 21, 2010, in partial fulfillment of the
requirements for the degree of
Masters of Science in Aerospace Engineering

Abstract

We desire a capability for the safety monitoring of complex, mixed hardware/software systems, such as a semi-autonomous car. The field of runtime verification has developed many tools for monitoring the safety of software systems in real time. However, these tools do not allow for uncertainty in the system's state or failure, both of which are essential for the problems we care about. In this thesis I propose a capability for monitoring the safety criteria of mixed hardware/software systems that is robust to uncertainty and hardware failure.

I start by framing the problem as runtime verification of stochastic, faulty, hidden-state systems. I solve this problem by performing belief state estimation over a novel set of models that combine Büchi automata, for modeling safety requirements, with probabilistic hierarchical constraint automata, for modeling mixed hardware/software systems. This method is innovative in its melding of safety monitoring techniques from the runtime verification community with probabilistic mode estimation techniques from the field of model-based diagnosis. I have verified my approach by testing it on automotive safety requirements for a model of an actuator component. My approach shows promise as a real-time safety monitoring tool for such systems.

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

Acknowledgments

To many friends and mentors thanks are due;
To Mom and Dad, your love has been a light
Upon my heart, your wisdom certain through
Uncertain times. You helped me win this fight.

For valued counsel, and for patience long,
Mere thanks, Advisor, can't repay my debt.
For all my teachers, thanks cannot be wrong;
An ounce of learning I do not regret.

To all the names I couldn't cram in verse;
Fiancè, Sibling, comrade labmates, friends;
Forgive me, though my gratitude is terse,
It issues from a source that never ends.

And last, for Him who loves me, my redeemer,
Soli Deo gloria et honor.

This research was supported in party by the Ford-MIT Alliance agreement of 2007, and by a grant from the Office of Naval Research through John's Hopkins University, contract number 960101.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis provides a capability for the safety monitoring of embedded systems with stochastic behavior that have hidden state and may fail. I accomplish this by framing the monitoring problem as that of Bayesian belief state update on a combined plant and safety state. Knowledge of plant behavior is encoded as a stochastic model with discrete states.

1.1 Lifelong Runtime Verification of Stochastic, Faulty Systems with Hidden State

In this section I discuss the motivation behind runtime verification and elaborate on its utility for complex embedded systems that include hardware and software. I also introduce the need to handle stochastic systems and systems with hidden state. The subsequent section sketches my approach to estimating safety for such systems, while Section 1.3 discusses related work.

From Model Checking to Runtime Verification

Most complex software systems are deployed with bugs. The field of formal verification strives to prove that software is correct through model checking, but the efficiency and practicality of these methods are hindered by a problem of state explosion, also known as the curse of dimensionality [4]. Though progress has been made and model checking has proven to be a valuable tool for specific applications such as airline reservation system software and space probe control software [4, 8, 12, 25], these methods still do not apply to a wide range of real world systems. In practice, extensive testing instead is used to expose bugs. However, testing is never guaranteed to be exhaustive, and many complex software systems are riddled with problems not caught at design time that are addressed through frequent patches and service packs. Consequently, the field of runtime verification [14] has emerged to check program correctness at runtime, circumventing the combinatorial problem and thus providing complex systems with a safety net for design time verification and testing.

Runtime verification complements testing methods by providing a framework for automated testing that can be extended into a capability for monitoring a system post-deployment. With a runtime verification capability in place, an operational system can detect deviations from formally specified behavior and potentially take corrective action. In this way, runtime verification complements testing methods and provides a capability for fault-tolorance which is desirable for safety critical systems.

Formal Methods in Hardware Design and Operation

Formal verification techniques were developed for software systems, but the use of these techniques as a part of hardware design has been advocated [33] and shown to be feasible for electronic embedded systems such as logical circuits [5, 9, 11]. Formal verification of hardware design is suitable for small systems that can be modeled precisely and who's inputs are known.

However, from a practical standpoint, complex hardware systems have operational environments that may cause significant deviations from modeled behavior, rendering formal verification of design ineffective. These systems can benefit from design verification, but this does not prove that they will operate correctly when deployed. Runtime verification has therefore extended formal verification of hardware systems to deal with complex *mixed systems*, that is, systems that are a mix of hardware and software [7, 29]. Runtime verification of mixed systems provides a capability for monitoring the behavior of a system in the field, with the potential for a corrective functionality that acts based on the output of the monitor.

The Need for Hidden State

Runtime verification for mixed systems assumes observability of properties to be monitored. This thesis argues that for complex hardware systems such as a space probe or a car, the system's state is generally unobservable, due to the high cost of sensing all variables reliably. Hence, in order to perform general runtime verification of these mixed systems, this thesis extends proven runtime verification techniques so that they handle systems with hidden states.

To deal with hidden states, I draw upon inference techniques from the field of Model-based diagnosis (MBD) [15, 41], which require an accurate model of the system components and constraints. MBD applies conflict-directed search techniques in order to quickly enumerate system configurations, such as failure modes, that are consistent with the model and observations. These techniques are suitable for mixed systems and scale well [26–28, 40].

Dealing with Systems that Fail

A second issue, not directly addressed by runtime verification, is that complex systems with long life cycles experience performance degradation due to seemingly random hardware failure. Many systems function well when manufactured, but may become unsafe over time, especially when they are in use for longer than their intended life span. For example, car owners occasionally fail to have their vehicles inspected promptly, which can result in a component, such as the braking system, receiving more use than it was designed for. We want to be able to detect any breaches of safety due to wear and tear in such a situation.

Thus, this thesis advocates the use of a plant model that incorporates stochastic behavior [40], allowing wear and tear to be modeled as stochastic hardware failure. With such a model, specification violations resulting from performance degradation can be detected online and recovery action can be taken, such as the removal of unsafe functions.

1.2 Architecture of the proposed solution

This thesis proposes a capability for the monitoring of formal specifications for mixed systems that are written in Linear Temporal Logic (LTL) [30]. Linear Temporal Logic is a well studied logic that is similar to plain English and expressive enough to capture many important high-level safety requirements. Additionally, the requirements are allowed to be written over hidden system states.

This safety monitoring capability will also have a model of the stochastic, faulty plant captured as a Probabilistic Hierarchical Constraint Automaton (PHCA) [40]. This automaton representation allows for the abstract specification of embedded software, as well as the specification of discrete hardware modes, including known failure modes. Additionally, stochastic transitions may be specified in order to model random hardware failure. Such a model of the system allows the safety monitoring capability to identify hidden system state, including in the case of sensor failure, unmodeled failures, intermittent failures, or multiple faults.

Given sensory information, the safety monitoring capability will then compute online the likelihood that the LTL safety requirements are being met. This is accomplished by framing the problem as an instance of belief state update over the combined physical/safety state of the system, as described in Chapter 4.

Together, the use of LTL and PHCA provide a clean specification method for performing safety monitoring of mixed stochastic systems. Viewing safety monitoring as belief state update on a hybrid of BA and PHCA state provides an elegant framing of the problem as an instance Bayesian filtering.

1.3 Related Work

Next, consider the work presented in this thesis compared to recent prior art.

Some examples of the successful application of runtime verification techniques in software systems are JPaX by Havelund and Roşu [19] and DBRover by Drusinsky [16], both shown to be effective monitors for a planetary rover control program [8]. Another approach given by Kim *et al.*, MaC [23], has proven effective at monitoring formations of flying vehicles [22]. This thesis builds on such work by extending these techniques to deal with mixed stochastic systems.

Peters and Parnas [29] and Black [7] have suggested monitors for runtime verification of systems including hardware, but these works do not consider hidden state, which this thesis does.

Techniques have been developed for the model checking of systems that exhibit probabilistic behavior [4, 20, 34, 38]. While these methods are appropriate for randomized algorithms and have even been applied to biological systems [20], they are not concerned with complex mixed systems, as these mixed systems operate in environments that may cause significant deviations from modeled behavior. Additionally, if a mixed system is modeled having randomly occurring hardware failures, proving its correctness becomes problematic because the model will fail by definition.

More recently, techniques have been demonstrated for the runtime verification of systems that exhibit probabilistic behavior [32, 36]. Sistla and Srinivas present randomized algorithms for the monitoring of liveness properties on simple software systems given as hidden Markov models [36]. Their approach is subtly different from the one presented in this thesis, as they focus on the probabilistic monitoring of **liveness** properties. The properties they are concerned with are not written over hidden states of the system, but instead over the observations that the system generates. They employ counters and timeouts to probabilistically predict whether the system will satisfy the liveness requirement. This thesis does not attempt to predict the satisfaction of liveness requirements, because in doing so a monitor may reject system executions that have not been proven to violate the requirements. Instead of attempting to monitor liveness requirements, we would prefer to convert liveness properties into more specific timed properties (see Chapter 5), making them more useful for specifying the true requirements of the system. Unlike the approach presented in this thesis, Sistla and Srinivas do not provide the capability to monitor safety properties that are written over hidden system states, and thus their methods do not suffice for the purpose of safety monitoring of mixed systems.

Sammapun *et al.* perform monitoring of quantitative safety properties for stochastic systems that have periodic behaviors, such as soft real-time schedulers [32]. A quantitative property says, for instance, that a bad thing e such as a missed deadline must occur no more than n percent of the time. The authors statistically evaluate an execution trace for conformance to the property by checking subsections of the trace for occurrences of the proscribed event e, and counting the number of subsections on which e occurs. If e occurs in five out of 100 subsections, for instance, then they estimate that e occurred 5 percent of the time. In order to assert that a property has been violated, their method must gain confidence in its estimation by evaluating a sufficiently long history of program state. Therefore property violations that occur early in the operation of the system will not be caught. Their approach for the verification of quantitative properties is sound, but they do not monitor properties written over hidden system states, which this thesis does. Additionally, their approach is built on an assumption of periodic system behavior, which this thesis is not limited by.

Runtime verification has been moving towards the monitoring of general properties for mixed stochastic systems, but no work I know of has attempted to monitor properties written over unobservable system states. Additionally, no work has employed a system model appropriate for faulty hardware systems. This thesis provides these novel capabilities.

The rest of this thesis is organized as follows: Chapter 3 details an approach to runtime verification for observable systems. Chapter 4 then solves the problem formally outlined in Chapter 2 by extending runtime verification methods into the realm of stochastic, partially-observable systems. Chapter 5 presents and discusses empirical validation as well as future work.

Chapter 2

The Problem

2.1 Problem Statement

This thesis provides a capability for performing runtime safety monitoring of an embedded system which will fail over its lifetime. This capability should assume that the system is partially observable and behaves stochastically. In addition, the system is implemented as a combination of hardware and software. In order to provide this capability, this thesis solves the following problem:

> Given a safety specification in Linear Temporal Logic, a model of the physical system as a Probabilistic Hierarchical Constraint Automaton, and an observation and command history for the system from time 0 to t, for each time t return the probability that the safety specification is satisfied up to t. This estimate is to be performed online.

2.2 Motivating the Problem

Consider a car as a complex, safety-critical embedded system. The quantity of software onboard cars has increased dramatically over the past decade and is expected to continue increasing exponentially [1]. A luxury vehicle today is shipped with roughly 100 million lines of control code encompassing everything from the powertrain and brakes to onboard entertainment systems. This number is expected to increase to 200 or 300 million lines of code in the near future. The complexity inherent in so great a magnitude of software raises concerns about the safety and reliability of modern automobiles.

Consider specifically the SAFELANE system [2], a module designed to prevent unintentional lane departures due to operator inattention or fatigue. When active, this module detects accidental lane departures, and either warns the operator or actively corrects the vehicle's trajectory. In order to perform course corrections, SAFELANE must have the ability to steer the vehicle. Such a degree of control requires safety measures, thus the design of SAFELANE should include every reasonable precaution to avoid collisions, rollovers, and other dangerous control situations.

This type of semi-autonomous control of a vehicle, along with advanced longitudinal controllers such as adaptive cruise control, has the potential to greatly improve automobile safety and efficiency. Car manufacturers are slow to include these features not because the technology does not exist, but because of the prohibitive liability it represents. Cars are truly safety critical systems. Small design errors in such a subsystem could have serious consequences, such as property damage or loss of human life. These systems would be tested extensively, yet this provides no safety guarantees. To supplement testing, this thesis proposes a capability for monitoring the safety requirements of such an embedded system. With this capability, a system such as a car can be monitored during operation, and violations of the formalized safety requirements can be detected.

2.2.1 Description of the SAFELANE

Consider a model of SAFELANE contained within a simplified car, see Figure 2-1. This car consists of a steering wheel, brake and gas pedal, SAFELANE, and a touchscreen interface that a human operator may use to command SAFELANE (known as a Human-Machine Interface or HMI). SAFELANE consists of a visual sensing system, a decision function that calculates appropriate control actions, and actuation on steering, brake and gas pedals. In addition to the main decision function of SAFELANE, a redundant calculation occurs in parallel. This backup calculation is used as a sanity check for the control system. SAFELANE can be overridden by certain driver actions, like a steering action, and can be disabled by the human operator via the HMI. The steering wheel, gas pedal, and brake pedal are all observable variables, as is the state of the HMI and the visual sensing unit. The autonomous subsystem is able to request control of brake, steering, and acceleration, and receive control based on results of an arbitration between the driver inputs and all autonomous subsystem requests.

2.2.2 An Example Safety Requirement

For a system to behave in a 'safe' manner, it must not endanger the operator through any action or lack of action. In cases where it would be unsafe to act, for instance, if the system cannot determine the correct control action, then the system is excused from its obligation to act.

This is a general description of safety in any system. For a specified system, we can specialize and formalize this description. The main safety requirement of the SAFELANE system is that the system must take appropriate corrective action if it detects an imminent lane departure. Restated:

> If ever the visual sensing system determines that the vehicle is experiencing a lane departure, SAFELANE must request a control action appropriate to the situation, or issue a warning signal.

To formalize this requirement, we write it in a logic known as Linear Temporal Logic (LTL). This logic is described in Section 3.2. The formalized version of the above requirement is:

$$\Box\big(\text{VISUAL\_SENSING} = \mathsf{imminent\text{-}departure} \rightarrow \\ \text{SAFELANE} = \mathsf{Requesting\ Control} \vee \text{SAFELANE} = \mathsf{Warning\ Signal}\big)$$

This is read as "For all time, if visual sensing detects an imminent departure, then

SAFELANE is requesting control or signaling a warning." This requirement specifies the correct functionality of SAFELANE.

2.2.3 A PHCA Plant Model

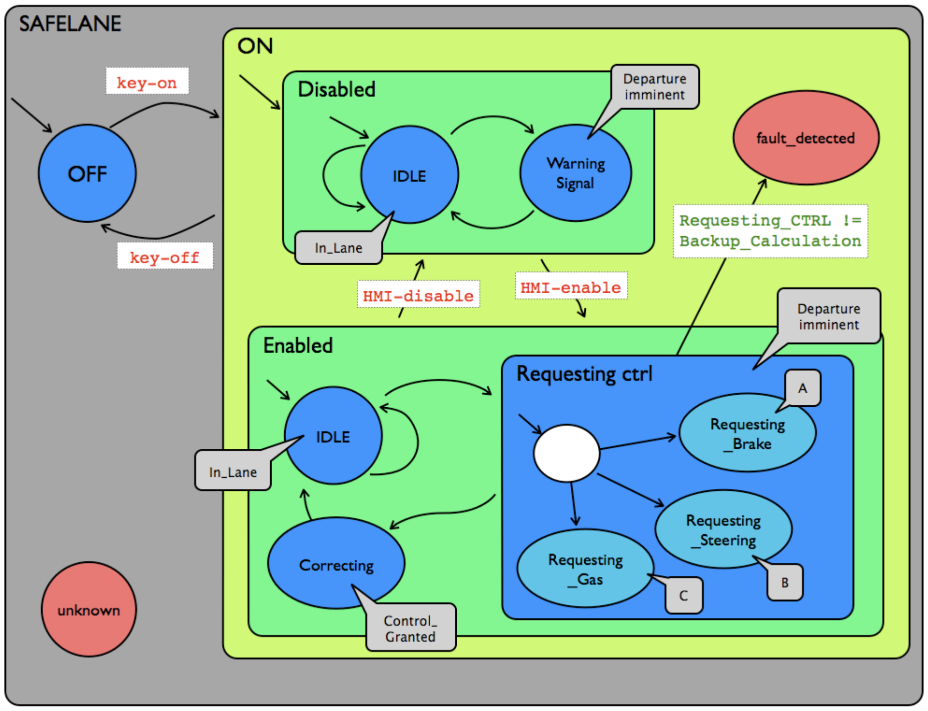


Figure 2-1: The SAFELANE autonomous subsystem and its subcomponents. The system is also comprised of a visual sensing unit that is observable, and actuation units. The system's purpose is to prevent unintentional lane departures.

In order to estimate the current state of the system, our capability requires a model of system behavior. Because the system consists of both hardware and software, the model is of both hardware and software behavior. In Figure 2-1, I show a model of the SAFELANE system as a Probabilistic Hierarchical Constraint Automaton. This type of automaton, detailed in Section 4.3, captures the discrete modes of system operation as sublocations. Transitions may occur probabilistically, component automata may contain sub-automata, and locations within the automata may be constrained by values of system variables or by the operational mode of other components.

In this model, the SAFELANE component may be ON or OFF, and may transition between these modes based on the primitive command from the user of KEY-ON()

or KEY-OFF(). Within the ON mode, the component may be Disabled, Enabled, or there may be a Fault Detected. The user may toggle between Disabled and Enabled with the HMI (Human-Machine Interface), represented by the primitive methods HMI-ENABLE() and HMI-DISABLE(). If SAFELANE is Disabled, it is Idle if the visual sensing unit determines that the vehicle is safely IN LANE, and will transition into a state where it issues a warning signal if the visual sensing system detects a potential lane departure. Similarly, within the Enabled mode, the system transitions from Idle to Requesting Control if the visual system senses a potential lane departure. Within this mode, SAFELANE may request different types of control actions. These choices are constrained by A, B, and C, which are results of the decision function. When control is granted to SAFELANE, it is in the Correcting mode. If ever the result of the decision function does not match the result of the backup calculation, then the system enters the Fault Detected mode. At all times the model may transition into the Unknown mode with a small probability.

The visual sensing system has the following modes, which are observable:

1. Lanes Not Sensible - LNS
2. In Lane - IL
3. Imminent Departure - ID
4. Intentional Lane Departure - ILD
5. Unintentional Lane Departure - ULD

Other observable variables include whether or not SAFELANE is requesting control of the vehicle (Requesting Control or RQC), and whether or not SAFELANE's control request has been granted (CONTROL-REQUEST-GRANTED or CRG).

Example Execution Trace of Capability

Recall the problem statement from Section 2.1:

> Given a safety specification in Linear Temporal Logic, a model of the physical system as a Probabilistic Hierarchical Constraint Automaton, and

an observation (z) and command (c) history for the system from time 0 to t, for each time t return the probability that the safety specification is satisfied up to t. This estimate is to be performed online.

Earlier in this section an example safety specification for the SAFELANE system in LTL was presented, as well as a PHCA model of the system, completing the information that the proposed capability requires before runtime.

At runtime, the capability takes in a command and observation history. For example:

t	1	2	3	4	5
$c_{1:t}$	KEY-ON()	∅	HMI-ENABLE()	∅	∅
$z_{1:t}$	LNS	IL	ILD	IL	{ULD,RQC}

With these inputs, this capability will calculate that the probability of the system meeting its safety requirement as follows:

t	1	2	3	4	5
$\mathbf{P}(\text{SAFE})$	1	1	1	1	1

If, however, the observation at time five does not include RQC:

$$z_5 = \{\text{ULD}\}$$

Then the capability will calculate $\mathbf{P}(\text{SAFE})$ at $t = 5$ to be 0.

2.3 Motivating the Approach

In this section I present two examples of safety requirements to motivate the assumptions behind the approach taken. Those assumptions are that the plant state is hidden, and that the plant hardware may fail stochastically. The first example demonstrates the utility of runtime verification for a system such as SAFELANE when the safety state can be directly observed. The second example illustrates the need to relax the assumption of full system observability for hardware systems, and then further demonstrates the utility of allowing a stochastic plant model.

2.3.1 A Directly Monitorable Safety Requirement

For the SAFELANE example system in Figure 2-1, one important safety measure that can be enforced is to revoke SAFELANE's control privileges when it appears to be making a faulty calculation. If a fault is detected in the decision function (that is, if the main and redundant control computations do not agree), then we require that SAFELANE not be allowed to control the vehicle until it has been power cycled. Restated:

> If ever the result of the control calculation of the decision function does not agree with the backup calculation, from that time on the subsystem shall never receive control of the vehicle.[1]

This is a relatively conservative requirement, which assumes that a faulty computation may have been caused by a component outside the decision function, and thus will not be fixed by simply resetting SAFELANE. This requirement is revisited in Sections 3.2 and 3.3, see Equation (3.2).

The problem statement above assumes we are given the history of observations and control actions. As this safety requirement is written solely over the observable variables REQUEST-X, BACKUP-CALCULATION, and CONTROL-REQUEST-GRANTED, it is possible to check whether the given history logically satisfies the formal statement of the safety requirement. In fact, this is exactly what the field of runtime verification does.

In runtime verification, a safety monitor checks at each time step to see if the formal specification is being satisfied by the system. For example, assume that the following observations are received from the SAFELANE system:

t	1	...	n
z	SAFELANE-REQUEST-BRAKE = **T**, BACKUP-CALC-REQ-BRAKE = **F**	...	CONTROL-REQUEST-GRANTED = **T**

At time $t = 1$, a fault is observed, and so SAFELANE enters the fault-detected mode. However, this does not violate the safety requirement, so the safety monitor

[1] This statement assumes the monitor resets when the car is restarted.

returns a 1, indicating that the system is SAFE. At time $t = n$, control is granted to a faulty SAFELANE, hence the safety monitor calculates that the requirement is being violated and returns a 0.

2.3.2 A Hidden-State Safety Requirement

As was shown in the previous example, given requirements written over observable and control variables, runtime verification is sufficient to provide a safety monitoring capability. However, runtime verification does not currently address the problem of incomplete sensing that is generally associated with hardware. Take, for example, the following safety requirement:

> If ever the brake, gas, or steering actuation fails, from that time on the subsystem shall never receive control of the vehicle.

The requirement says that if any of the steering, brake, or gas pedal actuation mechanisms fail, then SAFELANE should never receive control of any of them. This will prevent the system from asserting control during situations that its controller was not designed for.

Assume the safety monitor has access to the steering wheel encoder data, and to the commands sent by SAFELANE to the steering actuator. Consider the scenario in which SAFELANE commands the steering wheel to turn, and at the next time step the encoder reports that the wheel has not turned. It seems clear that some component of the system is not behaving correctly, but it is unclear whether the actuator is at fault or the encoder.

In this example scenario the state of the actuator is hidden. Put another way, the safety monitor cannot directly observe whether the actuator is failed or not. This means that the safety requirement above is written over hidden states. Thus, the runtime verification approach employed in the previous example will no longer suffice. Instead, the safety monitor must be able to infer the values of these hidden variables.

To accomplish what the runtime verification approach could not, I introduce inference techniques from the field of model-based diagnosis [15]. Intuitively, having a model of system behavior provides a basis for reasoning over hidden system states.

For example, in the scenario described above, the model of correct actuator and encoder behavior dictates that the encoder output mirror the actuator input. An input of a "turn" command to the actuator and an output of "no turn occurred" from the encoder are therefore inconsistent with the model of nominal behavior. Thus the possible system configurations must include either ACTUATOR = failed or ENCODER = failed. This type of model-based reasoning allows the safety monitoring capability to identify system configurations that are consistent with the observed information, and therefore address the problem of incomplete sensing associated with hardware. The runtime verification approach described in the preceding example has no faculty for reasoning over hidden states.

2.3.3 A Safety Requirement for a Stochastic Plant

Consider again the previous requirement:

> If ever the brake, gas, or steering actuation fails, from that time on the subsystem shall never receive control of the vehicle.

When a system includes a hidden system state, such as the operational mode of the actuator, the observable information may allow for multiple consistent diagnoses. In addition to inferring the set of possible system states, it is desirable to be able to infer their relative likelihoods. If the system model is allowed to include stochastic behavior, knowledge of the likelihood of random hardware failure can be encoded as a stochastic transition to a failure mode. In this way, the safety monitor has information about the relative likelihood of failure based on the relative magnitudes of modeled stochastic transitions. Thus it is possible to infer a probability distribution over all possible values of system state. This allows the safety monitor to calculate a probability that the safety requirement is satisfied. The inference of this probability is described in Section 4.2.

2.4 Discussion

This chapter has given three examples of safety requirements that can be monitored for the example system, SAFELANE. These examples were used to highlight a series of behaviors that a safety monitoring system should embody. In particular, in order to monitor these requirements, a capability is required that augments traditional runtime verification by handling systems which have a hidden state and may fail stochastically. The problem then becomes that of estimating the probability that the safety requirements are being satisfied, given observations and commands. This problem can be solved using tools from model-based diagnosis in order to reason probabilistically over hidden system states.

Chapter 3 describes the traditional runtime verification approach. This approach allows for the monitoring of requirements written over observable system variables. Chapter 4 augments the runtime verification approach with inference techniques enabling the monitoring of safety requirements that are written over hidden states.

Chapter 3

Calculating System Safety: Runtime Verification for Observable Systems

x	:	Plant (physical) state	$\mathbf{\Lambda}$	:	Formal safety specification
z	:	Observations	q	:	Safety state
c	:	Commands	α	:	An LTL statement
x_t	:	x at time t	W	:	A word legible to a Büchi Automaton
$z_{1:t}$	:	short for $\{z_1, z_2, \ldots, z_t\}$	$\mathbf{T}$	:	the Boolean truth value

Recall that the specific problem solved by this thesis is that of taking a formal specification $\mathbf{\Lambda}$ of desired system behavior in Linear Temporal Logic (LTL), a description $\mathbf{\Phi}$ of the behavior of the system as a Probabilistic Hierarchical Constraint Automaton (PHCA), as well as a series of commands c to and observations z from the system, and then returning a probability that the system is consistent with $\mathbf{\Lambda}$. I will perform this calculation under the assumption that the system may behave stochastically and has hidden states.

In this chapter I revisit the solution to the problem of runtime verification of a system in which we assume full observability and deterministic behavior. In Chapter 4, I extend the runtime verification approach to an exact algorithm handling mixed hardware/software systems by dealing with hidden-state and stochasticity.

Within this chapter I present the runtime monitoring approach in Section 3.1. I then give an in depth introduction to Linear Temporal Logic in Section 3.2, then Büchi Automata (BA), a representative state machine format for LTL, in Section 3.3. I discuss the conversion from LTL to BA in Section 3.4.

3.1 Runtime Verification

The field of runtime verification [14] arose from the desire to check the correctness of complex programs without needing to check every possible execution of the software. This is in contrast to model checking methods [4, 13], where a mathematical model of the system is checked against a formal specification in order to prove correctness before a system is deployed. Model checking only proves that the given model of the program meets the specification, as opposed to proving that the implementation is correct, and is therefore only as good as the model. Additionally, model checking can also only check those properties that do not depend on environmental input, as these inputs are not known completely at design time. Lastly, these methods suffer from a problem of state explosion related to the need to check every possible execution. There are serious problems involved with enumerating all possible executions explicitly or symbolically, and many of these executions are quite unlikely. Therefore runtime verification only checks one relevant execution, that is, the one that actually occurs. Because this execution is generated by the running system, verifying its correctness verifies that the system is behaving correctly, not just the model. Additionally, properties dependent on the environment may also be checked.

The goal of runtime verification is to determine, at every time step, if the system is currently meeting its correctness requirement $\mathbf{\Lambda}$. This description of system correctness $\mathbf{\Lambda}$ is a set of formally specified, high-level and time-evolved behaviors that have been determined to be necessary for safe system operation. Under the assumptions that the system is fully observable and behaves deterministically, we can accomplish this using the tools I present later in the chapter.

Section 3.2 describes Linear Temporal Logic (LTL), a logic that is suitable for formally specifying the correct behavior of a reactive system; with this logic we can write $\mathbf{\Lambda}$. Section 3.3 then presents an automaton format that is equivalent to LTL, the Büchi Automaton (BA). A Linear Temporal Logic statement is compiled into a Büchi Automaton using the method presented in Section 3.4. When a Büchi Automaton for an LTL statement $\mathbf{\Lambda}$ is run on a program execution trace, it will reject if $\mathbf{\Lambda}$ is violated by the trace, and it will accept if the requirement is satisfied thus far by the trace. Therefore, using the tools presented later in this chapter, we are able verify at runtime that a system is behaving correctly.

3.2 Linear Temporal Logic

In this section I informally and formally describe Linear Temporal Logic [3, 30] as well as give some examples of important temporal properties that can be stated in LTL.

Temporal or tense logics were created in part by researchers wishing to ensure program correctness [30]. These logics were and are used to describe correctness criteria for computer programs, such as in [25]. It was shown that Linear Temporal Logic is as expressive as first order logic [21]. This fact, combined with LTL's similarity to natural language, makes it a powerful yet concise and straightforward language for representing desired operating criteria.

LTL extends standard Boolean logic to include the notion of time. The truth of the propositions comprising an LTL statement $\mathbf{\Lambda}$ may vary over time. A set of truth assignments to all propositions of $\mathbf{\Lambda}$, for all time, is called a word $\mathcal{W}$, and is said to have infinite length. For example, if $\mathbf{\Lambda}$ is written over the proposition α, then is is possible for α to be true at every time step, which we write:

$$\mathcal{W}_0 = \alpha\,\alpha\,\alpha\ldots$$

If $\mathcal{W}$ is such that the truth assignments to the propositions across time satisfy $\mathbf{\Lambda}$, then that word is said to be in the language of $\mathbf{\Lambda}$. Hence the language of $\mathbf{\Lambda}$ is the set of infinite-length words that satisfy the statement. For example, the language of

$\mathbf{\Lambda}_0 = \Box\alpha$, which requires that α be true for all time, is exactly $\mathcal{W}_0$. If $\mathbf{\Lambda}$ is a statement of system correctness, then the language of $\mathbf{\Lambda}$ represents all possible correct system executions.

However, in runtime verification, we can never observe an infinite sequence of system states, only a finite prefix W of that infinite execution $\mathcal{W}$. The task of runtime verification is then to determine whether or not the finite word W is in the language of $\mathbf{\Lambda}$. Without the infinite execution trace, this is not necessarily decidable. The definition of LTL presented in this thesis is based on those provided by the recent runtime verification literature [6, 18], which take into account our desire to decide the membership of the *finite* word W in the language of $\mathbf{\Lambda}$.

3.2.1 An Informal Description of LTL

LTL augments standard Boolean logic with temporal operators, and hence is comprised of temporal and non-temporal (which can be thought of as *spatial*) operators. Boolean operators describe relationships between the truth of propositions *at a single point in time*. For instance, the formula $\alpha \vee \beta$ says that currently, either α is true, or β is true. If α and β are propositions, then one way to satisfy this formula is:

$$\alpha = \mathbf{T}$$

The other Boolean operators are included in LTL as well, such as conjunction ($\wedge$), negation ($\neg$), and implication ($\rightarrow$).

Temporal operators describe relationships between propositions that *span time*. One example is the temporal operator "eventually" (F, for future, or $\Diamond$), which requires that at some time now or in the future its argument must be true. For instance, the formula $\Diamond\alpha$ (read: "eventually alpha") can be satisfied thusly, where the ellipses indicate that the intervening values are similar to what precedes them, and the dash indicates that the value of the variable does not matter:

$t = 1$	2	3	...	n	$n+1$
$\alpha = \mathbf{F}$	$\alpha = \mathbf{F}$	$\alpha = \mathbf{F}$	...	$\alpha = \mathbf{T}$	-

The dual of F is "always" (G or $\Box$), which says that a proposition must be true for all time, or globally. So to satisfy $\Box\alpha$, only this will do:

$t = 0$	1	...	n	...	t_{end}
$\alpha = \mathbf{T}$	$\alpha = \mathbf{T}$	...	$\alpha = \mathbf{T}$	...	$\alpha = \mathbf{T}$

Intuitively, G is the dual of F because if α must always be false, then it is never the case that it is eventually true:

$$\Box\neg\alpha = \neg\Diamond\alpha$$

The third temporal operator we consider is "until" (plain U or $\mathcal{U}$). It is a binary operator that says that its first argument must hold until the second is true, and the second must eventually happen. The following values satisfy $\alpha\mathcal{U}\beta$ (read: "alpha until beta"):

$t = 1$	2	...	n	$n + 1$
$\alpha = \mathbf{T}$	$\alpha = \mathbf{T}$	$\alpha = \mathbf{T}$	$\alpha = \mathbf{F}$	-
$\beta = \mathbf{F}$	$\beta = \mathbf{F}$	...	$\beta = \mathbf{T}$	-

The $\Diamond$ operator can be written in terms of $\mathcal{U}$, as well. For example, $\Diamond\alpha$ may also be written:

$$\mathbf{T}\mathcal{U}\alpha$$

The dual of $\mathcal{U}$ is "release" or $\mathcal{R}$ (sometimes written V). $\alpha\mathcal{R}\beta$ says that α becoming true releases β from its obligation to be true at the next time step. This operator does not require that α eventually be true, in contrast to the $\mathcal{U}$ operator. Thus, there are two general ways for a $\mathcal{R}$ formula, such as $\alpha\mathcal{R}\beta$, to be satisfied:

	$t = 1$	2	...	n	$n + 1$
1	$\alpha = \mathbf{F}$	$\alpha = \mathbf{F}$	...	$\alpha = \mathbf{T}$	-
	$\beta = \mathbf{T}$	$\beta = \mathbf{T}$	$\beta = \mathbf{T}$	$\beta = \mathbf{T}$	-
2	$\alpha = \mathbf{F}$	$\alpha = \mathbf{F}$	...	$\alpha = \mathbf{F}$	...
	$\beta = \mathbf{T}$	$\beta = \mathbf{T}$	$\beta = \mathbf{T}$	$\beta = \mathbf{T}$	$\beta = \mathbf{T}$

The first way is similar to $\mathcal{U}$, with the difference that β must be true up to and *at the same time as* the first time that α holds (at time n). After $t = n$, the formula is satisfied and the values of neither matter. The second way for the example release formula to be satisfied is simply for β to always be true.

The final temporal operator we introduce is called "next" (X or $\bigcirc$). This operator is different from the other temporal operators because it describes what must happen at the next time step, and doesn't constrain any other time steps. If $\bigcirc\alpha$ holds at $t = 1$, then the following satisfies the property:

$t = 1$	2	...
-	$\alpha = \mathbf{T}$	-

This type of property can be useful, but also introduces ambiguity into the specification if the amount of real time between $t = 1$ and $t = 2$ is left unspecified. For example, asserting the property $\textsc{turn-key} \rightarrow \bigcirc\textsc{car-on}$ does not specify whether the car should be $\textsc{on}$ one second or one hour from when then key is turned. In other words, it is unclear what system behavior will satisfy this formula. Additionally, as noted by [18], this operator is problematic for finite execution traces because the meaning of $\bigcirc\alpha$ at the last state in the trace is undefined. Thus, in order to avoid the ambiguity $\bigcirc$ causes, we do not allow it to appear explicitly in the safety specification $\mathbf{\Lambda}$. However, $\bigcirc$ is an essential concept, and we still use the notion when compiling LTL statements into Büchi Automata.

3.2.2 Examples

In the temporal logic literature, *safety* properties are defined as those which say that some "bad" thing must never occur [3]. Pure safety properties can be expressed by using the always ($\Box$) operator. A simple safety property for your life might be:

$$\Box(\neg\textsc{car-accident})$$

Equivalently, we can say that something "good" must constantly be occurring. A simple yet essential safety property for a graduate student is:

$$\Box(\textsc{positive-thesis-progress})$$

A different type of property is a *liveness* property, which says that some "good" thing must eventually happen. In a software system, important liveness properties might include termination, that is, that a program will eventually return control:

$$\Diamond(\textsc{program-returns})$$

or responsiveness, that is, that a program will eventually respond when a request is made:

$$\Box\big(\textsc{receive-request} \rightarrow \Diamond(\textsc{respond-to-request})\big)$$

These types of statements can never be definitively violated, though they can be completely satisfied at some point. In other words, there is always hope that the required "good thing" may still happen. For a graduate student, the all-important liveness statement is:

$$\mathbf{\Lambda}_1 = \Diamond(\textsc{graduate}) \tag{3.1}$$

Alpern and Schneider [3] formally define safety and liveness, and show that the set of all properties expressible by LTL is the union of all safety and all liveness properties.

For the SAFELANE example system presented in Chapter 2, we can describe some important high-level desired properties in LTL. The example given in Section 2.3.1 was that we might wish an autonomous subsystem of an automobile to be denied control of the vehicle if an error is detected in its calculations:

$$\mathbf{\Lambda}_2 = \Box(\textsc{fault-monitor}\!=\!\mathbf{T} \rightarrow \Box(\textsc{control-request-granted}\!=\!\mathbf{F})) \tag{3.2}$$

For the same autonomous subsystem, we may require that the subsystem not command an acceleration for a vehicle that has stopped unless the driver releases the brake, depresses the gas pedal, or enables the system through the human-machine interface (HMI):

$$\begin{aligned}&\Box(\textsc{vehicle-halts}\!=\!\mathbf{T} \rightarrow\\ &\quad(\textsc{brake}\!=\!\mathbf{F} \vee \textsc{gas-pedal}\!=\!\mathbf{T} \vee \textsc{hmi-enable}\!=\!\mathbf{T})\ \mathcal{R}\ \textsc{control-granted}\!=\!\mathbf{F})\end{aligned}$$

These final two examples due in part to Black [7].

3.2.3 LTL Formal Definition

If $\mathfrak{p}$ is a proposition (a statement that evaluates to $\mathbf{T}$ or $\mathbf{F}$), then a well formed LTL statement α is outlined by the the following grammar:

$$\begin{aligned} \alpha = {} & \mathbf{T} \\ & \mid \mathfrak{p} \\ & \mid \neg\alpha \\ & \mid \alpha \wedge \alpha \\ & \mid \alpha \mathcal{U} \alpha \end{aligned}$$

I also use the following standard abbreviations for LTL statements α and β:

$$\alpha \vee \beta \equiv \neg(\neg\alpha \wedge \neg\beta)$$

$$\alpha \rightarrow \beta \equiv \neg\alpha \vee \beta$$

$$\Diamond\alpha \equiv \mathbf{T}\mathcal{U}\alpha$$

$$\Box\alpha \equiv \neg(\Diamond\neg\alpha)$$

$$\alpha\mathcal{R}\beta \equiv \neg(\neg\alpha\mathcal{U}\neg\beta)$$

Say W is a finite word over time consisting of the individual letters $(\sigma_1\sigma_2\sigma_3\ldots\sigma_n)$, where each σ_t represents an assignment to a set of propositions Σ at time t. Then the original LTL operators are formally defined as follows, where $W \vDash \alpha$ means that α is entailed by W:

$$\begin{array}{lll} W \vDash \mathfrak{p} & \text{iff} & \mathfrak{p} \in \sigma_1 \\ W \vDash \neg\alpha & \text{iff} & W \nvDash \alpha \\ W \vDash \alpha \wedge \beta & \text{iff} & W \vDash \alpha \text{ and } W \vDash \beta \\ W \vDash \alpha\mathcal{U}\beta & \text{iff} & \exists k \text{ s.t. } \sigma_k..\sigma_n \vDash \beta \text{ and} \\ & & \qquad\qquad \sigma_1..\sigma_{k-1} \vDash \alpha \end{array}$$

This specifies that a word W satisfies a proposition $\mathfrak{p}$ if the first letter of W satisfies

$\mathfrak{p}$. Restated, W satisfies $\mathfrak{p}$ if $\mathfrak{p}$ is in the set of propositions that hold at time $t = 1$. A negated formula holds if its positive form does not hold. A conjunction holds if both arguments hold individually. Lastly, a formula involving the until operator $(\alpha \mathcal{U} \beta)$ is entailed by W if there is some partition of W into $\sigma_1..\sigma_{k-1}$ and $\sigma_k..\sigma_n$, such that α is entailed by the first substring $(\sigma_1..\sigma_{k-1})$, and β is entailed by second $(\sigma_k..\sigma_n)$.

3.2.4 LTL Summary

In this section I introduced Linear Temporal Logic informally and formally, and gave some examples of different classes of properties expressible by LTL, as well as some examples of properties that are relevant to the SAFELANE example system, presented in Chapter 2. LTL is a simple yet expressive language that allows us to write formal specifications of correct behavior for complex embedded systems. Next we explore an executable form of LTL, which we use to monitor LTL statements at runtime.

3.3 Büchi Automata

Nondeterministic[1] Büchi Automata (BA) extend nondeterministic finite automata (NFA) [35] to operate on infinite-length words, allowing us to use a Büchi Automaton to represent the language of a Linear Temporal Logic statement [4,39]. This provides us with an executable form of an LTL statement. With a BA we can perform monitoring of its corresponding LTL statement on some input. Recall that we are only interested in finite-length state trajectories, as no infinite execution can be observed. Therefore we will modify the accepting conditions of a canonical BA [10] to better represent our interests. In this section I define the modified BA and give an example.

3.3.1 BA Formal Definition

A Nondeterministic Büchi Automaton is a tuple $\langle Q, Q_0, F, \Sigma, T \rangle$ such that:

[1]The set of properties expressible by Nondeterministic Büchi Automata is not the same as the set expressible by Deterministic Büchi Automata. Specifically, the full set of ω-regular languages is *not* expressible by the latter. In this chapter we work with Nondeterministic Büchi Automata, which I abbreviate as simply “BA,” as is common.

Q	Finite set of states
$Q_0 \subseteq Q$	Set of start states
$F \subseteq Q$	Set of accepting states
Σ	Input alphabet
$T : Q \times \Sigma \rightarrow 2^Q$	Transition function

States

The states Q of a BA can be thought of as representing abstract *safety states* of the underlying physical system.[2] A BA is a tool for tracking the progress of the system through these meta-states. For instance, recall the example LTL statement, Equation (3.1) given in Section 3.2.2:

$$\mathbf{\Lambda}_1 = \Diamond(\textsc{graduate})$$

This example liveness requirement is modeled with two safety states: GRADUATED and HAVE-NOT-GRADUATED. Consequently, the BA corresponding to $\mathbf{\Lambda}_1$ has two states:

$$\mathrm{BA}_{\mathbf{\Lambda}_1}(Q) = \{q_g, q_h\} = \{\textsc{graduated}, \textsc{have-not-graduated}\}$$

The underlying physical system may have very complicated dynamics involving funding and advising situation, class schedule, red tape, writer's block, *et cetera*, but the BA does not represent these dynamics, only the safety state of the system.

Start States

The start or initial states Q_0 are the states that are marked before the automaton begins execution. They represent the initial safety state of the system. The automaton must have at least one start state. The BA for $\mathbf{\Lambda}_1$ starts in the HAVE-NOT-GRADUATED state:

$$\mathrm{BA}_{\mathbf{\Lambda}_1}(Q_0) = \{q_h\}$$

[2]This thesis refers to the states of the BA as safety states, whether or not the BA is constructed to monitor a safety property as defined in Section 3.2.2.

Accepting States

The accepting states F represent safety states in which the LTL specifications corresponding to the BA are being satisfied. The example $\mathbf{\Lambda}_1$ is satisfied upon graduation, and not before. As soon as this liveness requirement is met once, it is satisfied permanently. Therefore, the GRADUATED state is the single accepting state for this example:

$$\mathrm{BA}_{\mathbf{\Lambda}_1}(F) = \{q_g\}$$

Alphabet

The alphabet Σ of a BA is the set of symbols that it may read in to advance the safety state. Because the safety state of the system depends on the progression of the system's physical configurations, Σ will consist of all possible physical configurations. We assume each configuration, or physical state, can be fully represented by a unique set of literals. Returning to the example safety requirement $\mathbf{\Lambda}_1$, we can represent one configuration of the underlying physical system with the set of literals:

$$\sigma_{eg} = \{\neg\text{THESIS-COMPLETED}, \text{CLASS-REQUIREMENTS-SATISFIED}, \ldots\}$$

where $\sigma_{eg} \in \Sigma$.

The size of the alphabet depends on the number of unique underlying system configurations and is generally exponential in the number of system components.

Transition Function

The transition function T defines the transitions that are enabled from a certain state by a certain letter in the alphabet. In nondeterministic automata, many transitions may be enabled at once, or no transitions may be. Hence the resulting value of T is in the power set of Q.

In the graduation example above, perhaps the transition between states q_h = HAVE-NOT-GRADUATED and q_g = GRADUATED is enabled by the following letter of Σ:

$$\begin{aligned}\sigma_g &= \{\textsc{thesis-completed}, \\ &\quad \textsc{class-requirements-satisfied}, \\ &\quad \textsc{advisor-signature}\}\end{aligned}$$

Written another way, the transition function contains the entry:

$$T(q_h, \sigma_g) = \{q_g\}$$

3.3.2 Operation of BA

Büchi Automata operate in almost the same way that simple finite-state machines do; they receive inputs one at a time, advance the currently marked states according to their transition function T, and then either accept or reject at the end of the input. In nondeterministic automata, more than one state may be marked at once and more than one transition may be taken at each time step. Additionally, we allow unguarded transitions, called ε-transitions in Sipser's text [35]. We denote these as transitions that are guarded by $\mathbf{T}$.

Accepting condition

In contrast to nondeterministic finite automata, instead of determining acceptance by the set of marked states at the end of the run, canonical BA accept if at least one accepting state (one of F) is visited infinitely often during the run. However, since we are considering only finite runs over the BA in this thesis, corresponding to the finite state sequence generated by an embedded system, we alter the stopping condition of a canonical BA. Therefore we say that we *accept permanently* if all eventualities (liveness conditions) are satisfied and there are no safety conditions. We say that we are *are accepting* at a time when all safety conditions are being met, and we say that we *reject* if any safety condition is ever violated. Bauer *et al.* [6] define a three-valued LTL for this purpose.[3] Hence, our accepting states will be states that represent (1) the satisfaction of eventualities, or (2) the perpetual fulfillment of a safety requirement.

[3]The three values are $\{\mathbf{T}, \mathbf{F}, ?\}$, which the authors jocularly call the good, the bad, and the ugly.

Algorithm 1: BA-RUN

Algorithm 1 gives the formal description of running a Büchi Automaton.

Algorithm 1 BA-RUN

```
1: procedure BA-RUN($w_1 w_2 ... w_n, Q_0, T, F$)
2:     $Q_0^m \leftarrow Q_0$.                          ▷ Initialize the set of marked states.
3:     for each time $t = 1..n$ do
4:         $Q_t^m = \{\}$
5:         for marked states $q \in Q_{t-1}^m$ do
6:             $Q_t^m = Q_t^m \bigcup T(q, w_t)$
7:         end for
8:         if $Q_t^m = \varnothing$ then                ▷ No transitions were enabled.
9:             reject
10:        else if $\exists q$ st $q \in \left(Q_t^m \bigcap F \bigcap T(q, \mathbf{T})\right)$ then ▷ An accepting state with an
                                                      $\varepsilon$ self-transition is marked.
11:            accept
12:        end if
13:    end for
14:    if $Q_n^m \bigcap F = \varnothing$ then          ▷ No accepting state is marked.
15:        reject
16:    else
17:        accept
18:    end if
19: end procedure
```

Initially, the set of states in Q_0 is marked (line 2). The algorithm then loops for each input letter w_t. The set of currently marked states Q_t^m is created in line 4. For each previously marked state q, any states reachable from q with the input letter w_t are added to the set of currently marked states (line 6). If the algorithm finds in line 8 that no transitions were enabled by w_t, then it *rejects*. This corresponds to a safety requirement being violated at time t. On the other hand, if it finds in line 10 that one of the currently marked states is an accepting state with a self-loop guarded by $\mathbf{T}$, then it *accepts*. This corresponds to the satisfaction of all eventualities (i.e. liveness conditions that are described in Section 3.2.2).

Every time the end of the time loop (line 13) is reached without rejecting or accepting, the automaton *is accepting*, but continues operating. This corresponds to the indeterminate case in which membership of the input word in the language of the automaton has not yet been proved or disproved. In this case, the system has satisfied all safety requirements up to this point, but has not yet fulfilled all liveness requirements. The automaton therefore *accepts* based on the fact that the system is safe thus far, and then continue operating in order to continue monitoring the requirements and in hopes that the liveness requirements will be met.

After all inputs have been read, the algorithm checks to see if any accepting states are marked (line 14). If no accepting state is marked, then it *rejects* in line 15. This corresponds to the failure of the system to satisfy a liveness condition. On the other hand, if an accepting state is marked, then it *accepts*. In this case, all safety requirements were met at all times, and so at the end of the system execution, the system is deemed safe.

An Example of BA Execution: Monitoring a Safety Property

Consider again the example specification, Equation (3.2), given in Section 3.2.2:

$$\mathbf{\Lambda}_2 = \Box(\textsc{fault-monitor}\!=\!\mathbf{T} \rightarrow \Box(\textsc{control-request-granted}\!=\!\mathbf{F}))$$

The corresponding Büchi Automaton for $\mathbf{\Lambda}_2$ is shown in Figure 3-1.

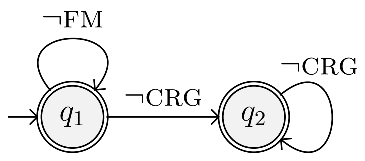


Figure 3-1: The Büchi Automaton for a safety requirement of the SAFELANE example system.

In Figure 3-1, the arrow coming from nowhere on the left indicates that state q_1 is the start state of the automaton. State q_1 has a self transition that is guarded by the requirement that $\textsc{fault-monitor}\!=\!\mathbf{F}$, shortened to $\neg\textsc{fm}$. Similarly, the transition from state q_1 to q_2 is guarded by $\textsc{control-request-granted} = \mathbf{F}$, as is the self transition on state q_2. Accepting states are denoted with a double circle. In this example, both states are accepting states.

Formally, the automaton is:

$$\begin{aligned} Q &= \{q_1, q_2\} \\ q_0 &= \{q_1\} \\ F &= \{q_1, q_2\} \\ \Sigma &= \{\text{FM}\wedge\text{CRG} = \sigma_1, \\ &\quad \text{FM}\wedge\neg\text{CRG} = \sigma_2 \\ &\quad \neg\text{FM}\wedge\text{CRG} = \sigma_3, \\ &\quad \neg\text{FM}\wedge\neg\text{CRG} = \sigma_4\} \end{aligned} \qquad T = \begin{cases} \{q_1\} & \text{for } q_1, \sigma_3 \\ \{q_1, q_2\} & \text{for } q_1, \sigma_4 \\ \{q_2\} & \text{for } q_1, \sigma_2 \\ \{q_2\} & \text{for } q_2, \sigma_{2,4} \\ \varnothing & \text{otherwise} \end{cases}$$

In the alphabet (Σ) specification, I introduce abbreviated names for each element of the set, for example, σ_1 for element FM $\wedge$ CRG.

To demonstrate the operation of the automaton, consider the input word:

$$W_2 = \begin{array}{ccccc} t=1 & 2 & 3 & 4 & 5 \\ \hline \neg\text{FM} & \neg\text{FM} & \neg\text{FM} & \text{FM} & \neg\text{FM} \\ \text{CRG} & \neg\text{CRG} & \text{CRG} & \neg\text{CRG} & \text{CRG} \end{array}$$

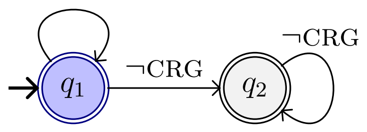
which could also be written:

$$W_2 = \sigma_3\sigma_4\sigma_3\sigma_2\sigma_3.$$

Next we examine BA-RUN(W_2).

0. Initialization ($t = 0$) The first step of execution creates the initial set of marked states, equal to the set of initial states. At the end of this step, the initial state is marked. State marking can be thought of either graphically - state q_1 is occupied - or in terms of the set of marked states, Q_0^m. In the automata depicted below, transitions taken at the current time step are bold, and the resulting marked states are blue.

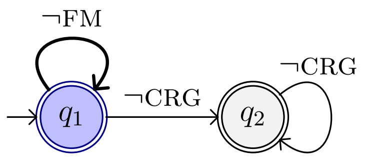
$Q_0^m = \{q_1\}$



1. First Iteration $\big(t = 1$, input $\sigma_3 = \neg\text{FM} \wedge \text{CRG}\big)$ For each marked state, the algorithm attempts to advance the automaton. The automaton has one marked state,

q_1, and the input letter $\sigma_3 = \neg\text{FM}\wedge\text{CRG}$ will enable one transition, the self loop on q_1. In other words, the transition function T outputs q_1 for inputs $\{q_1, \sigma_3\}$.

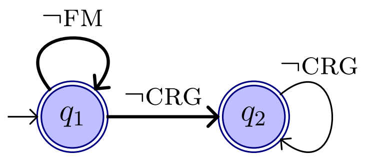
$$Q_1^m \leftarrow q_1 = T(q_1, \sigma_3)$$



For the physical system, this means that no fault was observed in the autonomous subsystem and a control request was granted at this time step. Because this time step ends with accept states marked, the system is currently satisfying the safety requirement and therefore the automaton *is accepting*. This result makes sense because the requirement is that the subsystem never have control if a fault is ever observed.

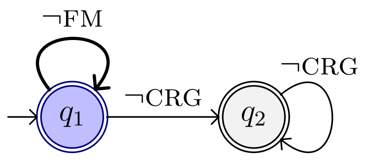
2. Second Iteration $\big(t = 2$, input $\sigma_4 = \neg\text{FM} \wedge \neg\text{CRG}\big)$ At this time step, both possible transitions are enabled, and the automaton is now in both states at once. No fault has been observed, hence the automaton *is accepting* still.

$$Q_2^m \leftarrow \{q_1, q_2\} = T(q_1, \sigma_4)$$



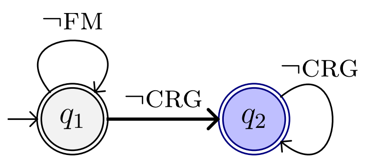
3. Third Iteration $\big(t = 3$, input $= \neg\text{FM} \wedge \text{CRG}\big)$ This input letter (σ_3) enables the self transition on q_1, but no others. Again, the automaton *is accepting* because the safety property is being satisfied.

$$Q_3^m \leftarrow \{q_1\} \bigcup \{\varnothing\} = T(q_1, \sigma_3) + T(q_2, \sigma_3)$$



4. Fourth Iteration $\big(t = 4$, input $= \text{FM} \wedge \neg\text{CRG}\big)$ For the fourth input letter σ_2, the sole transition enabled is the one from q_1 to q_2.

$Q_4^m \leftarrow q_2 = T(q_1, \sigma_2)$



For the physical system, this means that a fault was observed in the autonomous subsystem, but that no control request was granted to it. Requirement (3.2) has not yet been violated, and so the automaton *is accepting*. However, the only way to satisfy Equation (3.2) at this point is to never grant a control request to the autonomous subsystem ($\square\neg\textsc{crg}$).

5. Final Iteration $\big(t = 5$, input $= \neg\textsc{fm}\wedge\textsc{crg}\big)$ For the final letter, σ_3 again, no transitions are enabled.

$$Q_5^m \leftarrow \varnothing = T(q_2, \sigma_3)$$

The automaton now has no marked states, therefore the safety requirement has been irrevocably violated. The automaton now *rejects* immediately.

Another Example of BA Execution: Monitoring a Liveness Property

Now consider the example liveness property (3.1) from Section 3.2.2:

$$\mathbf{\Lambda}_1 = \Diamond(\textsc{graduate})$$

Section 3.3.1 describes the formal structure of the corresponding BA, represented in Figure 3-2.

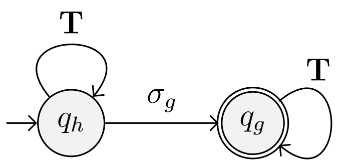


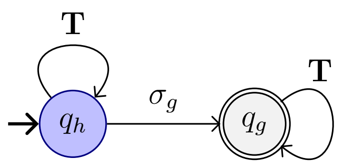
Figure 3-2: The BA for an example liveness property.

The automaton starts in q_h, the state HAVE-NOT-GRADUATED. When the letter σ_g is read, the automaton transitions to state q_g, GRADUATED. Recall that σ_g is the system configuration:

$$\begin{aligned}\sigma_g \quad = \quad \{&\text{THESIS-COMPLETED},\\ &\text{CLASS-REQUIREMENTS-SATISFIED},\\ &\text{ADVISOR-SIGNATURE}\}\end{aligned}$$

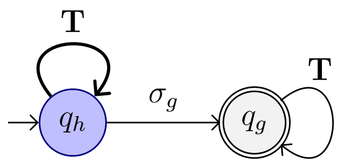
Another letter in the alphabet of this automaton is σ_c = CLASS-REQS-SATISFIED, representing the configuration in which class requirements have been satisfied, but neither THESIS-COMPLETED nor ADVISOR-SIGNATURE is true. Consider the operation of this automaton on $W_1 = \sigma_c\sigma_c\sigma_g\sigma_g$. The automaton begins in q_h:

$$Q_0^m = \{q_h\}$$



The first two input letters, both σ_c, only enable the self-transition on q_h:

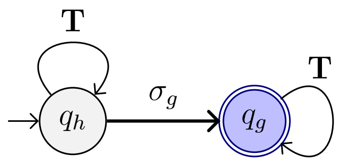
$$Q_2^m \leftarrow q_h = T(q_h, \sigma_c)$$



The automaton "waits" in q_h for the liveness requirement to be satisfied. As long as monitoring is still occurring, there is hope that the property will be satisfied, and so the automaton *is accepting*, even though no accepting states are marked.

The third input letter, σ_g, enables the transition from q_h to q_g:

$$Q_3^m \leftarrow q_g = T(q_h, \sigma_g)$$



At this point, an accepting state with a TRUE self-transition is marked, so the automaton *accepts* permanently per line 10 of Algorithm 1. Even though the system continues running, there is sufficient information to prove the membership of W_1 in $\mathbf{\Lambda}_1$.

3.3.3 BA Summary

In this section I introduced the Nondeterministic Büchi Automaton, a state machine format that operates on infinite words. We saw that this is relevant to the runtime verification of stochastic systems in that it allows us to monitor a formal requirement written in LTL. I also presented an example of this monitoring. In the next section I discuss how BA are obtained given an LTL statement.

3.4 LTL to BA conversion

In order to automate the monitoring of a Linear Temporal Logic statement $\mathbf{\Lambda}$, we convert it to a Büchi Automaton and execute it, as discussed above in Section 3.3. For example, LTL conversion maps the statement:

$$\mathbf{\Lambda}_2 = \Box(\textsc{fault-monitor}=\mathbf{T} \rightarrow \Box(\textsc{control-request-granted}=\mathbf{F})) \tag{3.3}$$

into the following automaton:

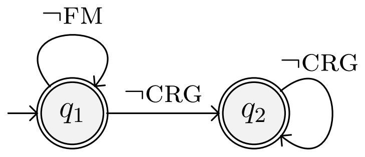


Figure 3-3: The Büchi Automaton for a safety requirement of the SAFELANE example system. This automaton is described in detail in Section 3.3.2.

To perform this conversion, I use the method specific to BA on finite inputs described by Giannakopoulou and Havelund in [18], which is based on earlier work [13, 17, 42] on converting LTL to a form of BA for the purposes of model checking. I do not prove the correctness of this algorithm, for this the reader is referred to [18].

State	Transition	
	Source	Guard
q_0	$\varnothing$	
q_1	q_0	$\neg$FM
	q_1	$\neg$FM
q_2	q_0	$\neg$CRG
	q_1	$\neg$CRG
	q_2	$\neg$CRG $\wedge$ $\neg$FM
	q_2	$\neg$CRG

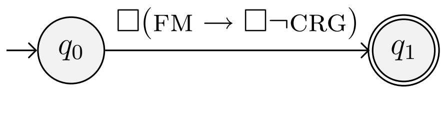
Table 3.1: This table depicts the result of compilation of Equation (3.3) into a Büchi Automaton. There are three states. Every line in the entry for a state represents a transition into that state, from the indicated state, guarded by the Guard. An equivalent automaton is shown in Figure 3-3.

However, it can be seen intuitively that this compilation maps each element of an LTL formula into an equivalent BA. Compilation will result specifically in a set of states, each with a list of transitions that lead to it. In the example given above, compilation produces the states shown in Table 3.1. Each line in the table represents a transition that is from the "Source" location, to the "State" location, and is guarded by the given "Guard".

In the following sections I describe an algorithm for generating a BA that is usable for monitoring an input formula $\mathbf{\Lambda}$. This is done through repeated decomposition of $\mathbf{\Lambda}$, described below. I also briefly discuss a post-processing step that may be performed in order to compactly encode the list of states; this results in an automata like the one shown in Figure 3-3. Section 3.4.2 presents the psuedo code for this compilation process. Section 3.4.3 provides a detailed example of compilation.

3.4.1 An Informal Description of Compilation

For monitoring purposes, it is not acceptable to have the following as an automaton for Property (3.3):



While this automaton is technically correct, it is not useful because it requires that the complete program trace be available at once in order to verify that the guard is satisfied. In contrast, the automaton in Figure 3-3 is able tell at each time step whether or not the property is currently being satisfied. Hence this automaton may be used to track the safety state of the system, as was shown in Section 3.3.2.

The key insight here is that the two safety states of the automaton in Figure 3-3 represent the two different system behaviors that satisfy Property 3.3. Safety state q_1 represents the scenario in which a fault has not been detected. If no fault is ever detected ($\Box\neg\text{FM}$), then the automaton is always in state q_1 and the property is satisfied. State q_2 represents the scenario in which a fault has been detected, but control has not been granted. If control is never granted ($\Box\neg\text{CRG}$), then the automaton is always in state q_2 and the property is satisfied. The goal of compilation is to identify these distinct behaviors that satisfy the input formula $\mathbf{\Lambda}$ and record these behaviors as separate states in the automaton.

Therefore we decompose the input formula by creating *nodes* or graph nodes that represent the different ways a formula may be satisfied, allowing us to associate states - and therefore label transitions - with conjunctions of literals rather than more complex temporal formulae. Once a node has been completely processed, it either becomes a distinct state in the automaton, is folded into an existing state, or is discarded due to a logical contradiction. After every node has been processed, all that remains is to identify the accepting states.

Nodes

A node N has the following fields:

NAME: The unique name of the node.

INCOMING: A list of nodes that have transitions to this node.

NEW: Properties which must hold at this state, yet to be processed.

OLD: A list of properties which guard incoming transitions, already processed.

NEXT: Properties to hold at immediately subsequent states.

A node N consists of a unique name, a field called INCOMING that contains a record of every node that may transition to N, and three fields that contain LTL formulae. First is the NEW field, which contains a set of LTL properties, yet to be processed, that must hold at N. After formulae in NEW have been broken down into literals, they are placed in the OLD field. These literals guard the transition from the INCOMING node to node N. While a node is being processed, it only has one INCOMING node, but after a node is added to the set of states it may have multiple INCOMING transitions, and so OLD may then contain sets of literals, one set to guard each transition. Finally, the NEXT field contains LTL formulae that must hold at any state immediately following the current.

I denote nodes as:

$$N_{name}\big(\{N_{incoming}\}, \{\Lambda_{new}\}, \{\Lambda_{old}\}, \{\Lambda_{next}\}\big)$$

Initialization

We begin compilation by creating an initial node N_0 to be the start state of the automaton:[4]

$$N_0\big(\varnothing, \varnothing, \varnothing, \varnothing\big)$$

This node, a convenience node for identifying the start states of the automaton, does not need to be processed and is added directly to the set of states. Next, we create the node that will contain the input formula and add it to the list of nodes to be processed:

$$N_{input}\big(N_0, \mathbf{\Lambda}_{input}, \varnothing, \varnothing\big)$$

Now the goal is to process every node by repeatedly decomposing the formulae in the NEW field. This decomposition is accomplished by identifying the distinct ways in which a formula may be satisfied and by creating new nodes to deal with different satisfaction scenarios. A node is fully processed when its NEW field is empty.

[4]In this section, subscripts such as N_0 are used merely as an indexing number, *the initial node*, rather than a time index, *the node N at time 0.*

Compilation ends when all nodes have been fully processed.

Processing Nodes

Our goal in processing a certain node N_1 is to reduce all formulae in its NEW field to literals. If a formula is encountered during processing that may be satisfied in more than one way, then a new node N_2 is created so that together, N_1 and N_2 represent all possible ways that the formula may be satisfied. Any literals remaining in a node's OLD field after processing represent the literals that must hold at that node of the automaton.

Spatial Processing Some formulae may be satisfied in two *spatially* distinct ways, meaning that there two distinct assignments to literals at a certain time that will satisfy the formula. For instance, the formula $\alpha \vee \beta$ may be true if either α or β is currently true. If we remove this formula from the NEW field of node N_1 during processing, we have N_1 assert that α is true, and we also create a new node N_2 that asserts β rather than α. Because α and β might be formulae requiring additional decomposition, we put α back into the NEW field of N_1, and β into the NEW field of the new node, N_2.

$$N_1\big(N_i, \boxed{\alpha \vee \beta}, \Lambda_{old}, \Lambda_{next}\big) \quad \Rightarrow \quad \begin{array}{l} N_1\big(N_i,\ \alpha\ , \Lambda_{old}, \Lambda_{next}\big) \\ N_2\big(N_i,\ \beta\ , \Lambda_{old}, \Lambda_{next}\big) \end{array}$$

Notice that all other fields remain the same, including the incoming field.

This decomposition may be thought of in terms of the automaton as well. Before decomposition, node N_1 had an incoming transition guarded by $\alpha \vee \beta$, and by Λ_{old}.

$$\begin{tikzcd} q_i \arrow[r, "(\alpha \vee \beta) \wedge \Lambda_{old}"] & N_1 \end{tikzcd}$$

After decomposition, there exist two nodes, each with a non-deterministic incoming transition guarded by either α or β, and Λ_{old}.

$$\begin{tikzcd} & N_1 \\ q_i \arrow[ur, "\alpha \wedge \Lambda_{old}"] \arrow[dr, "\beta \wedge \Lambda_{old}"'] & \\ & N_2 \end{tikzcd}$$

Temporal Processing A formula may be satisfied in two *temporally* distinct ways, meaning that there are two classes of assignments to literals across time that will satisfy the formula. These types of formulae will be handled differently; temporal obligations will be pushed down to the next node. For example, consider the case in which we encounter the formula $\alpha\mathcal{U}\beta$ while processing a node N_1. The decomposition of this property relies on the following identity:

$$\alpha\mathcal{U}\beta \equiv \beta \vee \big[\alpha \wedge \bigcirc(\alpha\mathcal{U}\beta)\big] \tag{3.4}$$

This equivalence states "either β is true now, or α is true now and $\alpha\mathcal{U}\beta$ is true at the next time step." The two halves of the disjunction represent all ways in which $\alpha\mathcal{U}\beta$ may be satisfied. If β is true now, then $\alpha\mathcal{U}\beta$ is satisfied for all time. If α is true now then $\alpha\mathcal{U}\beta$ has not been violated, yet has not been permanently satisfied; therefore, it must be enforced again at the next time step.

Following the algorithm for the $\alpha \vee \beta$ case, we create a new node N_2 to enforce the first half of the disjunction in Equation (3.4), and leave N_1 to enforce the second half. The obligation represented by the "next" ($\bigcirc$) portion of the formula is pushed down to subsequent nodes by adding it to the NEXT field of node N_1.

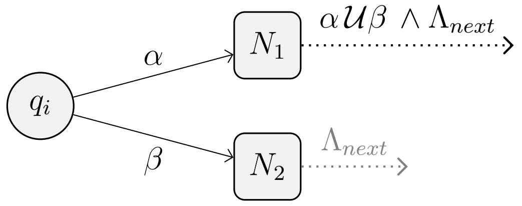
$$N_1\big(N_i, \boxed{\alpha\mathcal{U}\beta}, \varnothing, \Lambda_{next}\big) \quad \Rightarrow \quad \begin{array}{l} N_1\big(N_i,\ \alpha\ , \varnothing, \Lambda_{next} \bigcup(\alpha\mathcal{U}\beta)\big) \\ N_2\big(N_i,\ \beta\ , \varnothing, \Lambda_{next}\big) \end{array}$$

When a node N_1 is turned into a state, a new node is created to handle any unfulfilled temporal obligations in its NEXT field. This is discussed in the next section.

In terms of automata, this decomposition looks very similar to the case of a disjunction.

$$\begin{tikzcd} q_i \arrow[r, "\alpha\mathcal{U}\beta"] & \boxed{N_1} \end{tikzcd}$$

Two nodes exist after decomposition, each with a nondeterministic incoming transition guarded by either α or β. Additionally, node N_1 records the fact that subsequent nodes must enforce $\alpha\mathcal{U}\beta$.



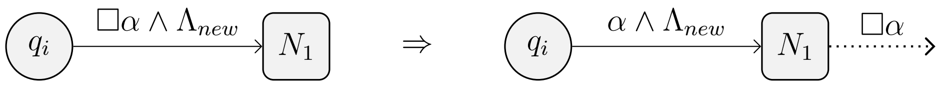
Next consider the formula $\Box\alpha$. If we encounter this property while processing node N_1, we decompose it according to the following:

$$\Box\alpha \equiv \alpha \wedge \bigcirc(\Box\alpha)$$

This equivalence states that "at the current time, α must be true, and starting at the next time, α must always be true." A new node is not needed spatially because there are not two ways *at one time* that this formula may be satisfied. Therefore, a new node is not created. However, unfilled temporal obligations exist. These are pushed down to subsequent nodes. α may require more processing, hence it is added back to the list of properties in the NEW field.

$$N_1\big(N_i, \{\boxed{\Box\alpha}, \Lambda_{new}\}, \varnothing, \varnothing\big) \quad \Rightarrow \quad N_1\big(N_i, \{\,\alpha\,, \Lambda_{new}\}, \varnothing, \varnothing \bigcup \Box\alpha\big)$$

Considered graphically, the transition between node N_i and N_1 is simplified, and N_1 records the temporal obligation:


q_i
$\Box\alpha \wedge \Lambda_{new}$
N_1
$\Rightarrow$
q_i
$\alpha \wedge \Lambda_{new}$
N_1
$\Box\alpha$

A similar case analysis can be performed for each type of LTL operator. The rules for all operators are summarized in Table 3.2. In this table we see both halves of a conjunction must hold at a node, and so both are processed separately for the node. The release operator $\mathcal{R}$ is decomposed in almost the same manner as $\mathcal{U}$. Finally, the operators $\rightarrow$ and $\Diamond$ are translated into equivalent statements using $\vee$ and $\mathcal{U}$ respectively, from which decomposition proceeds as already described.

As formulae are processed, any literals are put directly into the OLD set. If ever

Table 3.2: Rules for decomposing an LTL formula f that was taken from the NEW field of a node N_1. For each type of formula, corresponding values indicated for NEW and NEXT are added to the existing NEW and NEXT sets of N_1. If a NEW entry exists for N_2, then a new node is cloned from (the original) N_1 and augmented with the NEW value for N_2.

f	NEW N_1	NEXT N_1	NEW N_2
$\alpha \land \beta$	$\{\alpha, \beta\}$	$\varnothing$	
$\alpha \rightarrow \beta$	$\neg\alpha \lor \beta$	$\varnothing$	
$\Diamond\alpha$	$\mathbf{T}\mathcal{U}\alpha$	$\varnothing$	
$\Box\alpha$	α	$\Box\alpha$	
$\alpha \lor \beta$	α	$\varnothing$	β
$\alpha\mathcal{U}\beta$	α	$\alpha\mathcal{U}\beta$	β
$\alpha\mathcal{R}\beta$	β	$\alpha\mathcal{R}\beta$	$\{\alpha, \beta\}$

there is a contradiction in OLD, that node is discarded.[5] When the NEW field of a node is empty, it has been fully processed.

Adding Nodes to the Set of States

After a node N_1 has been fully processed, it may be folded into an existing state or added to the set of states as a unique new member. A node N_1 is folded together with a state q_1 if the NEXT field of N_1 is equivalent to the NEXT field of q_1. Otherwise, the node is not equivalent to any existing state and is added as a new state. In either case, the addition of a node to the set of states adds a new transition to the automaton, which is guarded by the set of literals in the OLD field of N_1.

Adding New States When a node N_1 is added to the set of states, an incoming transition is created from the state N_i in INCOMING. This transition is guarded by the set of literals in OLD, as they represent everything that is true at N_1. If the OLD field is empty ($\varnothing$), then the transition is *unguarded*, or equivalently, guarded by $\mathbf{T}$. To enforce the temporal obligations represented in the NEXT field, a new node N_2 is

[5]The OLD field contains the literals that must be true in order to be in the current state, so if there are contradictions in the OLD field, then we can never reach this state, and thus we can safely discard it.

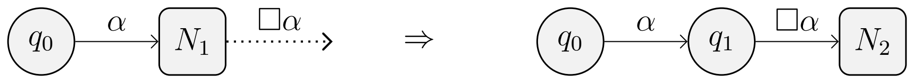
created that follows N_1 and is added to the queue of nodes waiting to be processed:

$$N_2\big(N_1, \Lambda_{N_1\text{NEXT}}, \varnothing, \varnothing\big)$$

For example, consider a node $N_1(N_0, \varnothing, \alpha, \Box\alpha)$. This node is fully processed because the NEW field is empty, and so we add it to the set of states. We then create a node N_2 as described above:

$$N_2\big(N_1, \Box\alpha, \varnothing, \varnothing\big)$$

Graphically:



Folding States Together Because two nodes with identical NEXT fields represent equivalent states[6], we do *not* add a new state to the automaton if one already exists with the same NEXT field.[7] Instead, we collapse the states together. For example, if node $N_2\big(N_j, \varnothing, \beta, \Box\alpha\big)$ has the same NEXT field as the already existing state q_1 that was created from the node $N_1\big(N_i, \varnothing, \alpha, \Box\alpha\big)$, then we incorporate the new information by adding a new transition to state q_1. This transition will be from N_2's incoming state, to q_1, and is guarded by N_2's old field.

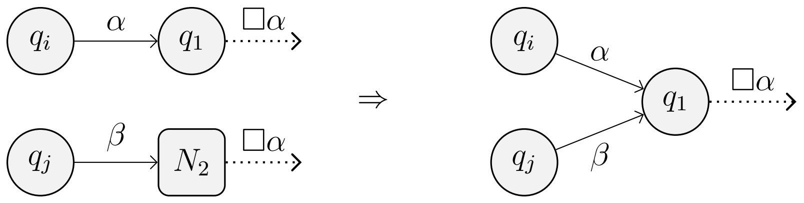


Figure 3-4: Example of folding nodes together when adding a new node to a BA under construction. After node N_2 is completely processed, it is folded into state q_1 because they have the same NEXT fields.

When a node is folded into an existing state, no new node is added to the process queue. Intuitively, this is why compilation will eventually halt.

[6]Equivalent in the sense that they encode the same temporal obligations.

[7]A node may not be equivalent to the initial state q_0, even if it has the same NEXT field ($\varnothing$).

Wrapping Up

Finally, after all nodes have been processed and all states of the automaton obtained, it remains to identify the accepting states. Intuitively, the Nondeterministic Büchi Automaton operates by rejecting immediately as soon as any safety ($\Box$) constraint is violated. Therefore any state q_s representing the fulfillment of a safety constraint is an accepting state. As long as q_s is marked, the safety constraint is being met and therefore the automaton is accepting. Compilation operates by pushing temporal obligations into the NEXT field, so any eventualities that have not been satisfied at a state will reside in the NEXT field. Additionally, these eventualities must be represented by $\mathcal{U}$ due to the decomposition rule of converting "eventually" ($\Diamond$) properties into a $\mathcal{U}$ properties. Therefore any state q_u with a formulae containing $\mathcal{U}$ in its NEXT field is not an accepting state; it represents an unfulfilled temporal obligation. This type of state is a waiting state. All other states, except for the start state, are marked as accepting states. Additionally, once construction is complete and accepting states are identified, the NEXT fields of states may be removed, as this information is only used to identify equivalent states and accepting states.

In the next section I give a formal algorithm for compilation. In Section 3.4.3 I give a detailed example of how compilation is performed on the example LTL requirement (3.3), given at the beginning of this section.

3.4.2 A Procedural Description of Compilation

In this section I present pseudocode for the algorithm described in the preceding section, for compiling an LTL statement $\mathbf{\Lambda}_{input}$ into a Büchi Automaton. See Algorithms 2 and 3 on page 59.

Algorithm 2: BA-COMPILE

The set of states Q is initialized in line 2 of BA-COMPILE and the initial state q_0 is added to Q. The queue of nodes to be processed P is initialized in line 3 and the node containing the input formula is added. The boolean *trash* (line 4) records whether or

Algorithm 2 : BA-COMPILE

```
procedure BA-COMPILE($\mathbf{\Lambda}_{input}$)
    $Q \leftarrow N_0(\varnothing, \varnothing, \varnothing, \varnothing)$
    $P \leftarrow N_{input}(N_0, \{\mathbf{\Lambda}_{input}\}, \varnothing, \varnothing)$        ▷ Create process queue and add $N_{input}$
    $trash = \mathbf{F}$
    while $P \neq \varnothing$ do                                  ▷ While process queue is not empty
        $N \leftarrow dequeue(P)$
        while $N$.NEW $\neq \varnothing$ do
            $[N_{new}, trash] \leftarrow$ EXPAND($N$)
            $enqueue(P, N_{new})$                              ▷ add $N_{new}$ to process queue, may
                                                               have no effect if $N_{new} = \varnothing$
        end while
        if $trash$ then
            continue                                           ▷ Node is discarded
        end if
        if $\exists(q \in Q)$ st $q$.NEXT $= N$.NEXT then        ▷ State $q$ is equivalent to $N$
            $q$.INCOMING $= q$.INCOMING $\cup N$.INCOMING
            $q$.OLD $= q$.OLD $\cup \{N$.OLD$\}$
        else                                                   ▷ add $N$ to states
            $add(Q, N)$
            $N_{new} = (N, N$.NEXT$, \varnothing, \varnothing)$
            $enqueue(P, N_{new})$                              ▷ add $N_{new}$ to process queue
        end if
    end while
end procedure
```

Algorithm 3 : EXPAND

```
procedure $\{N_2, trash\}$ = EXPAND($N_1$)
    $N_2 = \{\varnothing\}$
    $trash = \mathbf{F}$
    $f \leftarrow pop(N_1$.NEW$)$
    if $f$ is a literal then
        $N_1$.OLD $= N$.OLD $\cup f$
        if $N_1$.OLD is unsatisfiable then
            $trash = \mathbf{T}$
        end if
    else if $f$ is one of ($\vee$, $\mathcal{U}$, or $\mathcal{R}$) then
        $N_2 = N_1$                                            ▷ clone node $N_1$
        $[N, N_2] \leftarrow$ TABLE-3.2($f$)
    else                                     ▷ $f$ IS ONE OF ($\wedge, \rightarrow, \Diamond$, OR $\Box$)
        $N_1 \leftarrow$ TABLE-3.2($f$)
    end if
    return $\{N_2, trash\}$
end procedure
```

not a node contains a contradiction and is to be discarded.

For each node N in P, the algorithm performs EXPAND until N.NEXT is empty. EXPAND returns $\mathbf{T}$ for *trash* if N is to be discarded, and may also return a new node (line 8). If a new node is returned ($N_{new} \neq \varnothing$), then it is added to P in line 9. After a node is fully processed (line 10), then it is either discarded in line 12, folded in with equivalent state q in lines 15 and 16, or added to Q in line 18. In the last case, a new node is created to process N.NEXT in line 19, and is added to P. When all nodes have been processed (line 22), the set of states Q contains all states of the BA.

After the procedure BA-COMPILE is performed, accepting states of Q are identified as described in the preceding section, and the NEXT fields may be deleted.

Algorithm 3: EXPAND

The EXPAND procedure begins by removing its working formula f from the NEW field of the input node N_1 in line 4. If f is a literal, then no more processing is required, so f is placed in N_1.OLD (line 6) and N_1.OLD is checked for satisfiability. If a conflict is found, the *trash* flag is set to $\mathbf{T}$ (line 9), indicating that N_1 is to be discarded. If f is not a literal, then it is decomposed according to Table 3.2. If the operator of f is either $\vee$, $\mathcal{U}$, or $\mathcal{R}$, a new node N_2 is created with a unique name, and all its fields are set equal to those of N_1 in line 11. The NEW and NEXT fields of nodes N_1 and N_2 are then augmented according to the table in line 12. If f does not require a new node, then only N_1 is altered according to Table 3.2 in line 14. EXPAND returns a new node if one was created, and the *trash* flag indicating the presence or absence of contradictions in N_1.OLD.

3.4.3 An Example of Compilation

In this section I detail the compilation of an LTL property into a Büchi Automata according to Algorithm 2. Compilation is applied to the property stated in Equation (3.3), which is restated here:

$$\mathbf{\Lambda}_2 = \Box(\text{FAULT-MONITOR} = \mathbf{T} \rightarrow \Box(\text{CONTROL-REQUEST-GRANTED} = \mathbf{F}))$$

Recall that compilation of Property (3.3) produces the following list of states and transitions:

State	Transition	
	INCOMING	Guard
q_0	$\varnothing$	
q_1	q_0	$\neg$FM
	q_1	$\neg$FM
q_2	q_0	$\neg$CRG
	q_1	$\neg$CRG
	q_2	$\neg$CRG $\wedge$ $\neg$FM
	q_2	$\neg$CRG

Table 3.3: The list of states that is produced when Algorithm BA-COMPILE is performed on formula (3.3).

Beginning with Algorithm 2, we create N_0 and add it to the set of states. This node is the initial state of the automaton. For the purposes of this example, states are named after the node that they were created from, hence node N_0 becomes state q_0.

$$Q = \{\ N_0(\varnothing, \varnothing, \varnothing, \varnothing)\ \}$$

$$\begin{tikzcd} {} \arrow[r] & q_0 \end{tikzcd}$$

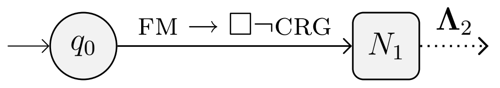
Next we create N_1 and add it to the to-process queue, per line 3 of the algorithm. This node begins with the input formula $\mathbf{\Lambda}_2 = \Box(\text{FM} \rightarrow \Box\neg\text{CRG})$ in its NEW field, and has the start state q_0 as its INCOMING state. For the purposes of this example, we depict nodes graphically as members of the automaton, though they technically do not become part of the automaton as states until they are fully processed. Transitions are guarded by the contents of the NEW and OLD fields of a node. Recall nodes are written N_{name}(INCOMING,NEW,OLD,NEXT).

$$P = \{\ N_1(N_0, \mathbf{\Lambda}_2, \varnothing, \varnothing)\ \}$$

$$\begin{tikzcd} {} \arrow[r] & q_0 \arrow[r, "\Box(\text{FM} \rightarrow \Box\neg\text{CRG})"] & N_1 \end{tikzcd}$$

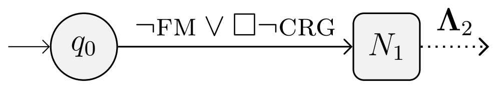
Now we EXPAND node N_1, which is the only node on the queue. First we remove $\mathbf{\Lambda}_2$ from the NEW field. This formula is of the form $\Box\alpha$, so according to Table 3.2, we put (FM $\rightarrow \Box\neg$CRG) back in the NEW field for further processing, and we put the full formula, $\mathbf{\Lambda}_2$, into the NEXT field.

$$P = \{\ N_1\big(N_0, (\textsc{fm} \rightarrow \Box\neg\textsc{crg}), \varnothing, \mathbf{\Lambda}_2\big)\ \}$$



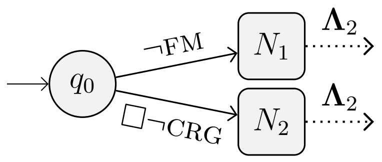
Another expansion of N_1 turns the implication into a disjunction:

$$P = \{\ N_1\big(N_0, (\neg\textsc{fm} \vee \Box\neg\textsc{crg}), \varnothing, \mathbf{\Lambda}_2\big)\ \}$$



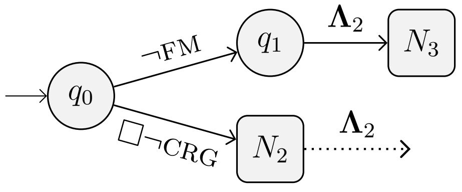
The next expansion of N_1 first removes the disjunction $f = \neg\textsc{fm} \vee \Box\neg\textsc{crg}$ from NEW. According to Table 3.2, we must create a new node to enforce the second half of the disjunction. This node, N_2, is cloned from N_1 after the removal of f, and then one half of the formula is added to each NEW field.

$$\begin{aligned} P =\{\ & N_1\big(N_0, \neg\textsc{fm}, \varnothing, \mathbf{\Lambda}_2\big), \\ & N_2\big(N_0, \Box\neg\textsc{crg}, \varnothing, \mathbf{\Lambda}_2\big)\ \} \end{aligned}$$



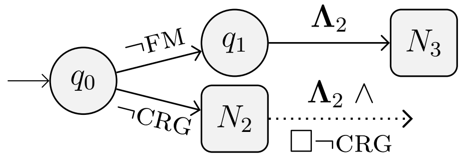
To finish processing N_1, we remove $f = \neg\textsc{fm}$ from the NEW field. Because f is a literal, we put it directly into the OLD field; no further decomposition is required. Now we may add N_1 to the set of states. Additionally, because there are unfulfilled temporal obligations in the NEXT field of N_1, we must create a new node N_3 to enforce them. This node is created downstream of N_1, with $\mathbf{\Lambda}_2$ in its NEW field.

$$\begin{aligned} Q =\{\ & N_0\big(\varnothing, \varnothing, \varnothing, \varnothing\big), \\ & N_1\big(N_0, \varnothing, \neg\textsc{fm}, \mathbf{\Lambda}_2\big)\ \} \\ P =\{\ & N_2\big(N_0, \Box\neg\textsc{crg}, \varnothing, \mathbf{\Lambda}_2\big), \\ & N_3\big(N_1, \mathbf{\Lambda}_2, \varnothing, \varnothing\big)\ \} \end{aligned}$$



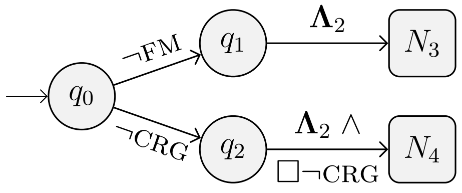
After fully processing N_1, the next node on the process queue P is N_2. We remove $f = \Box\neg\textsc{crg}$ from N_2.NEW and decompose it as before, putting $\neg\textsc{crg}$ in N_2.NEW, and all of f in N_2.NEXT. One more decomposition step moves the literal $\neg\textsc{crg}$ to the OLD field of N_2, which completes the decomposition process for this node.

$$\begin{aligned} P =\{\ & N_2\big(N_0, \varnothing, \neg\textsc{crg}, \{\mathbf{\Lambda}_2, \Box\neg\textsc{crg}\}\big), \\ & N_3\big(N_1, \mathbf{\Lambda}_2, \varnothing, \varnothing\big)\ \} \end{aligned}$$



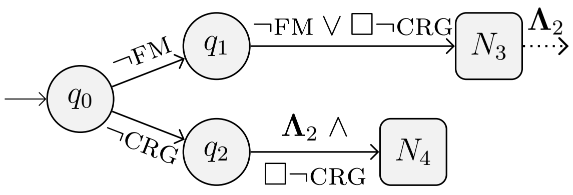
Now N_2 is fully processed. Because it has a unique NEXT field, it is also added to the set of states and we create a new node N_4 to enforce the contents of N_2.NEXT.

$$\begin{aligned} Q = \{ & N_0(\varnothing,\varnothing,\varnothing,\varnothing), \\ & N_1(N_0,\varnothing,\neg\text{FM},\mathbf{\Lambda}_2), \\ & N_2(N_0,\varnothing,\neg\text{CRG},\{\mathbf{\Lambda}_2,\Box\neg\text{CRG}\}) \ \} \\ P = \{ & N_3(N_1,\mathbf{\Lambda}_2,\varnothing,\varnothing), \\ & N_4(N_2,\{\mathbf{\Lambda}_2,\Box\neg\text{CRG}\},\varnothing,\varnothing) \ \} \end{aligned}$$



Next on the queue is N_3. This node has same content that N_1 did when it was first added to P. Processing it will have a similar effect. First, the “always” ($\Box$) obligation is pushed down to the NEXT node, and the implication becomes a disjunction.

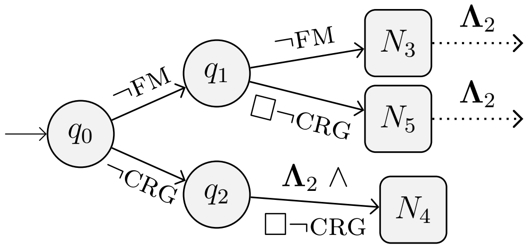
$$\begin{aligned} P = \{ & N_3(N_1,\neg\text{FM}\vee\Box\neg\text{CRG},\varnothing,\mathbf{\Lambda}_2), \\ & N_4(N_2,\{\mathbf{\Lambda}_2,\Box\neg\text{CRG}\},\varnothing,\varnothing) \ \} \end{aligned}$$



q_0
$\neg$FM
q_1
$\neg$FM $\vee$ $\Box\neg$CRG
N_3
$\mathbf{\Lambda}_2$
$\neg$CRG
q_2
$\mathbf{\Lambda}_2 \wedge \Box\neg$CRG
N_4

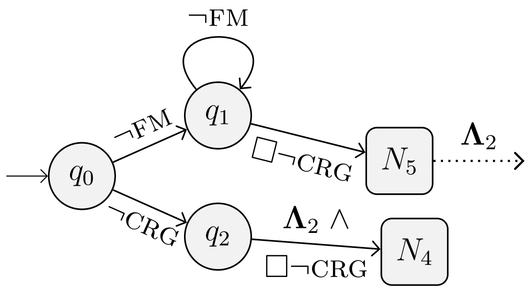
Then the node is split on the disjunction, spawning node N_5.

$$\begin{aligned} P = \{ & N_3(N_1,\varnothing,\neg\text{FM},\mathbf{\Lambda}_2), \\ & N_4(N_2,\{\mathbf{\Lambda}_2,\Box\neg\text{CRG}\},\varnothing,\varnothing) \ \} \\ & N_5(N_1,\Box\neg\text{CRG},\varnothing,\mathbf{\Lambda}_2) \ \} \end{aligned}$$



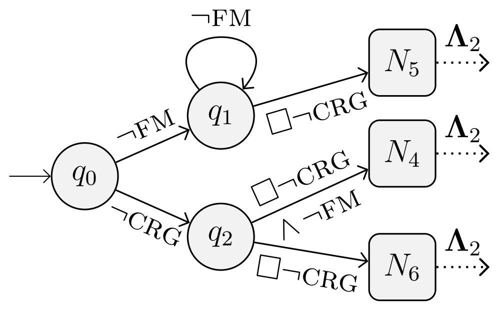
Now N_3 is fully processed. However, it is equivalent to the state q_1, as they have the same NEXT fields. Hence we fold it in with q_1, which means that we add the INCOMING and OLD fields of N_3 into those of q_1 as parallel entries. In terms of the automaton, this means that instead of a new state, we add a new transition from N_1 to N_3: a self transition.

$$\begin{aligned} Q = \{ & N_0(\varnothing,\varnothing,\varnothing,\varnothing), \\ & N_1(\{N_0,N_1\},\varnothing,\{\neg\text{FM},\neg\text{FM}\},\mathbf{\Lambda}_2), \\ & N_2(N_0,\varnothing,\neg\text{CRG},\{\mathbf{\Lambda}_2,\Box\neg\text{CRG}\}) \ \} \\ P = \{ & N_4(N_2,\{\mathbf{\Lambda}_2,\Box\neg\text{CRG}\},\varnothing,\varnothing) \\ & N_5(N_1,\Box\neg\text{CRG},\varnothing,\mathbf{\Lambda}_2) \ \} \end{aligned}$$



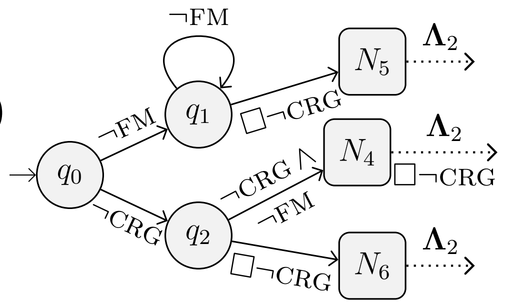
The next node, N_4, has two formulae in its NEW field. First we pop $\mathbf{\Lambda}_2 = \Box(\textsc{fm} \rightarrow \Box\neg\textsc{crg})$ off the NEW field. Similar to the other times we decomposed this formula, we push the always ($\Box$) obligation down, create a new node N_6 for half of the disjunction, and move the literal $\neg\textsc{fm}$ to N_4's OLD field.

$$\begin{aligned} P = \{\ & N_4\big(N_2, \Box\neg\textsc{crg}, \neg\textsc{fm}, \mathbf{\Lambda}_2\big) \\ & N_5\big(N_1, \Box\neg\textsc{crg}, \varnothing, \mathbf{\Lambda}_2\big)\ \} \\ & N_6\big(N_2, \{\Box\neg\textsc{crg}, \Box\neg\textsc{crg}\}, \varnothing, \mathbf{\Lambda}_2\big)\ \} \end{aligned}$$



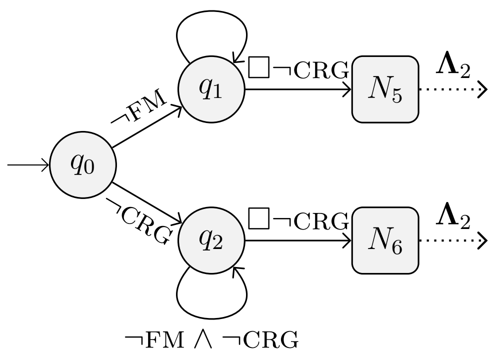
We continue processing N_4 by removing $f = \Box\neg\textsc{crg}$ from NEW, pushing the temporal obligation down, and depositing $\neg\textsc{crg}$ in the OLD field.

$$\begin{aligned} P = \{ & N_4\big(N_2, \varnothing, \{\neg\textsc{fm}, \neg\textsc{crg}\}, \{\mathbf{\Lambda}_2, \Box\neg\textsc{crg}\}\big) \\ & N_5\big(N_1, \Box\neg\textsc{crg}, \varnothing, \mathbf{\Lambda}_2\big)\ \} \\ & N_6\big(N_2, \{\Box\neg\textsc{crg}, \Box\neg\textsc{crg}\}, \varnothing, \mathbf{\Lambda}_2\big)\ \} \end{aligned}$$



Now we may add N_4 to the set of states. However, N_4 is equivalent to state q_2, so we fold N_4 and q_2 together, resulting in a self transition on q_2. Note how the list of states and their corresponding transitions, rewritten here as a table, is beginning to resemble Table 3.3.

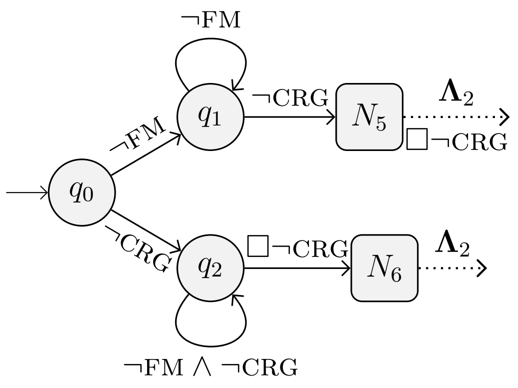
$$\begin{aligned} P = \{\ & N_5\big(N_1, \Box\neg\textsc{crg}, \varnothing, \mathbf{\Lambda}_2\big)\ \} \\ & N_6\big(N_2, \{\Box\neg\textsc{crg}, \Box\neg\textsc{crg}\}, \varnothing, \mathbf{\Lambda}_2\big)\ \} \end{aligned}$$



State	INC	OLD	NEXT
q_0	$\varnothing$		
q_1	q_0	$\neg$FM	$\mathbf{\Lambda}_2$
	q_1	$\neg$FM	
q_2	q_0	$\neg$CRG	$\{\mathbf{\Lambda}_2, \Box\neg\text{CRG}\}$
	q_2	$\neg$FM $\wedge$ $\neg$CRG	

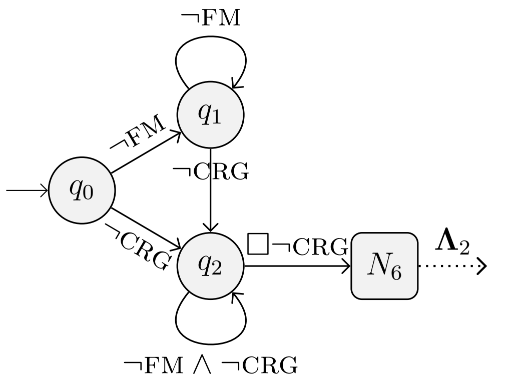
The next node on the queue for processing is N_5, which contains $\Box\neg$CRG in its NEW field. As with many nodes before it, the temporal obligation will be pushed to the NEXT node and $\neg$CRG is added to N_5.OLD:

$$\begin{aligned} P = \{\ & N_5\big(N_1, \varnothing, \neg\text{CRG}, \{\mathbf{\Lambda}_2, \Box\neg\text{CRG}\}\big)\ \} \\ & N_6\big(N_2, \{\Box\neg\text{CRG}, \Box\neg\text{CRG}\}, \varnothing, \mathbf{\Lambda}_2\big)\ \} \end{aligned}$$



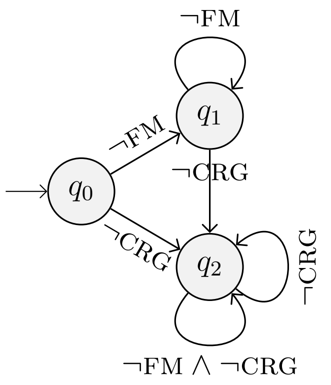
Node N_5 is equivalent to q_2. Folding it into the automaton results in a transition from q_1 to $N_5 = q_2$.

$$P = \{\ N_6\big(N_2, \{\Box\neg\text{CRG}, \Box\neg\text{CRG}\}, \varnothing, \mathbf{\Lambda}_2\big)\ \}$$

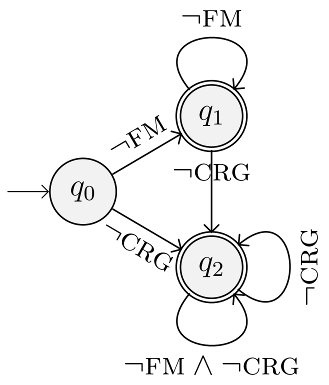


The final node, N_6, decomposes exactly as N_5 did. N_6 is also equivalent to q_2, hence results in another self transition on q_2.

State	INC	OLD	NEXT
q_0	$\varnothing$		
q_1	q_0	$\neg$FM	$\mathbf{\Lambda}_2$
	q_1	$\neg$FM	
q_2	q_0	$\neg$CRG	$\{\mathbf{\Lambda}_2, \Box\neg\text{CRG}\}$
	q_2	$\neg$FM $\wedge$ $\neg$CRG	
	q_1	$\neg$CRG	
	q_2	$\neg$CRG	



All nodes have been processed. Now we identify the accepting states by examining the content of the NEXT fields for $\mathcal{U}$ formulae. Because no states contain a $\mathcal{U}$ formula in NEXT field, none represent a liveness condition that has not been met. Consequently, both q_1 and q_2 are marked as accepting states. In general, the initial state may not be accepting in case the input formula contains eventualities [18].



This automaton is equivalent to the one presented in the BA operation example in Section 3.3.2. We see this by combining the self transitions on state q_2, and by removing the start state q_0. Combination of the transitions on q_2 is simple enough: we create a disjunction of the two guards, and then note that the sentence $(\neg\text{FM} \wedge \neg\text{CRG}) \vee \neg\text{CRG}$ can only be satisfied if CRG $= \mathbf{F}$. Removal of the start state is less trivial. This thesis does not present a general method for doing so.[8] In many cases, including this

[8]In the construction presented by Gerth *et al.* [17], the INIT state is created solely to identify the set of starting states Q_0; every state that has INIT as its INCOMING state is a member of Q_0. The INIT state is not actually a state of the automata they build. When dealing with finite traces, Giannakopoulou and Havelund [18] also discuss the removal of the start state but do not present a general method.

one, the start state may be removed and all of the states it transitions to may be marked as initial states of the automaton. In this case, marking q_2 as an initial state is unnecessary due to the configuration of the transitions, hence q_1 becomes the sole initial state.

$$\begin{tikzcd}
{} \arrow[r] & q_1 \arrow[loop above, "\neg\text{FM}"] \arrow[r, "\neg\text{CRG}"] & q_2 \arrow[loop right, "\neg\text{CRG}"]
\end{tikzcd}$$

It is worth mentioning a few decomposition cases involving the empty set $\varnothing$ that did not arise in this example.

1. If a node is added to the automaton as a state with an empty OLD field, then the transition to this state is guarded by **T**. This transition is satisfied no matter what the input to the automaton is, and so is taken automatically.

2. If a node has an empty NEXT field when being added to the automaton, it **is** equivalent to another state (not the start state) that has an empty NEXT field, and therefore if there exists such a state, the two should be folded together.

3.5 Chapter Summary

This chapter presented an automata-based approach to runtime verification. System specifications are written in Linear Temporal Logic, compiled to an equivalent Büchi Automaton, and the automaton is executed on the program trace at runtime. The automaton is accepting if the specification is being met, and rejects if the specification is violated.

The next chapter presents a novel approach to runtime verification that extends the methods in this chapter, allowing for the monitoring of stochastic, faulty hardware systems.

Chapter 4

Estimating System Safety: Runtime Verification for Embedded Systems

$\mathbf{\Phi}$	:	Plant model	$\mathbf{\Lambda}$	:	Formal safety specification
x	:	Plant (physical) state	q	:	Safety state
z	:	Observations	c	:	Commands
x_t	:	x at time t	W	:	A word legible to a Büchi Automaton
$z_{1:t}$	:	short for $\{z_1, z_2, \ldots, z_t\}$	Q_{SAFE}	:	The set of SAFE states of a DBA

4.1 Derivation of Exact Runtime Verification for Stochastic Systems

In the previous chapter I presented the solution to the problem of safety monitoring of software systems in which state can be directly observed. In this chapter I extend the problem to that of safety monitoring of mixed hardware / software systems that can fail, and I solve this problem by incorporating stochastic behavior and hidden state. I then present the exact equations needed to perform safety monitoring of these embedded systems.

4.1.1 Calculating Safety for Observable Systems

This thesis presents a capability for monitoring the safety of complex hardware/software embedded systems at runtime. Safety monitoring for purely software systems is accomplished using the technique of *runtime verification*, presented previously. As we saw in Chapter 3, runtime verification of a system involves monitoring the system online and verifying that it behaves correctly, where the definition of "correctness" is supplied as a formal specification. We also saw that we can build a capability for runtime verification when this formal specification is written in Linear Temporal Logic. This capability, denoted RV, takes as input a correctness specification $\mathbf{\Lambda}$ in LTL and an incremental program trace W at runtime. W is a sequence of letters σ_i such that each letter σ_i in W is a representation of the system state x at time i, abbreviated as x_i. At every time step, RV returns TRUE if W is consistent with $\mathbf{\Lambda}$, and FALSE if it is not.

$$\begin{tikzcd}
\mathbf{\Lambda} \arrow[r] & \boxed{\mathrm{RV}} \arrow[r] & \{1,0\} \\
& W = x_{0:t} \arrow[u] &
\end{tikzcd}$$

In Section 3.1 I described an algorithm that provides this function: LTL specifications are compiled into Nondeterministic Büchi Automata (BA), which are then run on the program trace. If any BA ever rejects, then the program is violating its corresponding safety specification and is deemed unsafe. Thus the RV capability is realized through the execution and monitoring of a BA, a function presented in Section 3.3.2.

$$\begin{tikzcd}
\mathbf{\Lambda} \arrow[r] & \boxed{\text{BA-}\textsc{compile}(\mathbf{\Lambda})} \arrow[r, "BA"] & \boxed{\text{BA-}\textsc{run}(W)} \arrow[r] & \{1,0\} \\
& & W = x_{0:t} \arrow[u] &
\end{tikzcd}$$

We can apply the same method to mixed hardware/software systems if we assume that the state of the hardware and software is directly observable. In this case, instead of running on a software trace, the BA is run on the state history for the entire hardware/software system. Practically, the difference is that the system state x is extended to include hardware as well as software state.

However, due to incomplete or faulty sensing, it is not realistic to assume that the state of an embedded system is generally observable. In the next section we consider the case in which the system state x is hidden and $\mathbf{\Lambda}$ involves these hidden states. Since the system state trajectory W can no longer be directly observed, we can no longer directly calculate the safety of the system using traditional runtime verification. Instead, we estimate the safety as a belief distribution.

4.1.2 Extension to Hidden-State

Similar to the case of an HMM, drawing the system as a time-evolving graphical model is a compact way to represent variable dependences. See Figure 4-1:

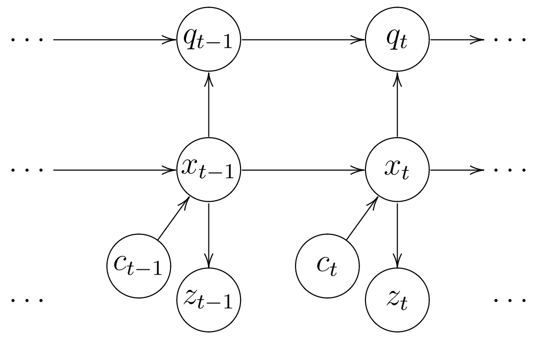


Figure 4-1: A graphical model of an embedded system. The commands into the system are represented by c, observations z, physical system (hardware *and* software) state is x, and safety state is q. Subscripts denote time.

In this graphical model, c represents commands sent to the system and z represents observations received from the system, sometimes called the evidence. The state of the physical system, including both hardware and software, is represented by x. Additionally, q is the safety state of the system, defined as the state of the BA that describes a safety constraint on the system. In a graphical model, arrows denote conditional dependencies, so x_t is conditionally dependent on c_t and x_{t-1}, but is independent of all other variables given c_t and x_{t-1}. We make the standard simplifying assumption that commands c_t are independent of previous state x_{t-1}.

Under the assumption that x is observable, it is apparent from Figure 4-1 that we have all the information needed to calculate q_t, the state of the BA at time t. However, when we generalize this model and remove the assumption that x is observable, this is no longer possible. The problem of safety monitoring can no longer be solved by runtime verification methods alone.

Instead, we want a capability that will evaluate the safety of the system given the available information: a safety specification $\mathbf{\Lambda}$, a plant model $\mathbf{\Phi}$, the control sequence $c_{1:t}$, and observation sequence $z_{1:t}$.[1] Because we can no longer estimate q_t precisely, we instead estimate the *probability* that the system remains consistent with $\mathbf{\Lambda}$, that is, the probability that the system is SAFE.

$$\begin{tikzcd} \mathbf{\Lambda}, \mathbf{\Phi} \arrow[r] & \boxed{?} \arrow[r] & \mathbf{P}(\textsc{safe}) \\ & c_{1:t}, z_{1:t} \arrow[u] & \end{tikzcd}$$

Let Q denote the set of states of the Deterministic Büchi Automaton (DBA) for $\mathbf{\Lambda}$, and let $Q_{\textsc{safe}}$ denote the set $Q/q_{\varnothing}$. That is, $Q_{\textsc{safe}}$ is the set Q with the trap state $q_{\varnothing}$ removed. This trap state is discussed at the end of Section 4.2.3. The probability $\mathbf{P}(\textsc{safe})$ is then equivalent to the probability of being in a SAFE state of the BA at time t:[2]

$$\mathbf{P}(\textsc{safe}) = \mathbf{P}(q_t \in Q_{\textsc{safe}})$$

This probability can be derived from the probability distribution over states q of the DBA at time t, given the commands and observations, by summing over the SAFE states $Q_{\textsc{safe}}$:

$$\mathbf{P}(\textsc{safe}) = \sum_{q^j \in Q_{\textsc{safe}}} \mathbf{P}(q_t^j | z_{1:t}, c_{1:t}) \tag{4.1}$$

Thus the problem of stochastic safety monitoring of embedded systems reduces to the problem of finding the probability distribution over BA states q, conditioned on the history of observations and commands. This probability distribution over q is often

[1]Here subscripts denote time, hence x_t denotes x at time t, and $z_{1:t}$ is the vector of z's from time 1 to t.

[2]Summing over all states of the automaton except the trap state is necessary for the correct monitoring of liveness conditions.

called a *belief state*, hence we abbreviate it as $\mathbf{B}(q_t)$.

The remainder of this chapter is concerned with the calculation of $\mathbf{B}(q_t)$. Section 4.2 derives an expression for $\mathbf{B}(q_t)$ in terms of the system model and the DBA characteristics. Section 4.3 goes into more detail on the system model.

4.2 Calculating Safety Belief

This section presents a derivation for

$$\mathbf{B}(q_t) = \mathbf{P}(q_t | z_{1:t}, c_{1:t}) \tag{4.2}$$

the belief state over states q of the Büchi Automaton at time t.

An intuitive but inefficient solution is presented first, followed by a more elegant yet abstract solution in Section 4.2.3.

4.2.1 The Folly of Ignoring Conditional Dependencies

Given that we have mature capabilities for estimating system state on an appropriate and expressive plant model [28, 40], one might hope that such a capability could be incorporated directly into the safety estimator developed for this thesis. Unfortunately, this is not the case. In this section we briefly consider why.

Assume we are given the belief $\mathbf{B}(x_t) = \mathbf{P}(x_t | z_{1:t}, c_{1:t})$. Additionally, assume $\mathbf{P}(q_t | x_t, q_{t-1})$, the DBA transition probability, is known. One might hope that $\mathbf{B}(q_t)$ could be obtained recursively as follows:

$$\mathbf{B}(q_t) = \sum_{x_t} \sum_{q_{t-1}} \mathbf{P}(q_t | x_t, q_{t-1}) \, \mathbf{B}(x_t) \, \mathbf{B}(q_{t-1}) \tag{4.3}$$

However, an attempt to derive Equation (4.3) from $\mathbf{B}(q_t) = \mathbf{P}(q_t | z_{1:t}, c_{1:t})$ fails. First, summing over physical state x_t and previous safety state q_{t-1} gives Equation (4.4), applying the Chain Rule gives (4.5), and noting conditional independencies yields

(4.6):

$$\mathbf{P}(q_t|z_{1:t}, c_{1:t}) = \sum_{x_t} \sum_{q_{t-1}} \mathbf{P}(q_t, x_t, q_{t-1}|z_{1:t}, c_{1:t}) \tag{4.4}$$

$$= \sum_{x_t} \sum_{q_{t-1}} \mathbf{P}(q_t|x_t, q_{t-1}, z_{1:t}, c_{1:t})\mathbf{P}(x_t|q_{t-1}, z_{1:t}, c_{1:t})\mathbf{P}(q_{t-1}|z_{1:t}, c_{1:t}) \tag{4.5}$$

$$= \sum_{x_t} \sum_{q_{t-1}} \mathbf{P}(q_t|x_t, q_{t-1})\mathbf{P}(x_t|q_{t-1}, z_{1:t}, c_{1:t})\mathbf{P}(q_{t-1}|z_{1:t}, c_{1:t}) \tag{4.6}$$

Equation (4.6) is not in the recursive form of a belief state update equation as the previous belief $\mathbf{B}(q_{t-1}) = \mathbf{P}(q_{t-1}|z_{1:t-1}, c_{1:t-1})$ does not appear. Neither does the known quantity $\mathbf{B}(x_t)$ appear. In order to equate Equations (4.6) and (4.3), the following assumptions must be made:

1. $\mathbf{P}(x_t|q_{t-1}, z_{1:t}, c_{1:t}) \approx \mathbf{P}(x_t|z_{1:t}, c_{1:t}) = \mathbf{B}(x_t)$

2. $\mathbf{P}(q_{t-1}|z_{1:t}, c_{1:t}) \approx \mathbf{P}(q_{t-1}|z_{1:t-1}, c_{1:t-1}) = \mathbf{B}(q_{t-1})$

Figure 4-1 shows that these assumptions are incorrect for our system, therefore the approach to safety estimation represented by Equation (4.3) is flawed.

In order to preserve the conditional dependencies illustrated in Figure 4-1 while also separating estimation of the plant state x from estimation of the safety state q, we need to keep a record of the history of state trajectories $x_{0:t}$, as seen in the next section.

4.2.2 A Slow Approach: Counting Every Trajectory

Alternatively, the desired safety monitoring capability could build on mature capabilities for calculating a belief over system state trajectories, rather than system state. That is, we assume that $\mathbf{B}(x_{0:t}) = \mathbf{P}(x_{0:t}|z_{1:t}, c_{1:t})$ is known. While this second approach is technically correct, tracking trajectory history is expensive. This section presents one approach for doing so.

To obtain a relation for $\mathbf{B}(q_t)$, first we sum over all system state trajectories $x_{0:t}$ in Equation (4.7), then apply the Chain Rule in Equation (4.8). Using Figure 4-1, we note that q_t is conditionally independent of commands and observations, given a state history $x_{0:t}$, hence Equation (4.9). We apply (4.1) to obtain the final relation (4.10).

$$\mathbf{P}(q_t|z_{1:t}, c_{1:t}) = \sum_{x_{0:t} \in \mathbf{X}_{0:t}} \mathbf{P}(q_t, x_{0:t}|z_{1:t}, c_{1:t}) \tag{4.7}$$

$$= \sum_{x_{0:t} \in \mathbf{X}_{0:t}} \mathbf{P}(q_t|x_{0:t}, z_{1:t}, c_{1:t})\mathbf{P}(x_{0:t}|z_{1:t}, c_{1:t}) \tag{4.8}$$

$$= \sum_{x_{0:t} \in \mathbf{X}_{0:t}} \mathbf{P}(q_t|x_{0:t})\mathbf{P}(x_{0:t}|z_{1:t}, c_{1:t}) \tag{4.9}$$

$$\mathbf{P}(\textsc{safe}) = \sum_{q^j \in Q_{\textsc{safe}}} \sum_{x_{0:t} \in \mathbf{X}_{0:t}} \mathbf{P}(q_t^j|x_{0:t})\mathbf{P}(x_{0:t}|z_{1:t}, c_{1:t}) \tag{4.10}$$

Intuitively, to calculate $\mathbf{P}(\textsc{safe})$ by Equation (4.10), we count all of the state trajectories $x_{0:t} \in \mathbf{X}_{0:t}$ that terminate in SAFE states $q \in Q_{\textsc{safe}}$ and weight each trajectory by its likelihood $\mathbf{P}(x_{0:t}|z_{1:t}, c_{1:t})$. However, while this approach is consistent with the conditional dependencies illustrated in Figure 4-1, the cost of enumerating all feasible state trajectories increases exponentially with time, a rather undesirable quality.

To address this problem we note that the probability $\mathbf{P}(q_t, x_{0:t}|z_{1:t}, c_{1:t})$ from Equation (4.7) can be viewed as the belief over a combined system state $< q_t, x_{0:t} >$. This combined state actually records more information than is necessary for safety estimation. Analysis of the graphical model in Figure 4-1 reveals that there is a cheaper combination of system states that will record the necessary information. We exploit this analysis in the next section.

4.2.3 A Good Approach: Combining Physical and Safety States

As is apparent in the graphical model of Figure 4-1, what makes this estimation problem computationally expensive are the undirected cycles in the graph. Because the physical state x and safety state q at each time are interdependent,[3] they cannot

[3]That is, they are not d-separated given observations and commands.

be estimated separately without expense, as in Equation (4.10). If we do not insist that the plant state x and the safety state q are estimated separately, then we can remove these cycles and therefore estimate over a much smaller space than $< q_t, x_{0:t} >$.

Let y_t represent the complete system state $< q_t, x_t >$ and let $\mathbf{B}(y_t)$ denote the belief over y at time t, that is $\mathbf{B}(y_t) = \mathbf{P}(q_t, x_t|z_{1:t}, c_{1:t})$. The graphical model in Figure 4-1, viewed in terms of y, is equivalent to a canonical hidden Markov model:

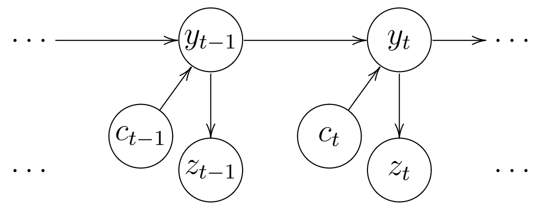


Figure 4-2: Graphical model from Figure 4-1 with clustered state $y = q \otimes x$

The belief $\mathbf{B}(q_t)$ is obtained by marginalizing x_t out of $\mathbf{B}(y_t) = \mathbf{P}(q_t, x_t|z_{1:t}, c_{1:t})$:

$$\mathbf{B}(q_t) = \mathbf{P}(q_t|z_{1:t}, c_{1:t}) = \sum_{x_t} \mathbf{P}(q_t, x_t|z_{1:t}, c_{1:t}) \tag{4.11}$$

and $\mathbf{B}(y_t)$ is obtained through standard HMM filtering. For completeness, Equations (4.12)-(4.16) derive the Forward algorithm. Applying Bayes' rule to (4.12) yields (4.13), where η is a normalizing constant. Summing over previous state y_{t-1} gives (4.14). By conditional independence of y_t given y_{t-1} and c_t, we have (4.16):

$$\begin{aligned}
\mathbf{B}(y_t) = \mathbf{P}(y_t|z_{1:t}, c_{1:t}) &= \mathbf{P}(y_t|z_t, z_{1:t-1}, c_{1:t}) && (4.12)\\
&= \eta\mathbf{P}(z_t|y_t, z_{1:t-1}, c_{1:t})\mathbf{P}(y_t|z_{1:t-1}, c_{1:t}) && (4.13)\\
&= \eta\mathbf{P}(z_t|y_t) \sum_{y_{t-1}} \mathbf{P}(y_t, y_{t-1}|z_{1:t-1}, c_{1:t}) && (4.14)\\
&= \eta\mathbf{P}(z_t|y_t) \sum_{y_{t-1}} \mathbf{P}(y_t|y_{t-1}, z_{1:t-1}, c_{1:t})\mathbf{P}(y_{t-1}|z_{1:t-1}, c_{1:t}) && (4.15)
\end{aligned}$$

$$\mathbf{B}(y_t) = \eta\mathbf{P}(z_t|y_t) \sum_{y_{t-1}} \mathbf{P}(y_t|y_{t-1}, c_t)\, \mathbf{B}(y_{t-1}) \tag{4.16}$$

Equation (4.16) computes the belief state over the combined system state y, which can also be thought of as the combined BA / PHCA state. To obtain a relation in terms of functions specified by these models, we manipulate (4.16) further by expanding y in the observation probability $\mathbf{P}(z_t|y_t)$ and the transition probability $\mathbf{P}(y_t|y_{t-1}, c_t)$, giving us (4.17). Applying the Chain Rule and simplifying based on conditional independence arguments yields (4.18):

$$\mathbf{B}(y_t) = \eta \mathbf{P}(z_t|q_t, x_t) \sum_{y_{t-1}} \mathbf{P}(q_t, x_t|q_{t-1}, x_{t-1}, c_t)\, \mathbf{B}(y_{t-1}) \tag{4.17}$$

$$= \eta\, \mathbf{P}(z_t|x_t) \sum_{y_{t-1}} \mathbf{P}(q_t|x_t, q_{t-1})\mathbf{P}(x_t|x_{t-1}, c_t)\ \mathbf{B}(y_{t-1}) \tag{4.18}$$

Substituting Equation (4.18) into (4.11) produces the following, where η is a normalization constant:

$$\mathbf{B}(q_t) = \eta \sum_{x_t} \mathbf{P}(z_t|x_t) \sum_{y_{t-1}} \mathbf{P}(q_t|x_t, q_{t-1})\mathbf{P}(x_t|x_{t-1}, c_t)\ \mathbf{B}(y_{t-1}) \tag{4.19}$$

Equation (4.19), which computes the belief state over the BA, is similar to the standard Forward algorithm for HMM belief state update (4.16). First, the next state is stochastically predicted based on each previous belief $\mathbf{B}(y_t)$ and on the transition probabilities of the models, then this prediction is corrected based on the observations received. An additional sum marginalizes out x_t, and the result is normalized by η. The observation probability $\mathbf{P}(z_t|x_t)$ and the transition probability $\mathbf{P}(x_t|x_{t-1}, c_t)$ are both functions of the model of the physical system. Section 4.3 discusses the calculation of these probabilities in the case that the plant is modeled as a Probabilistic Hierarchical Constraint Automaton. The transition probability $\mathbf{P}(q_t|x_t, q_{t-1})$ is a function of the safety specification, and is computed according to (4.21), given in the next subsection.

The cost of computing (4.19) is entirely dependent on the sizes of Q and X. In order to find the probability of each q_t, we must loop twice over these sets. If n is the size of the combined set, $n = |Q \times X|$, then we have a time complexity of $O(n^2)$, and a space complexity of $O(n)$.

Finally, given the belief state determined by Equation (4.19), the probability that the system is currently SAFE is given by:

$$\mathbf{P}(\text{SAFE}) = \sum_{q_t \in Q_{\text{SAFE}}} \eta \sum_{x_t} \mathbf{P}(z_t|x_t) \sum_{y_{t-1}} \mathbf{P}(q_t|x_t, q_{t-1})\mathbf{P}(x_t|x_{t-1}, c_t)\ \mathbf{B}(y_{t-1}) \tag{4.20}$$

Deterministic Büchi Automata

The value of the transition probability $\mathbf{P}(q_t|x_t, q_{t-1})$ from Equation (4.19) depends on the transition function T of the underlying Büchi Automaton for the safety requirements. However, for a nondeterministic Büchi Automaton (NBA), T does not represent a true probability distribution because $\sum_{\sigma \in \Sigma} T(q, \sigma) \neq 1$. One way to address this is to convert the NBA to a Deterministic Büchi Automaton (DBA) for the purposes of estimation.

It is known that canonical DBA on infinite inputs are not as expressive as their nondeterministic counterparts (see [4], page 188), and therefore canonical NBA cannot generally be converted to DBA. However, the NBA considered in this thesis are modified to accept finite traces. For finite traces, these automata are equivalent to nondeterministic finite automata (NFA) [18], which can be converted to an equivalent deterministic finite automaton (DFA) without loss of expressiveness. This conversion, known as subset construction or powerset construction, works by creating a state in the DFA for every possible combination of states in the NFA [35].

After conversion, a DBA contains a special state $q_{\varnothing}$ that represents a violation of the safety requirements. This state is the only state of the automaton that is not SAFE, therefore the probability that the system is SAFE is found by summing the likelihood of all other states, as Equation (4.1) shows. This 'trap' state $q_{\varnothing}$ represents the empty configuration of the NBA or the configuration in which no states are marked, and has a TRUE self-loop, representing the fact that a failed computation on an NBA cannot restart.

The transition probability $\mathbf{P}(q_t|x_t, q_{t-1})$ in Equation (4.19) can be obtained from the transition function T_D of the specified **Deterministic** Büchi Automaton as follows:

$$\mathbf{P}(q_t|x_t, q_{t-1}) = \begin{cases} 1 & \text{if } T_D(q_{t-1}, x_t) = q_t \\ 0 & \text{otherwise} \end{cases} \tag{4.21}$$

4.2.4 Section Summary

This section derived an equation for estimating the safety belief of a stochastic system. The belief is calculated according to Equation (4.19) at each time step, and the overall probability that the system is SAFE is calculated with Equation (4.20). The solution represented by these equations can be obtained at each time step for a cost in time of $O(n^2)$, where n is size of the combined state of the system $|Q||X|$.

The next section discusses the modeling formalism.

4.3 The Probabilistic Hierarchical Constraint Automata Model

This section details the Probabilistic Hierarchical Constraint Automata, a plant model suitable for representing reactive hardware/software systems.

4.3.1 Motivation for PHCA

This thesis provides a capability for the lifetime verification of embedded systems. These complex systems tend to suffer from performance degradation due to random hardware failure over their long and arduous life cycles.

In order to concisely and accurately model these mixed hardware/software systems that may fail, this thesis utilizes the Probabilistic Hierarchical Constraint Automaton (PHCA) formalism. PHCA allow for probabilistic behavior, a reasonable model of random hardware failure.

Section 4.3.2 introduces PHCA with an example. Section 4.3.3 discusses briefly the Reactive Model-based Programming Language (RMPL), which is the specification

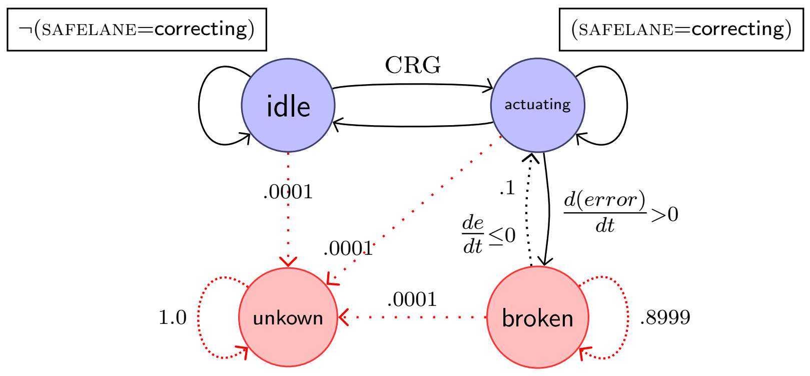


Figure 4-3: SAFELANE actuator component PHCA model.

language for PHCA. The formal definition of PHCA appears in Section 4.3.4. The final section discusses estimation on PHCA, relating the formalism to Equation (4.19).

4.3.2 PHCA Description

A *Probabilistic Hierarchical Constraint Automaton* (PHCA) [40] is an automaton that is designed to compactly and accurately model the behavior of complex systems involving both hardware and software [28]. Like an HMM, PHCA may have hidden states and transition probabilistically. Unlike an HMM, PHCA introduce the notion of constraints on states as well as a hierarchy of component automata.

Systems are modeled as a set of individual PHCA components that communicate through shared variables. Discrete modes of operation representing nominal and faulty behavior are specified for each component. Components may transition between modes probabilistically or based on system commands. Additionally, modes and transitions may be constrained by the modes of other components.

Example PHCA

In Chapter 2, the SAFELANE example system is shown modeled as a PHCA. (See Figure 2-1.) The same system could be modeled as an HMM, but the representation would be far less compact, since all possible mode combinations would have to be expanded.

As an example PHCA, an actuator component model is presented in Figure 4-3.

The actuator has two nominal and two failure modes. The nominal modes are idle and actuating. The truth of CRG and SAFELANE=correcting is assumed to be observable for the purposes of this example. Additionally, the error $e = \|\phi_{desired} - \phi_{actual}\|$, where $\phi_{desired}$ is the command sent to the actuator, is known. If the time derivative of this error is positive, then the actuator is assumed to be malfunctioning, or broken. The system may escape the broken mode, but will be assumed to be in some unknown operating mode perpetually if its behavior is ever inconsistent with its model.

4.3.3 Specifying PHCA in the Reactive Model-Based Programming Language

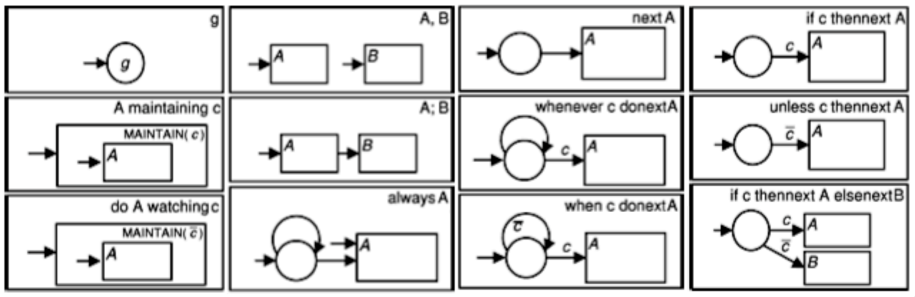


Figure 4-4: Examples of the mapping between RMPL constructs and PHCA format.

PHCA are specified in the Reactive Model-based Programming Language (RMPL) [41]. This section briefly introduces RMPL and its mapping to PHCA. See also [40,41] for more detail on PHCA construction and background on RMPL.

RMPL consists of five primitive constructs. Many other, more complex constructs can by derived from these five:

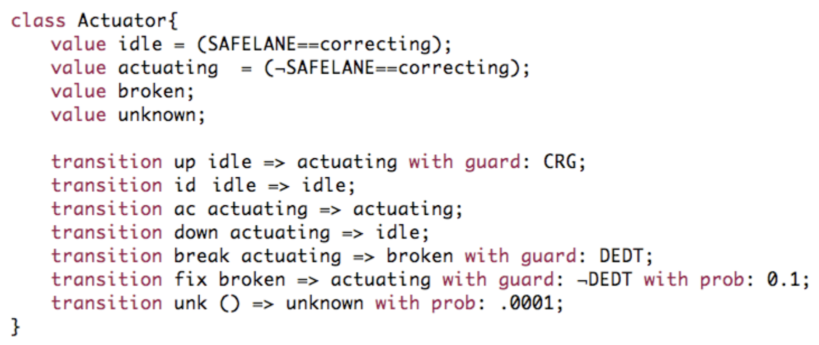
1. c : This constraint asserts that c is true at the initial time.
2. **if** c **thennext** A : This conditional branching construct causes A to begin executing at the next time step if c is currently entailed.
3. A, B : This basic concurrency construct causes both A and B to begin executing at the same time.
4. **always** A : This construct starts A at every time step

5. **choose** A **with** p, B **with** q : This is the basic probabilistic choice construct, allowing for the encoding of probabilistic knowledge about the system.

The translation of these and other combinators to PHCA is given in Figure 4-4.

Example RMPL

Below I give an example of the RMPL for the PHCA in Figure 4-3. This example is specified in RMPL$_J$, a Java-like derivative of RMPL, which includes many derived combinators. See Figure 4-5.



```
class Actuator{
    value idle = (SAFELANE==correcting);
    value actuating  = (¬SAFELANE==correcting);
    value broken;
    value unknown;

    transition up idle => actuating with guard: CRG;
    transition id idle => idle;
    transition ac actuating => actuating;
    transition down actuating => idle;
    transition break actuating => broken with guard: DEDT;
    transition fix broken => actuating with guard: ¬DEDT with prob: 0.1;
    transition unk () => unknown with prob: .0001;
}
```

Figure 4-5: Example of an RMPL$_J$ specification for a PHCA plant model.

4.3.4 PHCA Formal Definition

Formally, a PHCA is defined as a tuple: $C = \langle S, \mathbf{P}_\Theta, \Pi, \mathcal{C}, \mathbf{P}_\mathbb{T}, \mathbf{P}_G, \mathbf{P}_O \rangle$ [40].

S is the set of *locations* of the automaton, partitioned into S_p and S_c, which are the set of primitive locations and composite locations, respectively. Primitive locations have no sub-functions, whereas composite locations are sub-automata.

$\mathbf{P}_\Theta$ is the probability $\mathbf{P}(m_0^i)$, which is the probability that m^i is the initial marking of the automaton. A marking is a set of locations $m \in 2^S$, the powerset of S.

Π is a set of finite domain variables, where for each $v \in \Pi$, the domain is denoted as the set $\mathbb{D}_v$. This set includes observation variables O, dependent variables D, and control variables C. Let ϕ be a full assignment to Π, that is to say, ϕ is a set of assignments $\{v = u | \forall v \in \Pi, \text{such that } u \in \mathbb{D}_v\}$, and let Φ be the set of full assignments. Let π be a partial assignment to finite domain variables, and let the set of all partial assignments be denoted P.

$\mathcal{C} : S_p \rightarrow \mathcal{A}$ is a function that associates each primitive location $s \in S_p$ with a finite domain constraint a from the set of possible constraints $\mathcal{A}$. That is to say, a constrains the variables in its *scope* to take on only values from their domains that are allowed by a's *relation*. Constraints in this thesis are expressed in propositional state logic. A constraint $\mathcal{C}(s)$ is enforced whenever s is marked.

$\mathbf{P}_{\mathbb{T}} : S_p \times \mathbb{T} \rightarrow$ [0,1] is distribution over possible transition functions $\mathbb{T} : \mathcal{A} \rightarrow 2^S$. That is to say, a function $\mathbb{T}$ maps constraint $g \in \mathcal{A}$, also known as a *guard condition*, into a marking $m \in 2^S$. $\mathbf{P}_{\mathbb{T}}(s)$ denotes a probability distribution over the transitions associated with a primitive location s, and $\mathbf{P}_{\mathbb{T}}(s, \mathbb{T}^i)$ is the probability of the transition $\mathbb{T}^i$, which is a transition from s.

$\mathbf{P}_G : \mathcal{A} \times P \times 2^S \rightarrow$ [0,1]. This is the probability that a guard condition $g \in \mathcal{A}$ is enabled given a partial assignment π to control, observable, and dependent variables, and a marking $m \in 2^S$. If π and m are such that g is satisfied or not, then this probability is 1 or 0 respectively. In the case that some variables in the scope of g are not known, $\mathbf{P}_G$ is between 0 and 1.

$\mathbf{P}_O : O \times X \rightarrow$ [0,1] is the observation probability function. Given a state x which includes a marking m and a full assignment to D, this function returns the probability that an observation o is received, where o is a full assignment to O.

Example The example PHCA of an actuator in Figure 4-3 is formally written:

$$
\begin{aligned}
S &= \{\mathsf{idle}, \mathsf{act}, \mathsf{brk}, \mathsf{unk}\} \\
\mathbf{P}_\Theta(m) &= \begin{cases} 1 & \text{if } m = \{\mathsf{idle}\}, \\ 0 & \text{otherwise} \end{cases} \\
\mathcal{C}(s) &= \begin{cases} \text{SLC} = 0 & \text{if } s = \mathsf{idle} \\ \text{SLC} = 1 & \text{if } s = \mathsf{act} \end{cases}
\end{aligned}
\qquad
\begin{aligned}
\Pi &= \{\text{SLC}, \text{CRG}, \text{DEDT}\} \text{ where} \\
O &= \{\text{DEDT}\} \quad \mathbb{D}_{\text{DEDT}} = \{+, 0, -\} \\
C &= \{\text{CRG}\}, \quad \mathbb{D}_{\text{CRG}} = \{1, 0\} \\
D &= \{\text{SLC}\}, \quad \mathbb{D}_{\text{SLC}} = \{1, 0\}
\end{aligned}
$$

$\mathbf{P}_{\mathbb{T}}(s, \mathbb{T}^i)$	$\mathbb{T}^i$	$s \in S_p$	$g \in \mathcal{A}$	$m \in 2^S$
.9999	$\mathbb{T}_{\mathsf{idle}_n}$	idle	CRG	act
			$\mathbf{T}$	idle
.0001	$\mathbb{T}_{\mathsf{idle}_f}$	idle	$\mathbf{T}$	unk
.9999	$\mathbb{T}_{\mathsf{act}_n}$	act	$\mathbf{T}$	$\{$act, idle$\}$
			DEDT	brk
.0001	$\mathbb{T}_{\mathsf{act}_f}$	act	$\mathbf{T}$	unk
0.1	$\mathbb{T}_{\mathsf{brk}_1}$	brk	DEDT	brk
			$\neg$DEDT	act
0.8999	$\mathbb{T}_{\mathsf{brk}_2}$	brk	$\mathbf{T}$	brk
0.0001	$\mathbb{T}_{\mathsf{brk}_f}$	brk	$\mathbf{T}$	unk
1	$\mathbb{T}_{\mathsf{unk}}$	unk	$\mathbf{T}$	unk

The specification of $\mathbf{P}_G$ and $\mathbf{P}_O$ is discussed in the next section.

4.3.5 Calculating the Transition and Observation Probabilities

To perform safety monitoring, Equation (4.19) requires a transition probability and an observation probability for the plant model. This section derives these probabilities for a PHCA. The transition probability describes the likelihood of transitioning to a state x_t, given a previous state x_{t-1}. This likelihood also depends on the values assigned to finite domain variables. The observation probability describes the likelihood of observing an assignment o to O, given that x is the current state.

The Transition Probability

We are interested in knowing $\mathbf{P}(x_t|x_{t-1}, c_t)$, which is the probability of transitioning from a state x_{t-1} to a state x_t, given a control action. The state x of a PHCA consists of a marking m and a full assignment to dependent variables D, denoted d. Recall that a transition $\mathbb{T}$ is defined as a function $\mathcal{A} \to 2^S$. That is, $\mathbb{T}$ maps a guard to a marking. $\mathbf{P}_{\mathbb{T}}(s, \mathbb{T}^i)$ is the probability of the transition $\mathbb{T}^i$ from primitive location s. To compute the state transition probability, we assume that primitive transition probabilities are conditionally independent, given the current marking.[4] Hence, the composite transition probability between two markings is computed as the product of transition probabilities from each primitive location in the first marking to a subset of the second marking.

Specifically, assume we are given a previous state x_{t-1}, a current state x_t, an assignment c_t to C and z_t to O. Additionally, the state x_t is composed of a marking m_t and an assignment to dependent variables d_t. Together we denote c_t and d_t as π. Then:

$$\mathbf{P}(x_t|x_{t-1}, c_t) = \begin{cases} \prod_{s \in x_{t-1}} \mathbf{P}(s \triangleright m^i \subseteq x_t|\pi) & \text{if} \quad x_{t-1}, c_t \text{ can yield } x_t \\ 0 & \text{else} \end{cases} \tag{4.22}$$

where

$$\mathbf{P}(s \triangleright m^i \subseteq x_t|\pi) = \begin{cases} \mathbf{P}_{\mathbb{T}}(s, \mathbb{T}^i) * \mathbf{P}_G(g, \pi, m_{t-1}) & \text{if} \quad \exists \mathbb{T}^i(g) \subseteq m_t \\ 0 & \text{else} \end{cases} \tag{4.23}$$

To compute $p = \mathbf{P}(x_t|x_{t-1}, c_t)$ in equation (4), we compute the probability $n = \mathbf{P}(s \triangleright r^j \subseteq x_t|\pi)$ for each primitive location s in x_{t-1} separately. Equation (4.23) describes the computation of n: if there exists a transition $\mathbb{T}^i$ such that $\mathbb{T}^i(g) \subseteq x_t$, then n is the probability of that transition, times the probability that the guard g is satisfied, given the previous marking and π. If there exists no such transition $\mathbb{T}^i$, then p is 0. The probability p is then the product of n for all primitive locations s in x_{t-1}. However, if all satisfied transitions are insufficient to create the full marking x_t, this is, if there is a location in x_t that cannot be reached through transitions satisfied by π and m_{t-1}, then p is 0. This computation is presented formally in Algorithm 4 below.

Specifying the Guard Probability

The guard probability $\mathbf{P}_G(g, \pi, m)$ is the probability that a guard condition $g \in \mathcal{A}$ is enabled given a partial assignment π to control, observable, and dependent variables,

[4]This is analogous to the failure independence assumption from GDE and Livingstone [15], and is reasonable for most engineered systems.

Algorithm 4 : Probability computation $\mathbf{P}(x_t|x_{t-1}, c_t)$

```
1: $p = 1, m = \varnothing$
2: for each primitive location $s \in x_{t-1}$ do
3:     $n = 0$
4:     for all $\mathbb{T}^i$ from $s$ do
5:         if $\exists \mathbb{T}^i(g) = r^j | r^j \subseteq x_t$ then    ▷ If the transition contains a resultant marking $r^j$, enabled by $g$, such that $r^j \subseteq x_t$.
6:             $m \cup r^j$    ▷ Record the result in $m$.
7:             $n = \mathbf{P}_{\mathbb{T}}(s, \mathbb{T}^i) * \mathbf{P}_G(g, \pi, m_{t-1})$    ▷ Compute the prob. of taking $\mathbb{T}^i$.
8:             break    ▷ Assume there is only one such $\mathbb{T}^i$.
9:         end if
10:    end for
11:    $p = p * n$    ▷ If no transition was feasible, then $p$ becomes 0.
12: end for
13: if $m = x^t$ then    ▷ Check to see if all locations in $x_t$ can be reached.
14:     return $p$
15: else
16:     return 0
17: end if
```

and a marking $m \in 2^S$. The guard g may be written over control and dependent variables, as well as markings. For g to be enabled, it must be (1) from a primitive state $s \in m$, and (2) the constraint it represents must be satisfied by π and m. If π and m are such that the satisfaction of g is known, then this probability is 1 or 0, and is easily specified.

For example, for the actuator PHCA above:

$$\mathbf{P}_G(\textsc{crg}, \textsc{crg}{=}\mathbf{T}, \{\textsc{idle}\}) = 1$$

$$\mathbf{P}_G(\textsc{crg}, \textsc{crg}{=}\mathbf{F}, \{\textsc{idle}\}) = 0$$

$$\mathbf{P}_G(\textsc{crg}, \textsc{crg}{=}\mathbf{T}, \{\textsc{unknown}\}) = 0$$

In cases where the values of variables within the scope of the guard are not known, then the guard is indeterminate, and the guard probability is estimated in the same manner as the observation probability, described below.

Specifying the Observation Probability

Instead of specifying the entire observation probability function along with the model, we calculate approximately the observation function $\mathbf{P}_O$ for a state x_i similar to GDE [15]. Given the constraints $\mathcal{C}(s \in x_i)$ imposed by x_i, we test if each observation in o^i is entailed or refuted, giving it probability 1 or 0, respectively. If no prediction is made, then an *a priori* distribution on observables is assumed (e.g., a uniform distribution of $1/n$ for n possible values).

See [26] for a detailed discussion of this calculation.

4.4 Chapter Summary

This chapter presented an approach to runtime verification for mixed stochastic systems. Formal LTL safety specifications are converted to a Deterministic Büchi Automaton (DBA), and safety is estimated via the combined state of this automaton and the Probabilistic Hierarchical Constraint Automaton (PHCA) plant model. This approach is summarized by Equation (4.19). The details of Büchi Automata were presented in Chapter 3, and the details of PHCA were discussed in Section 4.3. The next chapter presents empirical validation and discusses future work.

Chapter 5

Validation and Conclusions

In this chapter I evaluate the utility of the monitoring algorithm presented in Chapter 4, as well as discuss future work.

Section 5.1 describes the experiment that was performed to test the validity and efficiency of the algorithm. Results are presented for the example. Section 5.2 presents possibilities for future work.

5.1 Validation

With the experiment described in this section I seek to prove that my approach provides a capability for real-time safety monitoring of complex mixed systems. In order for a safety monitoring capability to be able to operate in real time, it should:

1. Be capable of estimating on large models with reasonable speed and space usage,

2. Detect safety violations quickly, and

3. Detect safety violations accurately.

Any experiments should therefore seek to characterize the time and space usage of the algorithm as $|X|$ grows, the latency of violation detection, and the accuracy of violation detection.

The experiment performed verifies that the safety monitoring capability is able to quickly and accurately detect safety violations on a small example. Future work should more completely characterize the monitor's performance on larger models.

5.1.1 Description of Implementation

The Büchi Automaton compilation and operation, the inference algorithms, and the examples were all implemented in Java. The experiments were run on an Intel Core 2 Duo 2.16 GHz with 4 gigabytes of RAM.

5.1.2 Experimental Setup

For the experiment, I tested a small subset of the SAFELANE example from Chapter 2. This example was designed solely to validate the accuracy and speed of safety violation detection. For this model, I tested nominal scenarios and faulty scenarios, including both the case where the actuator experiences a modeled failure, and where it experiences an unmodeled failure. In both cases, the safety monitor is able to detect a safety violation.

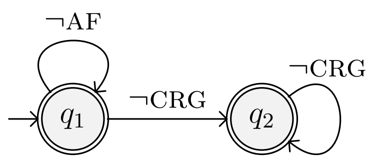
5.1.3 SAFELANE Actuator Example

In this example I monitored the following safety statement:

$$\Box\big(\textsc{actuator-fail} \rightarrow (\Box\neg\textsc{control-request-granted})\big) \tag{5.1}$$

This statement was presented in Chapter 2 as an example of a safety statement written over a hidden states of a stochastic plant. In this case, we require that if ever an actuator fails, then the autonomous subsystem should not subsequently acquire control of the vehicle. This is a good prototypical safety requirement. It is representative of the kinds of statements that could be monitored, and is of typical complexity.

The Büchi Automaton for this statement is:



In this graphic, the proposition ACTUATOR-FAIL from Statement (5.1) is abbreviated as AF, and the proposition CONTROL-REQUEST-GRANTED is abbreviated as CRG. As mentioned in Chapter 4, this nondeterministic automaton is converted into a deterministic automaton for the purposes of estimation. The automaton used in estimation is the following:

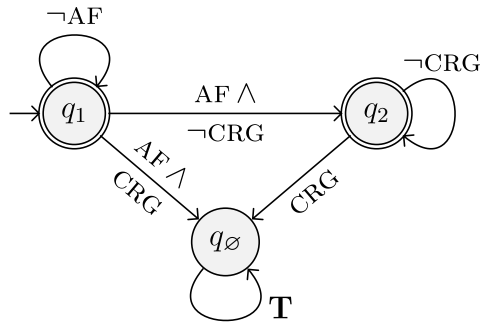


Figure 5-1: A Deterministic Büchi Automaton for testing Requirement (5.1).

PHCA Model

For this example, the actuator component of the SAFELANE system was modeled as a Probabilistic Hierarchical Constraint Automaton. (See Figure 5-2.)

The proposition AF in the above BA is true if the actuator is in either of the failure modes, which are broken and unknown. In this PHCA, the truth of CRG and SAFELANE = correcting is assumed to be observable for the purpose of this example. Additionally, the error $e = \|\phi_{desired} - \phi_{actual}\|$, where $\phi_{desired}$ is the command sent to the actuator, is known. If the time derivative of this error is positive, then the actuator is assumed to be malfunctioning, or broken. The system may escape the broken mode, but will be assumed to be in some unknown operating mode perpetually if it ever behaves inconsistently with its model.

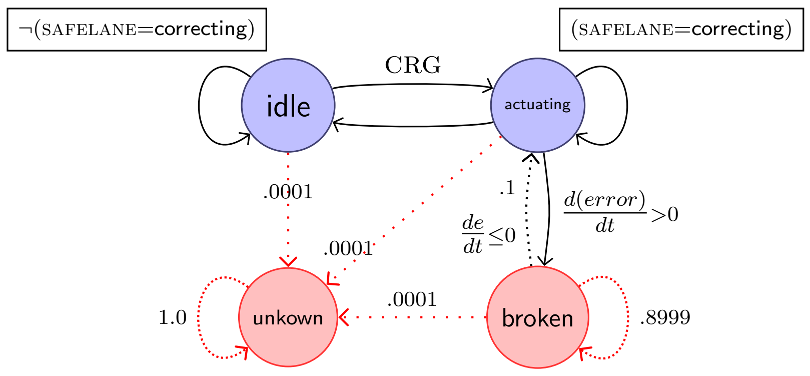


Figure 5-2: SAFELANE actuator component PHCA model.

This model exhibits probabilistic failure, as this thesis argued was necessary for lifelong verification of mixed systems.

5.1.4 Results

For this model, the nominal observation sequences were estimated as meeting requirement (5.1) with a probability greater than 99.99%. This requirement says that SAFELANE should not receive control of the vehicle if the actuator has failed. A nominal observation sequence is one in which a nominal system trajectory is consistent with the observations. An example of a nominal observation sequence is the following:

$$\vec{Z}_{nom} = \begin{array}{cccc} t=1 & 2 & 3 & 4 \\ \hline \neg\text{SLC} & \neg\text{SLC} & \neg\text{SLC} & \\ \neg\text{CRG} & \neg\text{CRG} & \neg\text{CRG} & \dots \\ \neg\frac{de}{dt}>0 & \neg\frac{de}{dt}>0 & \neg\frac{de}{dt}>0 & \end{array}$$

in which SAFELANE is never granted control and takes no corrective action.

Estimating a sequence like $\vec{Z}_{nom}$ of 200,000 input observations took 7.2 seconds[1], at which time the belief over the BA was $\mathtt{B(Q)} = \{\mathtt{q_1 = 0.99997, q_2 = 0.00003, q_\varnothing = 0.0}\}$; this is the steady-state belief for this observation sequence. The implementation runtime is constant at each time step, as expected.

Fault sequences such as the two examples below are estimated as having violated Requirement (5.1) with 100% probability. Fault sequences are those in which every

[1]For reference, the "slow" estimation described in Section 4.2.2 fails after 9 input observations due to insufficient memory space.

consistent trajectory contains a component failure.

$$\vec{Z}_{fault_1} = \begin{array}{ccc} t=1 & 2 & 3 \\ \hline \neg\text{SLC} & \text{SLC} & \text{SLC} \\ \neg\text{CRG} & \text{CRG} & \text{CRG} \\ \neg\frac{de}{dt}>0 & \neg\frac{de}{dt}>0 & \frac{de}{dt}>0 \end{array} \qquad \vec{Z}_{fault_2} = \begin{array}{cc} t=1 & 2 \\ \hline \neg\text{SLC} & \neg\text{SLC} \\ \neg\text{CRG} & \text{CRG} \\ \neg\frac{de}{dt}>0 & \neg\frac{de}{dt}>0 \end{array}$$

Many permutations of observation sequences were tested, and all faulty sequences were clearly shown to be violating the safety requirement.

This experiment showed that my approach and implementation are correct. Furthermore, I observed no latency in safety violation detection for such a model. I also observed an accuracy of safety violation detection of 1, meaning that every fault sequence was estimated as being in violation with the requirement.

5.2 Future Work

This section future work that could extend the applicability of my approach to safety monitoring.

5.2.1 Dealing with More Complex Models

Despite the encouraging results presented above, very complex systems may require models of sufficient size to render the monitoring approach presented in this thesis impractical. The estimation of system safety presented in this thesis has a time complexity that is polynomial in the number of combined system states. Specifically, the time required to update the safety estimate at each time is of order $O\big((|Q||X|)^2\big)$, where $|Q|$ is the number of states of the DBA representing the safety requirement, and $|X|$ is the number of unique configurations of the underlying physical system. For very complex physical systems requiring millions of states to be modeled accurately, this time complexity may prevent the safety monitoring capability from being useful as a real-time safety net. Additionally, the monitoring algorithm has a space complexity that is linear in the number of combined system states. For embedded systems with meager resources, the space required may be the limiting factor.

For such systems requiring large models, an approximate belief state update method may produce acceptably accurate estimates. Researchers in model-based estimation have shown examples of fast and accurate belief state update for Probabilistic Concurrent Constraint Automata (PCCA) [27] and Probabilistic Hierarchical Constraint Automata (PHCA) [40] using greedy methods. These approximations are based on a partial enumeration of the state space, and show promising results in terms of time and space savings.

If the time and space constraints of the system prevent the use of exact safety monitoring, these approximate belief state update methods provide a good starting point for an investigation into approximate safety estimation.

5.2.2 Introducting MTL or TLTL

In order to take advantage of the expressiveness of liveness properties, future work should include an investigation of timed logics. Liveness properties as defined by Alpern and Schneider [3] are those that can never be definitively violated. Liveness properties represent a large and expressive set of LTL properties, but these properties are not strictly monitorable [6]. The issue is that liveness can only proven for infinite executions. Additionally, as many authors have pointed out, liveness properties are not generally strong enough to specify the true requirements of a real-time system [29]. Despite the fact that liveness requirements are not usually strong enough and are not monitorable, the idea of liveness is important for describing the desired functionality of a reactive system.

For example, the requirement:

$$(\textsc{control-granted} \wedge \textsc{unintentional-lane-departure}) \rightarrow \Diamond\textsc{in-lane} \tag{5.2}$$

is descriptive of the desired functioning of the SAFELANE system. It is certainly true that if SAFELANE is granted control of the vehicle during an unintentional lane departure, it should eventually steer the car back into the lane. However, this requirement is not strong enough. In reality, SAFELANE should *quickly* correct the vehicle's course. It might be more accurate to require that SAFELANE complete course correction within three seconds, for example, but LTL provides no way to express this requirement.

Furthermore, when performing runtime verification on Requirement (5.2) one cannot ever make the statement that the requirement has been violated. Even if three hours have passed without the vehicle being IN-LANE since the requirement was triggered, there is still hope that the requirement will be satisfied in the future, and therefore the system behavior is not inconsistent with the requirement.

To take advantage of the expressiveness of eventualities while maintaining the strength and monitorability of safety properties, timed temporal logics such as Timed LTL [31] and Metric Temporal Logic [24] introduce time bounds on the F and U operators ($\Diamond$ and $\mathcal{U}$). With time bounds, these operators become much more useful for expressing the requirements of a real-time system. Additionally, introducing time bounds morphs a liveness statement into a monitorable form by allowing it to be disprovable with a finite execution. In other words, adding time bounds to a liveness property changes it from a property of the form *some good thing α must eventually happen*, to a property of the form *some bad thing - the passing of this finite time interval without α - must never happen.* Therefore time-bounded eventualities are actually part of the class of safety properties defined by Alpern and Schneider [3].

TLTL and MTL are similarly expressive, but syntactically different. Requirement (5.2) above could be expressed in MTL as follows:

$$\Box\big((\textsc{control-granted} \wedge \textsc{unintentional-lane-departure}) \rightarrow \Diamond_{\leq 3}\textsc{in-lane}\big)$$

where now the event IN-LANE is required to occur within 3 time units of the lane departure. The same requirement is written in TLTL as:

$$\Box\big((\textsc{control-granted} \wedge \textsc{unintentional-lane-departure}) \rightarrow \rhd_{\textsc{in-lane}} \in [0,3]\big)$$

Researchers have proven the viability of performing runtime verification using TLTL [6] and MTL [37]. Algorithms for converting both MTL and TLTL to equivalent automata on finite inputs have been shown, therefore the estimation methods in this thesis could be extended to use a timed temporal logic. The increased expressiveness of timed logics does not come without price, however, and the complexity of such an approach remains to be investigated.

5.2.3 Diagnosis

For safety monitoring to provide the most utility in a deployed system, the system should be able to react to detected violations by removing unsafe functions or curtailing mission objectives. This reactive functionality may be forced to severely restrict the system's functionality if it is unclear what components caused the safety violation. In the worst case, the system may be forced to shut down entirely.

If a safety monitoring capability were able to not only detect safety violations, but also to assign blame for the violation to one or more components, then the system would have more options when choosing a safe configuration to transition to. However, diagnosing the cause of a safety violation is an open area of research.

5.3 Chapter Summary

This chapter presented empirical results of runtime verification for mixed, faulty systems. These results showed that the approach presented in this thesis is sound and fast enough to be used in real-time for small and medium sized models. I believe that this approach to safety monitoring is valuable as a safety net for embedded systems post-deployment. Future work in this area should include investigation into using larger models, employing timed logics, and providing a diagnosis capability to supplement the safety monitoring.

Bibliography

[1] *This Car Runs on Code*, IEEE Spectrum article. Accessed online:. http://spectrum.ieee.org/green-tech/advanced-cars/this-car-runs-on-code. Last accessed May, 2010.

[2] SAFELANE, part of the PReVENT project. Website. http://www.prevent-ip.org/en/prevent_subprojects/lateral_support_driver_monitoring/safelane/. Last accessed May, 2010.

[3] Bowen Alpern and Fred Schneider. Defining liveness. Technical report, Cornell University Department of Computer Science, Ithaca, New York, October 1984.

[4] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, Cambridge, MA, 2008.

[5] F Balarin, H Hsieh, A Jurecska, L Lavagno, and A L Sangiovanni-Vincentelli. Formal verification of embedded systems based on CFSM networks. In *Proceedings of the 33rd annual Design Automation Conference*, pages 568–571, New York, NY, 1996. ACM.

[6] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for LTL and TLTL. Technical Report TUM-I0724, TU München, 2007.

[7] Jennifer Black. *System Safety as an Emergent Property*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, April 2009.

[8] G Brat, D Drusinsky, Dimitra Giannakopoulou, A Goldberg, K Havelund, M Lowry, C Pasareanu, A Venet, W Visser, and R Washington. Experimental

evaluation of verification and validation tools on martian rover software. *Formal Methods in System Design*, 25(2):167–198, 2004.

[9] M C Browne, Edmund M. Clarke, D L Dill, and B Mishra. Automatic verification of sequential circuits using temporal logic. *IEEE Transactions on Computers*, 35(12):1035–1044, 1986.

[10] J R Büchi. On a decision method in restricted second order arithmetic. In *Logic, Methodology and Philosophy of Science (Proc. of the 1960 International Congr.)*, pages 1–11, Stanford, CA, 1962. Stanford University Press.

[11] J R Burch, Edmund M. Clarke, K L McMillan, and D L Dill. Sequential circuit verification using symbolic model checking. In *Proceedings of the 27th ACM/IEEE Design Automation Conference (DAC '90)*, pages 46–51, New York, NY, 1990. ACM.

[12] J R Burch, Edmund M. Clarke, K L McMillan, D L Dill, and L Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[13] Edmund M. Clarke, Ernest A. Emerson, and A P Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[14] Séverine Coline and Loenardo Mariani. Runtime verification. In *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*, chapter 18, pages 525–555. Springer Berlin / Heidelberg, 2005.

[15] Johan de Kleer and Brian C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32:100–117, 1987.

[16] D Drusinsky. The Temporal Rover and the ATG Rover. In *SPIN Model Checking and Software Verification*, volume 1885 of *LNCS*, pages 323–330. Springer, 2000.

[17] R Gerth, D Peled, M Y Vardi, and P Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *IFIP Conference Proceedings*, volume 38, pages 3–18, 1995.

[18] Dimitra Giannakopoulou and Klaus Havelund. Automata-based verification of temporal properties on running programs. In *16th IEEE International Conference on Automated Software Engineering*, San Diego, CA, 2001.

[19] K Havelund and Grigore Roşu. Java pathexplorer - a runtime verification tool. In *The 6th International Symposium on AI, Robotics and Automation in Space*, May 2001.

[20] S K Jha, Edmund M. Clarke, C J Langmead, A Legay, A Platzer, and P Zuliani. A bayesian approach to model checking biological systems. In *Proceedings of the 7th International Conference on Computational Methods in Systems Biology (CMSB '09)*, pages 218–234, Berlin, Germany, 2009. Springer-Verlag.

[21] Hans Kamp. *Tense Logic and the Theory of Linear Order*. PhD thesis, University of California, Los Angeles, 1968.

[22] M Kim, S Kannan, I Lee, O Sokolsky, and M Viswanathan. Java-MaC: a runtime assurance approach for Java programs. *Formal Methods in System Design*, 24(2):129–155, March 2004.

[23] Moonjoo Kim and Mahesh Viswanathan. Formally specified monitoring of temporal properties. In *11th Euromicro Conference on Real-Time Systems*, 1999.

[24] Ron Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4):255–299, 1990.

[25] Orna Kupferman and Moshe Y Vardi. Model checking of safety properties. *Formal Methods in System Design*, 19:291–314, 2001.

[26] Oliver B. Martin, Seung H. Chung, and Brian C. Williams. A tractable approach to probabilistically accurate mode estimation. In *Proceedings of the 8th Interna-*

tional Symposium on Artificial Intelligence, Robotics, and Automation in Space (iSAIRAS-05), Munich, Germany, September 2005.

[27] Oliver B. Martin, Brian C. Williams, and Michel D. Ingham. Diagnosis as approximate belief state enumeration for probabilistic concurrent constraint automata. In *Proc. of the 20th National Conference on Artificial Intelligence*, pages 321–326, Pittsburgh, PA, 2005.

[28] Tsoline Mikaelian, Brian C. Williams, and Martin Sachenbacher. Model-based monitoring and diagnosis of systems with software-extended behavior. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI-05)*, pages 327–333, Pittsburgh, PA, July 2005.

[29] Dennis K. Peters and David Lorge Parnas. Requirements-based monitors for real-time systems. *IEEE Transactions on Software Engineering*, 28(2), February 2002.

[30] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (FOCS 1997)*, pages 46–57, 1977.

[31] J-F Raskin. *Logics, Automata and Classical Theories for Deciding Real-time*. PhD thesis, Namur, Belgium, 1999.

[32] U Sammapun, O Sokolsky, I Lee, and J Regehr. Statistical runtime checking of probabilistic properties. In *Proceedings of the 7th International Workshop on Runtime Verification (RV 2007)*, volume 4839 of *LNCS*, pages 164–175. Springer, 2007.

[33] A L Sangiovanni-Vincentelli, Patrick C McGeer, and Alexander Saldanha. Verification of electronic systems. In *33rd Annual Conference on Design Automation (DAC)*, pages 106–111. ACM Press, 1996.

[34] K Sen, Mahesh Viswanathan, and G Agha. Statistical model checking of black-box probabilistic systems. In *Proceedings of the 16th Intl. Conf. on Computer Aided*

Verification (CAV 2004), volume 3114 of *Lecture Notes in Computer Science*, pages 202–215, Berlin, Germany, 2005. Springer-Verlag.

[35] Michael Sipser. *Introduction to the Theory of Computation.* Thomson Course Technology, Boston, Massachusetts, 2nd edition, 2006.

[36] A P Sistla and Abhigna R Srinivas. Monitoring temporal properties of stochastic systems. In *Proc of 9th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2008)*, pages 294–308, 2008.

[37] Prasanna Thati and Grigore Roşu. Monitoring algorithms for metric temporal logic specifications. *Electronic Notes in Theoretical Computer Science*, 113:145–162, 2005.

[38] M Y Vardi. Automatic verification of probabilistic concurent finite-state programs. In *26th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 327–338. IEEE Computer Society Press, 1985.

[39] Moshe Y Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *First Symposium on Logic in Computer Science*, pages 332–344, 1986.

[40] Brian Williams, Seung Chung, and Vineet Gupta. Mode estimation of model-based programs: Monitoring systems with complex behavior. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 579–585, Seattle, WA, 2001.

[41] Brian C. Williams, Michel Ingham, Seung H. Chung, and Paul H. Elliott. Model-based programming of intelligent embedded systems and robotic space explorers. In *Proceedings of the IEEE: Special Issue on Modeling and Design of Embedded Software*, volume 91, pages 212–237, January 2003.

[42] Pierre Wolper. The tableau method for temporal logic: an overview. *Logique et Analyse*, pages (110–111):119–136, 1985.