

FORMAL VERIFICATION OF TIME-TRIGGERED SYSTEMS

Lee Pike

Submitted to the faculty of the University Graduate School
in partial fulfillment of the requirements
for the degree
Doctor of Philosophy
in the Department of Computer Science
Indiana University
December 12, 2005

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Steven D. Johnson, Ph.D.

Geoffrey Brown, Ph.D.

Lawrence S. Moss, Ph.D.

Paul S. Miner, Ph.D.

December 12, 2005

Copyright 2005
Lee Pike
ALL RIGHTS RESERVED

To Bloomington, Indiana
We parted too soon.

Acknowledgements

I thank Professor Steven Johnson for advising this work while I was in residence at the NASA Langley Research Center. He introduced me to the field of formal methods through his excellent graduate course on the subject at Indiana University, Bloomington. He taught me both how to think and write.

Paul Miner served as my “advisor in residence” at NASA. I thank him for inviting me to participate in the SPIDER project, for providing me the freedom to pursue this research, and for his guidance in doing so.

I thank the other members of the SPIDER team including Jeffery Maddalon, Alfons Geser, Wilfredo Torres-Pomales, and Mahyar Malekpour for their advice and collaboration. Their contributions to this work are inextricable.

I thank Ricky Butler, the NASA Langley Formal Methods Group leader, for providing me the opportunity to pursue this research during my employment in his group. Additionally, I benefited from many discussions with other members of NASA Langley/National Institute of Aerospace Formal Methods Group, particularly Cesar Muñoz, Ben Di Vito, Victor Carreño, and Radu Siminiceanu.

It was a pleasure to have Professors Geoffrey Brown and Lawrence Moss on my committee. In particular, Geoffrey kindly gave his time and advice in teaching me about hardware verification.

The tools employed in this work were developed at SRI, International. I thank the developers for both excellent tools and for providing guidance on using them

effectively. The problems addressed and the approach taken in this work are heavily influenced by the work of John Rushby of SRI.

I thank Gerald Allwein for his advisement early in my graduate career and for easing my transition to computer science research. I benefited from collaborations with Darren Abramson in formal methods at Indiana University (despite losing many pizzas to him in friendly theorem-proving wagers). I also thank my undergraduate advisor, Michael O'Rourke, for his continuous support.

Finally, I thank my parents for their support, my brother for showing me what is important in life, and Mr. Dacolias, the consummate teacher.

Abstract

Fault-tolerant real-time distributed control systems are being developed for next-generation aircraft and automobiles. They employ numerous complex protocols; because their uses are safety-critical, the design and implementation of these protocols must be error-free. The following modeling considerations make the formal verification of these protocols difficult: faults, real-time constraints, distributed control, nonfunctional behavioral requirements, and intricate protocol interactions. We describe a methodology for the formal verification of *time-triggered systems*, a class of synchronized fault-tolerant control and communication architectures.

The methodology centers around the distinct timing assumptions made in time-triggered systems. First, we describe a set of abstractions for specifying time-triggered protocols in an untimed synchronous model of computation that is particularly well-suited for mechanical theorem-proving. The abstractions systematically abstract faults, data, communication, and fault-masking. An untimed synchronous specification simplifies the specification and verification overhead, but a large semantic gap exists between the timing characteristics of an untimed protocol specification and its implementation. We therefore extend previous work to formally demonstrate via mechanical theorem-proving that under certain assumptions, a simulation exists between a time-triggered implementation of a protocol and its untimed synchronous specification. We then use a combination of bounded model-checking and automated solvers to verify that realized protocol schedules satisfy the necessary time-triggered assumptions. Finally, some protocols do not satisfy the time-triggered model constraints due

to the fact they execute when the system is unsynchronized, such as during startup or restart. We also use bounded model-checking and automated solvers to verify explicit real-time models of such protocols.

The methodology is demonstrated by verifying NASA Langley's SPIDER fly-by-wire bus architecture.

Contents

List of Figures	xi
Chapter 1. Introduction	1
1. Motivation and Approach	3
2. Outline	9
Chapter 2. Preliminaries & Related Work	11
1. Fault-Tolerant Distributed Systems	11
2. Time-Triggered Systems	12
3. Time-Triggered Bus Architectures	14
4. SPIDER	21
5. Time-Triggered System Verification	25
6. Tools	27
7. Timing Models	39
8. Timeout Automata: A Real-Time Model	40
Chapter 3. Synchronous Protocol Verification	43
1. The Synchronous Model	44
2. Abstracting Messages	47
3. Abstracting Faults	48
4. Abstracting Fault-Masking	52
5. Abstracting Communication	55
6. Summary	62

Chapter 4. Time-Triggered Protocol Verification	63
1. The Time-Triggered Model	66
2. Extending The Axiomatization	71
3. Schedule Verification	85
4. Summary	96
Chapter 5. Partially-Synchronous Protocol Verification	97
1. Synchronizing Timeout Automata (STA)	98
2. Case-Study: The SPIDER Reintegration Protocol	108
3. Summary	132
Chapter 6. Conclusion	135
1. Limitations	135
2. Future Work	137
3. Concluding Remarks	140
Bibliography	141
Index	154
Appendix A. Inconsistent Axioms in Rushby’s Specification	160

List of Figures

1	Verification Strategy for Time-Triggered Systems	6
1	A Generic Fault-Tolerant Bus Architecture	15
2	SPIDER Architecture	22
3	Set Definition in PVS	30
4	Interactive Proof Session in PVS (Continued in Figure 5)	32
5	Interactive Proof Session in PVS (Continued from Figure 4)	33
6	Rushby's SAL Specification of the Bakery Algorithm [1]	38
1	Abstract Messages Datatype	48
2	The Inexact Function Condition for Inexact Communication	57
3	Inexact Validity	61
1	Axiom 4.5	68
2	Pipelined Communication Phase (Axiom 4.24)	76
3	The Reception Window (Axiom 4.25)	78
4	Type and Constant Declarations	88
5	SAL Specification of the Generalized System Assumptions	89
6	A Round-Based State Machine	91
7	Clock Synchronization Event-Triggered Schedule Update	94
1	The Train-Gate-Controller	103

2	The Frame Property	112
3	State Machine Model of the Protocol Mode Control	113
4	Synchronization Frame Module	118
5	Operational Node Module	119
6	Operational Clique Module	119
7	The Reintegrator TA Misses Echo Messages	121
8	Faulty Node Module	122
9	Mode Control Module	123
10	Preliminary Diagnosis Module	125
11	Frame Synchronization Module	126
12	Synchronization Capture Module	127
13	The Composition of the Reintegrator's Modes	127
14	Reintegrator Module	128
15	Full System Composition	128

CHAPTER 1

Introduction

Digital control systems are being designed for use in safety-critical contexts such as automobiles (“drive-by-wire”) and commercial aircraft (“fly-by-wire”) [2–6]. Safety-critical systems embedded in commercial aircraft must have a failure rate no worse than 10^{-9} per hour of operation [7, 8]. A design error causing a system to fail more often – say once in 10^8 hours – is unacceptable, yet it is infeasible to determine whether a system has this reliability through testing alone [9].

The inability to demonstrate correctness through testing motivates us to *prove* these systems are correct. A variety of protocols are executed in drive-by-wire and fly-by-wire systems. The following considerations make the formal modeling and verification of these protocols difficult: faults, distributed control, nonfunctional behavioral requirements, and intricate protocol interactions. Furthermore, regardless of the protocol or correctness condition to be proved, one must address the real-time properties of these systems and protocols. The specific class of systems considered in this dissertation are *time-triggered systems*. Time-triggered systems are implemented as distributed systems in which each node in the system is independently-clocked, and synchronization mechanisms maintain agreement among these local clocks [8]. Time-triggered systems and their protocols make multiple real-time assumptions and guarantees dependent on the system state; therefore, a variety of real-time models are used in their specification and verification. Under normal operating conditions, these systems maintain tight synchronization among the distributed nodes. When the nodes are tightly synchronized, the temporal behavior of the system can be abstracted

as if the nodes execute in lock-step. This sort of model is the *synchronous model* or *untimed model*. The synchronous abstraction depends on a *realization* (i.e., a concrete implementation – hardware and/or software executing on hardware) satisfying key properties regarding scheduling, message delays, clock skew, message-reception windows, and so on. A more fine-grained model that addresses these properties for time-triggered systems is the *time-triggered model*. Finally, time-triggered systems have states in which the temporal assumptions made in the time-triggered model are not satisfied. In these cases, a more general timing model, such as the *partially-synchronous model* is required. This dissertation develops a methodology for reasoning within and between these three timing models for time-triggered systems.

The first contribution is a set of abstractions for specifying time-triggered protocols in a synchronous model that is suited for verification in a higher-order logic mechanical theorem-prover; in particular, these abstractions model messages, faults, fault-masking computations, and communication.

A synchronous specification simplifies the specification and verification overhead by eliminating timing concerns, but a large semantic gap remains between the timing characteristics of a synchronous protocol specification and its implementation. Therefore, in the second contribution, the time-triggered model [10] is generalized, and arbitrary algorithms in the generalization are proved, using mechanical theorem-proving, to implement their synchronous specifications. We then describe the use of an automated theorem-proving technique that combines bounded model-checking with automated solvers for proving safety properties in infinite-state systems. The technique is used to verify that the protocol implementation schedules satisfy the time-triggered model constraints.

Finally, there are cases in which the time-triggered assumptions cannot be assumed to hold. In particular, because time-triggered systems are safety-critical, they

are designed to tolerate faults. The occurrence of faults can drive a time-triggered system into a state in which the assumptions do not hold. For protocols executing in these states, our third contribution is a description of how to use the same combination of bounded model-checking and automated solvers to verify parameterized real-time specifications of the protocols.

The motivating case-study to which we apply the methodology developed in this work is the SPIDER family time-triggered bus architecture being developed at the NASA Langley Research Center [11,12]. All of the specifications and proofs described herein are provided online [13].

1. Motivation and Approach

1.1. Motivation. Proving the correctness of an entire industrial-scale system such as SPIDER is infeasible, or at the least, it would be the outcome of a “grand challenge” similar in spirit to the ones outlined by Hoare and Moore [14,15]. Therefore, a preliminary challenge is to determine those aspects of the system in greatest need of formal analysis and for which the analysis is feasible.

In this work, we focus on the communication protocols that provide essential system services and fault-tolerance. One motivation is that the protocols are specifications of behavior, and a flawed specification results in flawed implementations. In our case, we are dealing with real-time fault-tolerant protocols, which are particularly complex, and previous designs have been subtly flawed [16,17]. Furthermore, the protocols are the most novel aspect of the design of SPIDER and similar bus architectures. Once the protocols are specified, the design of the hardware and interfaces of the replicated nodes may be substantial but is largely routine.

After determining what is to be verified, the next question is how to approach the verification. A way to reduce the complexity of formally verifying an industrial-scale

system is to develop models that distinguish aspects of its design. Johnson identifies four general aspects – the “ABCD’s” – of system design: *architecture*, *behavior*, *coordination*, and *data* [18]. Each aspect requires a different perspective. Some prominent characteristics of SPIDER that make its formal verification challenging are instances of the ABCD’s:

- *Architecture*: The number of nodes and interconnects in the topology is parameterized (SPIDER is a *family* of architectures).
- *Behavior*: The protocols are complex and interdependent.
- *Coordination*: The nodes are independently-clocked, yet communication has hard real-time deadlines.
- *Data*: The system is designed to be able to pass multiple kinds of data (e.g., clock readings, diagnostic data, sensor readings) and to tolerate data faults.

This work can be understood as developing a set of models for reasoning about the ABCD’s of safety-critical real-time embedded systems, with an emphasis on behavior and coordination. The synchronous model emphasizes behavior while abstracting issues relating to architecture, coordination, and data. The time-triggered model emphasizes coordination, and we show how to ensure this model is consistent with the behavioral model. The partially-synchronous model, however, emphasizes both behavior and coordination, while abstracting architecture and data. Because it emphasizes more than one design aspect, it is the most complex and detailed of the models. To reduce the complexity in reasoning within this model, we use a highly automated tool and reserve the model for only those cases in which behavior and coordination must be reasoned about simultaneously.

Of the four aspects, coordination is the most difficult to reason about in fault-tolerant systems. Real-time behavior in and of itself is notoriously difficult to get

correct since it requires modeling (some degree of) temporal nondeterminism. Reasoning about fault-tolerance, even in the synchronous model, requires modeling non-deterministic behavior and data modifications introduced by faults. Thus, reasoning about real-time fault-tolerance requires a model of both sorts of nondeterminism as well as the additional temporal nondeterminism faults may introduce. Therefore, we emphasize temporal abstractions in this work. By minimizing the amount of reasoning that is carried out in a real-time model, the verification effort becomes manageable.

Finally, we do not develop specialized tools or specification languages. Our work suggests that off-the-shelf tools are adequate for tackling industrial-scale formal verification challenges. PVS is a mature tool and has been widely applied. SAL is more recent, but it incorporates technology that has been shown to reduce the verification effort required in a mechanical theorem-proving approach by orders of magnitude for some real-time verifications [19]. Furthermore, the strong similarities between the languages of PVS and SAL simplify heterogeneous approaches to formal reasoning. The heterogeneous reasoning we carry out is an instance of a more general effort to integrate tools for formal specification and verification [20–22].

1.2. Approach. The methodology as applied to SPIDER is illustrated in Figure 1. Four protocols essential to SPIDER are shown: an *interactive consistency protocol* (IC Protocol), a *distributed diagnosis protocol* (DD Protocol), a *clock synchronization protocol* (CS Protocol), and a *reintegration protocol* (RI Protocol). Of these four protocols, three have synchronous specifications, as depicted. The protocols are interdependent, such that the guarantees provided by one protocol (the *satisfied by* relation) serve as assumptions or preconditions for another (the *assumes* relation). In particular, both the interactive consistency and distributed diagnosis protocols depend on the correctness of the clock synchronization protocol, and the clock synchronization protocol depends on the correctness of the distributed diagnosis

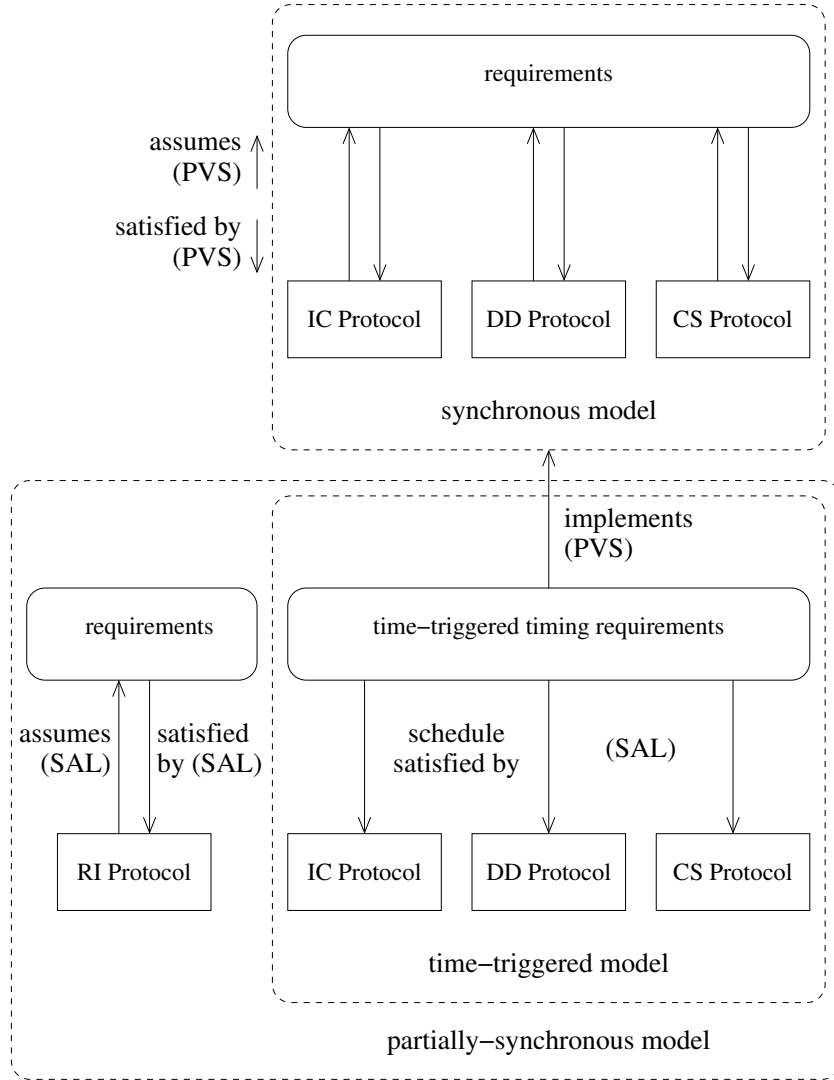


FIGURE 1. Verification Strategy for Time-Triggered Systems

protocol. The formulation¹ and verification of protocols at this level of abstraction is carried out using a mechanical theorem-prover; we use PVS [23].

¹The mathematical *formalization* of a model or proof, which can be done with paper and pencil, should not be confused with its *formulation* in the language of a mechanical theorem-prover or model checker. Nevertheless, the semantics of the language in which a formulation is given attach a formalization to a formulation.

As demonstrated by Rushby [10], the timing requirements for an arbitrary time-triggered protocol can be formulated in PVS, and then a theorem can be proved stating that if these timing requirements are satisfied, a time-triggered protocol implements its synchronous specification. The theorem is represented by the *implements* relation between the time-triggered timing requirements block and the synchronous model. The Symbolic Analysis Laboratory (SAL) [24], which implements an infinite-state bounded model-checker, is then used to verify that the implementation schedules satisfy the time-triggered model constraints. This verification is decomposed so that the scheduling characteristics of each protocol can be verified independently, and the verification is automatic. This is demonstrated using the schedules from the VHDL for the latest FPGA-based prototype of SPIDER’s underlying bus [11].

Finally, the reintegration protocol is a partially-synchronous protocol that cannot be meaningfully abstracted in the time-triggered or the synchronous models. We again use SAL to verify the correctness of the reintegration protocol in a real-time model. The SAL automated solvers allow us to complete a proof parameterized over a range of real-time values.

In general, verification should occur at the most abstract level that does not omit the important characteristics to be verified. Furthermore, in a time-triggered system, protocols do not execute in isolation. The protocols are tightly integrated and interdependent. In describing future work in the formal verification of the Time-Triggered Architecture (TTA), one particular time-triggered system, Rushby describes the importance of formally modeling and reasoning about the system-wide properties that “emerge” from individual protocols behaving and interacting correctly:

What makes TTA useful are not the individual properties of its constituent protocols, but the emergent properties that come about

through their combination. These emergent properties are understood by the designers and advocates of TTA, but they have not been articulated formally in ways that are fully satisfactory, and I consider this the most important and interesting of the tasks that remain in the formal analysis of TTA [25].

Specifying and verifying these emergent properties can be simplified if the interdependent protocols are specified in the same timing model. For example, in TTP/C (a version of TTA), for the clock synchronization protocol to behave correctly, diagnostic correctness conditions must be satisfied. The three essential correctness conditions are as follows [26].

- *Agreement*: nonfaulty nodes agree as to which nodes are faulty;
- *Self-Diagnosis*: faulty nodes diagnose themselves as faulty;
- *Validity*: nonfaulty nodes can believe that no more than one faulty node is nonfaulty.

The satisfaction of these conditions depends on the diagnosis protocol being correct. On the other hand, the diagnosis protocol requires that the clocks of nonfaulty nodes be synchronized within a small skew, which depends on the clock synchronization protocol being correct. Specifying these two protocols in distinct timing models makes it difficult to reason about their interdependence. This is the approach taken in one verification of the TTA: the TTP/C diagnosis protocol is verified in a synchronous model, the TTP/C clock synchronization protocol is verified in a partially-synchronous model, and then these models are related in an assume-guarantee style proof, all of which is carried out in the PVS mechanical theorem-prover [26]. While feasible, the approach is substantial and can require on the order of engineer-years to complete for detailed models of these protocols.

A more automated approach is described herein in which we combine the use of mechanical theorem-proving and infinite-state bounded model-checking. Mechanical theorem-proving is necessary in the specification and verification of fault-tolerant synchronous protocols that require reasoning about parameterized models (e.g., proofs for an arbitrary number of nodes in the distributed system), complex fault-hypotheses [27], complex distributed computations, and non-executable specifications; more automated tools are currently inadequate for this sort of reasoning. However, the verification cost is incurred once for a family of protocols at this level of abstraction. To connect these specifications to their implementations, we develop a refinement mapping from synchronous specifications to time-triggered implementations and use mechanical theorem-proving to verify the correctness of the mapping abstractly, but then use highly-automated infinite-state bounded model-checking to demonstrate that a particular protocol implementation satisfies the conditions necessary to instantiate the mapping.

2. Outline

In Chapter 2, we describe preliminary concepts of this work as well as present related work. In Chapter 3, we present the synchronous model and abstractions for specification and verification in the model. In Chapter 4, the time-triggered model is presented, and we describe how to prove a time-triggered specification implements a synchronous specification. We also demonstrate how to prove that a realized protocol schedule satisfies the constraints of the time-triggered model. In Chapter 5, we describe the use of a recently-developed real-time verification technique to specify and verify protocols in the partially-synchronous model.

Chapter 4 borrows heavily from a formal specification and verification of the time-triggered model presented by Rushby [10]; Appendix A gives proofs of inconsistency

for axioms in his presentation, and it gives appropriate replacements for the inconsistent axioms [28].

Full specifications and proof scripts of the work described herein can be found on-line [13].

CHAPTER 2

Preliminaries & Related Work

1. Fault-Tolerant Distributed Systems

Introductory material on the foundations of distributed systems and algorithms can be found in Lynch’s textbook [29]. Some examples of systems that have fault-tolerant distributed implementations are databases, operating systems, communication buses, file systems, and server groups [7, 30, 31].

A *distributed system* is modeled as a graph with directed edges. Vertices are called *nodes* or *processes*. Directed edges are called *communication channels* or *channels*. If channel c points from node p to node p' , then p can send messages over c to p' , and p' can receive messages over c from p . In this context, p is the *sending node* or *sender*, and p' is the *receiving node* or *receiver*. Channels may point from a node to itself.

The terms failure, error, and fault have technical meanings in the fault-tolerance literature. A *failure* occurs when a system is unable to provide its required functions. An *error* is “that part of the system state which is *liable to lead to subsequent failure*,” while a *fault* is “the *adjudged or hypothesized cause* of an error” [32]. For example, a sensor may break due to a fault introduced by overheating. The sensor reading error may then lead to system failure. A *fault-tolerant system* is one that continues to provide its required functionality in the presence of faults. A fault-tolerant system must not contain a *single point of failure* such that if that single subsystem fails, the entire system fails (for the faults tolerated). Thus, fault-tolerant systems are often implemented as distributed collections of nodes such that a fault that affects one

node will not adversely affect the whole system's functionality. This type of system is referred to as a *fault-tolerant distributed system*.

2. Time-Triggered Systems

Distributed systems coordinate the times at which computation and communication takes place. Events that cause these actions to occur are called *triggers*. In real-time distributed systems, there are two approaches for implementing triggers, *event-triggering* and *time-triggering*. Event-triggers signal the occurrence of some event, e.g., a sensor reaches some threshold. Time-triggers signal some predetermined time has been reached [8, p. 15]. Time-triggers can be considered a special case of event-triggers insofar as they signal the occurrence of an oscillator reaching a threshold [33].

Because the execution of a time-triggered system is driven by the passage of time, it is imperative that the nodes in the system are synchronized and that they agree with the *real time* or “wall-clock time”. If nodes are unsynchronized, then even if they share the same schedule of events, the time at which one node takes an action and the time at which another may expect that action to be taken may differ. Likewise, if the nodes do not agree with real time, then an external component cannot predict when the system will perform some action, even if that component knows the schedule. For example, a time-triggered system may drive an actuator for a motor, and the motor may need to be actuated every ten milliseconds to operate correctly.

There are two means by which synchronization can be achieved. The first and most obvious is if all the nodes in the system share an independent global clock that maintains a precise record of real time. However, often no global clock can be realized, and in a fault-tolerant system, it would represent a single point of failure. A second strategy is for each node to have its own clock. In this case, the nodes

must synchronize their clocks to ensure agreement with each other and with real time. Maintaining synchrony requires the nodes to preserve the *precision* of the local clocks, and maintaining agreement with real time (within a linear envelope) requires the nodes to preserve the *accuracy* of the local clocks [34]. Precision and accuracy can be achieved by periodically executing a clock synchronization protocol [35].

Time-triggered implementations have characteristics that make them attractive for use in real-time safety-critical systems [7, 8]. The execution of a time-triggered system is driven by a schedule agreed upon by the nodes in the system. Therefore, time-triggered systems can reduce the risk of faults propagating from one node to another. Each node knows when to expect certain messages from other nodes. Similarly, time-triggered systems can be easier to compose, and they can ensure hard real-time deadlines are met.

The main disadvantage of time-triggered systems as compared to event-triggered ones is that they are bound to their schedule and can therefore be inflexible in responding to external demands. This can be mitigated by introducing a *dynamic schedule*, as opposed to a *static schedule* [8]. A static schedule is loaded off-line before the system is started. A dynamic schedules can be loaded and unloaded on-line, in response to system demands.

A time-triggered system is designed to maintain synchrony the vast majority of time during which it is operable. However, there are “corner cases” during which the system will not be synchronized, and a system must contain protocols that behave correctly despite the lack of synchrony in these situations. After power-up, there may be a large skew (on the order of seconds) between the time at which the nodes become active. A similar situation occurs during a restart. Restart can be either manual, such as when it is triggered by a human operator, or automatic. Automatic restart can be triggered by massive correlated faults in the system. A special case of restart is when

a single node restarts independently of the rest of the system. If a node determines itself to have suffered a fault, it may attempt to restart itself and *reintegrate* with the other nodes in the system. Fault-tolerant protocols that behave correctly despite the lack of synchrony are used to gain synchronization in these corner cases.

3. Time-Triggered Bus Architectures

Safety-critical bus architectures are a kind of time-triggered system being developed for drive-by-wire and fly-by-wire applications. A generic example of such a bus architecture is presented in Figure 1. There are three essential components in the architecture. The *hosts* are computing platforms distributed throughout a vehicle. Associated with each host may be sensors and actuators for one or more *physical plants*, such as an engine or braking system. As illustrated, these platforms may host various distributed *control applications*. For example, an automobile may have both steer-by-wire and brake-by-wire control applications, and the applications may reside on the replicated microprocessors that are distributed at each wheel. Each host may have sensors to measure the wheel’s skew, rotational velocity, acceleration, etc., and actuators to control these attributes. The *bus interface units* (BIUs) are the interfaces between the hosts and the *interconnect* through which communication takes place. In the figure, the BIUs reside in between the hosts and interconnect. Depending on the architecture, the BIUs may physically be part of the hosts or distinct from them [7]. The entire system, including the hosts, BIUs, and interconnect, is the *bus architecture*.

In our context, the *bus* is conceptual rather than physical. The realization of the bus and what is considered to belong to it depends on the implementation. For example, in SPIDER (Section 4), the BIUs are considered to be part of the bus,

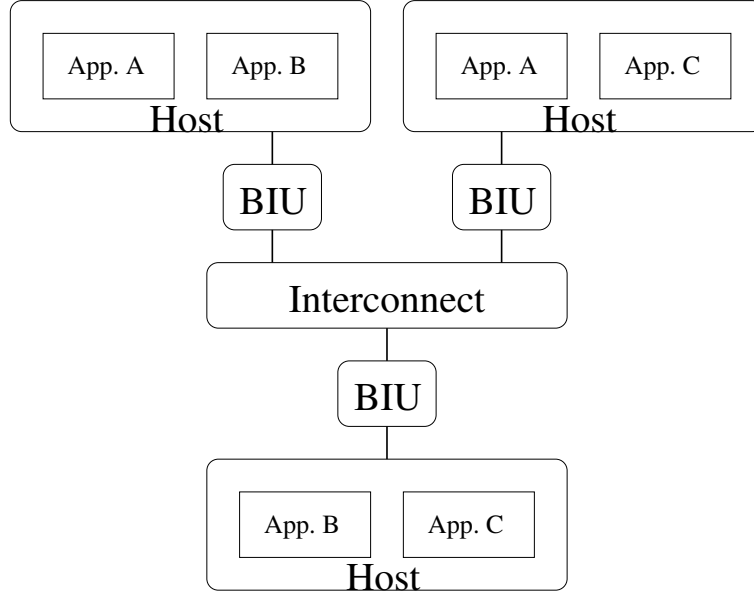


FIGURE 1. A Generic Fault-Tolerant Bus Architecture

whereas in SAFEbus [36], the BIUs are considered to be distinct from it. The bus may have a distributed implementation.

If the bus itself has a distributed implementation, then a bus architecture is a “second-order” distributed system: it is a distributed system at the *application level*, since applications are distributed over the hosts, as well as the at the *bus level*. The two levels of distribution increases the conceptual complexity of a distributed bus architecture. Nevertheless, the bus level distribution should be transparent to the hosts and applications; to them, the bus appears to be a single highly-reliable interconnect. A *time-triggered bus architecture* is distinguished by the property that essential bus-related activities are governed by a global schedule.

3.1. Desiderata. Bus architectures are designed to satisfy four essential desiderata: *fault-tolerance*, *integration*, *partitioning*, and *predictability*.

3.1.1. *Fault-Tolerance.* Fault-tolerant bus architectures must be extremely reliable as compared to off-the-shelf hardware components. Safety-critical applications

in commercial aircraft must have failure rates no worse than 10^{-9} per hour of operation. Because bus architectures may support many applications, their failure rates in commercial aircraft must be no worse than 10^{-10} per hour of operation [7,8]. Because physical components fail at a higher rate, these systems must be designed to tolerate faults. Various mechanisms are incorporated into the design to ensure as many faults as possible are caught and appropriately handled.

System-wide fault-tolerance depends on the existence of *fault-containment regions* (FCRs) [7,37]. FCRs are regions in a system designed to ensure faults do not propagate to other regions. The easiest way to ensure this is to completely isolate separate FCRs. However, because separate FCRs may need to communicate, they share inter-communication channels. Care must be taken to ensure faults cannot propagate over these channels.

What constitutes an FCR depends on the architecture, protocols, and hypothesized faults for the system. In general, a host and its associated BIU should be in a FCR that is distinct from FCRs of the other hosts and BIUs. Even a host and its BIU may be in separate FCRs. Additional FCRs may exist in the architecture; see Section 4 for an example. Generally, physical faults in separate FCRs are statistically independent, but under exceptional circumstances, simultaneous faults may be observed in FCRs. For example, widespread high-intensity radiation may affect multiple FCRs.

Faults can be classified according to the *hybrid fault model* of Thambidurai and Park [38]. The fault model, along with abstractions for it, are described in detail in Section 3 of Chapter 3. Briefly, all non-faulty nodes are also said to be *good*. A node is called *benign*, or *manifest*, if it sends only *benign messages*. Benign messages abstract various sorts of misbehavior. A message that is sufficiently garbled during

transmission may be caught by an error-checking code and deemed benign. In synchronized systems with global communication schedules, messages not received when expected and messages received but unexpected by their recipients are considered to be benign. A node is called *symmetric* if it sends every receiver the same message, but these messages may be arbitrary. A node is called *asymmetric* or *Byzantine* if it arbitrarily sends different messages to different receivers [39].

A *maximum fault assumption* (MFA) states the maximum kind, number, and arrival rate of faults for each FCR under which the system is hypothesized to operate correctly. If the MFA is violated, the system may behave arbitrarily. Therefore, the formal verification of a fault-tolerant system or protocol is always under the assumption of some MFA. The satisfaction of the MFA itself is established by statistical models that take into account experimental data regarding the reliability of the hardware, the environment, and other relevant factors [40]. For example, for bus architectures designed for commercial aircraft, statistical analysis should ensure that the probability of their MFAs being violated is less than 10^{-10} per hour of operation. Even if a system is proved to behave correctly under its MFA, but the probability of the MFA being violated is too high, the system will not reliably serve its intended function.

3.1.2. *Integration.* In a single vehicle, multiple control systems, or *functions*, as they are known in avionics, may be implemented. For example, an automobile may be designed with digitally-controlled braking, steering, and throttling systems. Implementing separate buses for each system increases the cost, weight, power consumption, hardware volume, and wiring volume. Furthermore, advanced integrated by-wire systems may be interdependent; for instance, the braking system may behave differently if the steering system notifies it that the vehicle is turning sharply. For these reasons,

a bus architecture should allow these multiple control systems to be composed. Ideally, the architecture provides a simple interface for integrating off-the-shelf control applications.

Besides the advantages described above, a single bus architecture allows fault-tolerance to be centralized. Without a centralized bus architecture, each system must implement fault-tolerant mechanisms. The fault-tolerance mechanisms for each system are often similar, and if they are implemented once at the bus level (keeping in mind that the bus itself is a distributed system), excessive redundancy of these mechanisms may be eliminated. Furthermore, the integrated design of a control application and fault-tolerance mechanisms is difficult. In fact, fault-tolerance mechanisms themselves have been the primary source of many design errors in control systems [41, pp. 123–135]. The ability to provide systematic fault-tolerance for applications is a primary feature of these architectures.

3.1.3. *Partitioning.* Contrasting with the need to integrate control systems is the need to *partition* them. There are two levels of application-level partitioning [25]. When two applications reside on the same host, they must be *isolated* in the sense defined by the security community in the design and verification of security kernels [42]. In particular, there must not exist illicit communication channels over which information may flow. This sort of isolation is a form of *physical partitioning*. However, communicating applications cannot be completely isolated so that no information flows between them. Another sort of partitioning is called *behavioral partitioning*. For communicating applications, the faulty behavior of one control system should not adversely affect the behavior of another other than by the loss of information from the failed system [25]. These two notions are orthogonal: two applications may be physically partitioned but not behaviorally partitioned, and vice versa.

A consequence of partitioning is that even if two applications reside on the same host, it may be that the only admissible communication channel is through the bus, so that the host broadcasts and then receives its own message. This scheme ensures that the messages passed are subject to the same scrutiny (e.g., error-checks, fault-tolerant voting, etc.) as all other messages passed between applications. Additionally, it relieves the additional complexity of designing hosts with admissible channels.

Partitioning is crucial when control applications of different levels of criticality are composed. Less critical applications may be more complex and undergo less stringent design and testing. Design errors in such systems cannot compromise more critical applications. Not only is this a safety issue, but it is an economic issue. For avionics, the criticality of an application determines the level of certification required [43, 44]; higher certification levels can be orders-of-magnitude more difficult to pass measured in time and money. When two systems are composed, they must be certified according to the highest criticality level possessed by the two systems. Partitioning must be demonstrated to avoid having to certify each application according to the highest criticality level of the constituent applications.

Both kinds of partitioning are difficult to specify and verify. In particular, the statement of behavioral partitioning is qualified: a fault in one control system may affect another, but its effect cannot be more substantial than the resulting loss of information. Rushby asserts that no precise formal statement of this property has been made, let alone verified [25].

3.1.4. *Predictability.* Finally, the bus must be predictable. It must simulate a highly reliable time-division multi-access (TDMA) bus. That is, each application has a globally agreed upon time at which to broadcast over the bus. In addition, the bus must satisfy performance guarantees, particularly *throughput*, the rate at which data

is transmitted from application to application, and *latency*, the response lag between communicating applications.

3.2. Services. Fault-tolerance, integration, partitioning, and predictability are high-level design criteria. In satisfying these, the bus architecture provides some specific services to the hosts and their applications.

3.2.1. *Time-Reference.* The bus provides a fault-tolerant time-reference to the hosts. It is important that the hosts maintain synchronization so they agree with respect to the global schedule, and that they stay within a linear envelope of real time. This can be achieved by periodically executing a clock synchronization protocol as described in Section 2.

3.2.2. *Guaranteed Consensus.* According to the schedule, each host broadcasts over the bus at a scheduled time. It must be ensured that the messages received from this *source host* by the other hosts are the same; this property is called a *consensus* or *agreement* property. To ensure consensus, the bus executes a fault-tolerant *interactive consistency protocol* [29] during each broadcast.

3.2.3. *Reconfiguration and Diagnostic Consensus.* If some FCR is determined to have suffered a fault by the bus, the bus may *reconfigure* to exclude that FCR from participating in subsequent activities. When messages are passed through the architecture during the execution of the protocols, faulty behavior may be observed by the FCRs. For example, the message a FCR receives from another may arrive at an unexpected time, or the message may fail a bit error detection algorithm [45]. When one FCR has reason to believe another has behaved in a faulty manner, we say that the FCR *accuses* it. The determination to exclude the faulty FCR is made by FCRs with synchronized state delivering the system services. These FCRs are called the

clique. A *group membership* protocol ensures that the accusations made are consistent among the FCRs in the clique. If agreement is reached that an FCR is faulty, that FCR can be excluded from the clique.

If the architecture is reconfigured too many times and too many FCRs are excluded, there may not be enough remaining to tolerate subsequent faults or even to deliver the necessary services. *Transient faults* cause a node to lose its volatile state, but do not physically damage the node. In some environments, transient faults can be the dominant kind of fault encountered [40]; in such cases, it is possible for a node that has suffered a fault to regain the necessary system state and *reintegrate* with the clique. Reintegration is another form of reconfiguration in which an FCR associated with the transiently-faulty node is re-admitted to the clique.

3.3. Realizations. Some prominent examples of time-triggered bus architectures currently in development include FlexRay, being developed by an automotive consortium [46], TTTech’s Time-Triggered Architecture (TTA) [47], and Honeywell’s SAFEbus [36]. NASA Langley’s SPIDER is the focus of this work and described in detail in the following section. The primary way in which these architectures differ is in the level of fault-tolerance they are designed to provide. The fault-tolerance requirements are mitigated by economic concerns. Time-triggered architectures designed for the automotive industry, for example, are generally less fault-tolerant and less costly than those for the aerospace industry (they contain less hardware and therefore are lighter and consume less power). Rushby compares and contrasts these time-triggered architectures [7].

4. SPIDER

The case-study in this dissertation is the Scalable Processor-Independent Design for Enhanced Reliability (SPIDER) being designed at the NASA Langley Research

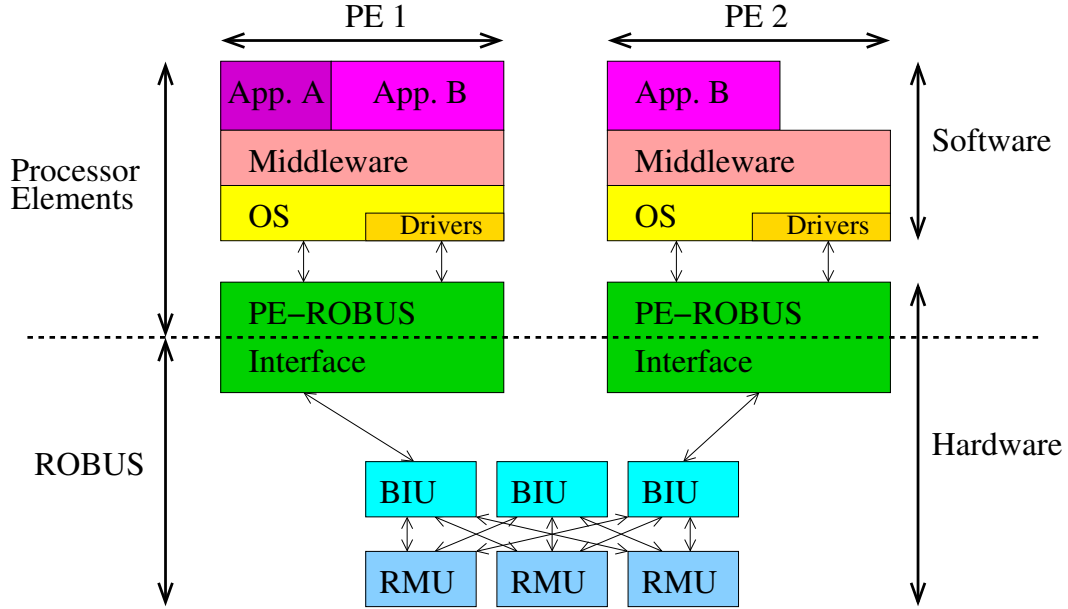


FIGURE 2. SPIDER Architecture

Center [11, 12]. SPIDER is a fault-tolerant bus architecture designed for fly-by-wire systems in commercial aircraft. SPIDER is designed to particularly execute correctly in the presence of electromagnetic disturbances that can lead to Byzantine faults [48].

4.1. Architecture. The SPIDER architecture can be physically divided into the set of *processor elements* (PEs) and the *Reliable Optical Bus* (ROBUS), as shown in Figure 2. Each PE contains a microprocessor, real-time operating system, and the necessary middleware and drivers. Software applications, such as controllers and actuators, run the PEs. A particular application may be distributed over the PEs. The ROBUS is a fault-tolerant virtual bus that is the backbone of the SPIDER architecture. It is a complete bipartite graph of *bus interface units* (BIUs) and *redundancy management units* (RMUs). BIU nodes are the interface between the ROBUS and the PEs. Each BIU is connected to exactly one PE. The RMUs are the additional nodes in the ROBUS that provide the redundancy necessary to deliver fault-tolerance

guarantees and in particular, Byzantine-resilience [39, 48]. Each BIU shares an interconnect with each RMU, and no two BIUs or RMUs share interconnects. Each PE, BIU, and RMU is designed to be a FCR.

The ROBUS is designed to be implemented by an arbitrary number of BIUs and RMUs; the number of each need not be equal. However, the number of BIUs and PEs is equal (for space considerations, one PE is not depicted in Figure 2). In a SPIDER configuration with a large number of PEs, there will usually be fewer RMUs. The number of RMUs is influenced by weighing the additional fault-tolerance provided by additional RMUs against the additional cost and size of the system and the additional hardware and interconnects that can suffer faults.

4.2. Operational Modes. Each node (i.e., a BIU or RMU) in the ROBUS has the following modes of operation [11]:

- a *disabled mode* in which the node is inactive;
- a *self-test mode* in which the node performs self-diagnostics upon power-up, restart, or if a local failure is detected;
- a *clique-detection mode* in which the node searches for the existence of a *clique*, or a set of non-faulty nodes with coordinated activity;
- a *clique initialization mode* in which the node attempts to form a new clique (if the node determines no clique to be present);
- a *clique join mode* in which the node attempts to join a pre-existing clique;
- a *clique preservation mode* in which the node is in a clique, and the clique is delivering its services to the attached PEs.

4.3. Protocols. The ROBUS executes a number of fault-tolerant distributed protocols to deliver the services required of the ROBUS by the attached PEs and also to maintain its internal state [11]:

- an *interactive consistency protocol* that ensures a fault-tolerant message broadcast from a single source to a set of receivers;
- a *distributed diagnosis protocol* that ensures non-faulty nodes maintain a consistent representation of the fault-status of the other nodes in the system;
- a *schedule update protocol* in which a new system schedule is loaded;
- a *clock synchronization protocol* that ensures the local clocks maintain accuracy and precision;
- *startup and restart protocols* that ensure nodes become synchronized and achieve consistent state upon power-up and restart, respectively;
- a *reintegration protocol* that is executed during the reintegration mode (see Section 2 in Chapter 5).

4.4. Maximum Fault Assumption. The MFA we present here is for the ROBUS with respect to faults suffered by the BIUs and RMUs (if a communication channel suffers a fault, we ascribe that fault to the sending node; see Section 3.3). Each protocol that is executed in the ROBUS must execute correctly when the MFA is satisfied. We call this MFA a *Dynamic Maximum Fault Assumption* (DMFA) to emphasize that the fault assumption is parameterized by the local diagnoses of nodes, which change over time.

DEFINITION 2.1 (ROBUS DMFA). Let B_G , B_S , and B_A denote the sets of BIUs that are good, symmetrically-faulty, and asymmetrically-faulty, respectively. Let R_G , R_S , and R_A represent the corresponding sets of RMUs, respectively. For good BIU b , let T_b denote the set of RMUs b trusts called b 's *trusted set*. Define T_r similarly – it is the set of BIUs that RMU r trusts. The following formulas together make up the DMFA. For all BIUs b and RMUs r ,

$$(1) |R_G \cap T_b| > |R_S \cap T_b| + |R_A \cap T_b| ;$$

- (2) $|B_G \cap T_r| > |B_S \cap T_r| + |B_A \cap T_r|$;
- (3) $|R_A \cap T_b| = 0$ or $|B_A \cap T_r| = 0$.

The first clause ensures that a good BIU b contains strictly more good RMUs in T_b than it does symmetrically-faulty or asymmetrically-faulty RMUs. The second clause ensures the same holds for the good RMUs. The third clause ensures that either no good BIU trusts an asymmetric RMU, or no good RMU trusts an asymmetric BIU. However, this does not preclude multiple good BIUs from trusting asymmetric RMUs, and similarly for the RMUs. Formal proofs in the mechanical theorem-prover PVS that the SPIDER Interactive Consistency, Distributed Diagnosis, and Clock Synchronization protocol tolerate all fault scenarios under the ROBUS MFA exist [27].

5. Time-Triggered System Verification

The two architectures that have undergone the most comprehensive formal verifications are TTTech’s Time-Triggered Architecture (TTA) and SPIDER, described in the subsequent sections. Other related formal verification efforts are presented in Section 5.3.

5.1. TTTech’s Time-Triggered Architecture Verification. TTA [8] is one of the most mature and most extensively formally-verified architectures in development. Two sets of protocols exist for the TTA: TTP/A and TTP/C. The latter is for safety-critical applications, whereas the former is for less critical applications [8]. The following verifications are generally with respect to the TTP/C implementation. A number of its protocols have been formally verified. Rushby overviews the formal verification of TTA [25]. Pfeifer and von Henke also describe the formal verification of TTA [49–51].

Pfeifer, Schwier, and von Henke formally verify the TTA Clock Synchronization Protocol in PVS [52]. Pfeifer’s dissertation extends this work by formally verifying

its clock synchronization and group membership protocols and then providing an assume-guarantee proof that each provides the requirements necessary to demonstrate the correctness of the other [26]. This work is carried out in PVS.

Dutertre and Sorea use the k -induction proof technique, described in Section 6.2.3, in SAL to give a real-time verification of the TTA Startup Protocol [53, 54]. Dutertre and Sorea’s work is a generalization of a discrete-time verification of the TTA Startup Protocol [55].

Finally, Rushby verifies in PVS the timing characteristics for message passing in TTA [56].

5.2. SPIDER Verification. Geser and Miner formally verify in PVS an early version of the SPIDER Distributed Diagnosis Protocol [57]. Pike, Miner, and Torres-Pomales use PVS to uncover an error in an early version of the SPIDER Interactive Consistency Protocol and use SAL to generate a concrete counter-example to its correctness (using symbolic and bounded model-checkers) [16]. The Unified Fault-Tolerance Protocol generalizes the clock synchronization, interactive consistency, and distributed diagnosis protocols implemented in SPIDER into one fault-tolerance protocol [27]. A formal verification of the protocol is presented in PVS. These proofs are being generalized to be applicable to other time-triggered systems.

5.3. Other Verifications. Rushby describes systematic techniques for the verification of time-triggered protocols [10, 58]. It is inspired by TTA, but is applicable to other time-triggered systems. These efforts generalize earlier work by the NASA Langley Research Group to develop the *Reliable Computing Platform* (RCP) [59–61]. The RCP is a fault-tolerant platform for fly-by-wire systems. Furthermore, it is able to recover from transient faults and provides a *uniprocessor system layer* interface hiding

the distributed implementation [59]. A goal of the project is to develop methodologies to formally specify and verify the system. Many of the design and verification principles for SPIDER were born out of the RCP project. The RCP project itself is an outgrowth of other fault-tolerant architectures that underwent some formal verification including SIFT [62] and MAFT [63].

Complimenting these endeavors are more specific efforts. A fault-tolerant clock synchronization protocol [35, 64] is crucial to the implementation of a time-triggered system. There have been many verifications of such protocols [34, 65–67]. Interactive consistency protocols [29] have undergone numerous verifications, too [17, 68–72]. Finally, the formal verification of distributed protocols in general is motivated by noting that although an interactive consistency protocol is one of the simplest in a time-triggered architecture, published peer-reviewed unmechanized proofs-of-correctness have been flawed [17].

6. Tools

The tools used in this dissertation are the mechanical theorem-prover PVS and infinite-state bounded model-checker SAL, both of which are developed by SRI, International. This work could be reproduced in a straightforward way using other mechanical theorem-provers. In particular, it would be straightforward to reproduce this work in other higher-order logic theorem-provers such as HOL [73] or Isabelle [74], although the use of predicate subtypes in the PVS specifications could not be directly reproduced. Finally, PVS and SAL are used in an integrated way in Chapter 4. This use is aided by their similar specification languages.

6.1. Prototype Verification System (PVS). The *Prototype Verification System* (PVS) is an interactive mechanical theorem-prover [23]. It belongs to the family of higher-order logic theorem-provers, including HOL and Isabelle mentioned above.

Below, we briefly describe the specification language and the verification environment of PVS. Many extensions to PVS have been developed; in particular, the verifications herein depend on the packages Manip [75] and Field [76] for high-level arithmetic reasoning in PVS. Results from the NASA Langley PVS Libraries are also used [77].

6.1.1. *Specification Language.* Specifications in PVS are in a simply-typed higher-order logic [78]. Some base types are predefined (e.g., booleans), and uninterpreted base types may be specified. Composite types can be constructed with function-type, tuple-type, record-type, enumeration-type, and co-tuple-type constructors. In addition, PVS supports predicate subtyping. A mechanism is provided for specifying abstract datatypes that are well-founded trees.

Constants and variables can be specified, and they must be typed. Constants may be either interpreted or uninterpreted. Because the language is higher-order, variables may range over any type declaration, including function-types. Definitions (i.e., specified interpreted constants) *conservatively extend* the language: inconsistencies cannot be introduced by definition, although they can be introduced by axiomatization.

Functions of type $[\mathcal{D} \rightarrow \mathcal{R}]$ are total mappings from the domain type \mathcal{D} to the range type \mathcal{R} . Recursive (interpreted) functions can be specified in the language (mutual recursion is not directly supported). Recursive function definitions generate a proof obligation that for all arguments, the recursion terminates. The proof obligation is generated automatically by PVS, and in many cases, is also discharged automatically.

Such proof obligations are generated by the PVS static type-checker. The type-checker ensures type inconsistencies have not been introduced. For example, one is obliged to demonstrate that if a constant is declared to be of type \mathcal{T} , then \mathcal{T} is a nonempty type. When PVS cannot determine whether a type inconsistency has been

introduced, it generates an unproven *type correctness condition* (TCC). The user is obliged to prove the TCC to demonstrate the typing is consistent.

Specifications are modularized by *theories*. A theory may contain type declarations, constant declarations, and variable declarations. It may also contain propositions stated in the language of the theory. A theory may be parameterized by both types and constants. Theories may import other theories. A theory that imports another theory inherits the language of the imported theory. Furthermore, propositions stated in an imported theory are inherited and may be used as lemmas in proving theorems in the current theory.

Additionally, every theory automatically inherits a collection of theories referred to as the *PVS Prelude*. These theories describe various foundational mathematical facts.

6.1.2. *Verification Environment*. The logic of PVS is the *sequent calculus*. A *sequent* is of the form

$$\bigwedge \Gamma \vdash \bigvee \Delta,$$

where Γ , called the *antecedent*, and Δ , called the *consequent* are finite sets of propositions. A sequent asserts that the conjunction of the antecedents implies the disjunction of the consequents. *Inference rules* are mappings from sequents to sequents. A sequent is manipulated with inference rules in PVS until an *initial sequent* is reached. A sequent is initial when the set of antecedents contains the boolean constant **F**, or the set of consequents contains the boolean constant **T**, or the same proposition is both an antecedent and a consequent.

Atomic proof rules may be combined into a sequence of proof rules called *proof strategies* [79]. Strategies are not guaranteed to terminate. For example, a strategy may contain nonterminating rewrite rules.

6.1.3. *Example: Sets.* In Figure 3 is a small PVS theory of sets. The theory reproduces part of the PVS Prelude [80], a set of theories that comes pre-installed with PVS.

```

sets [T: TYPE]: THEORY
BEGIN

  set: TYPE = [T -> bool]
  x, y: VAR T
  a, b, c: VAR set

  member(x, a): bool = a(x)
  emptyset: set = {x | false}

  extensionality: LEMMA
    (FORALL x: member(x, a) IFF member(x, b)) IMPLIES (a = b)

END sets

```

FIGURE 3. Set Definition in PVS

The identifier `sets` denotes the name of the theory. The theory takes one parameter, an arbitrary type `T`. The theory parameter can be instantiated by another type value when the theory is instantiated by another theory. For example, `T` might be instantiated by the type of the natural numbers in a theory describing sets of naturals. The first declaration in the theory `sets` is a *type declaration* in which `set` is declared to be a function type mapping elements of type `T` to a boolean value; thus, sets are formulated as functions in PVS. Two identifiers `x` and `y` are declared to be variables of the type `T`, and the identifiers `a`, `b`, and `c` are declared to be variables of the type `set`. Next, the function `member` is defined. The function returns the value of evaluating `a` at `x`, recalling that the set `a` is represented as a function. The function `a(x)` is true if and only if `x` is a member of `a`. The constant `emptyset` is defined to be the set of no elements, `{x | false}`, using set-comprehension notation.

The theory contains one lemma, **extensionality**, stating that two sets are equal if their members are the same. Before proving the theorem in PVS, the theory must be parsed and type-checked. PVS automatically parses and type-checks, but if a type-correctness obligation exists that PVS cannot prove automatically, the onus is on the user to prove it interactively. The theory above generates no type-correctness obligations.

A fine-grained proof-sketch of **extensionality** that can reproduced mechanically in the theorem-prover follows. Expanding the definition of **member**, we see that the result can be proved with *function extensionality*; that if two functions yield the same value when applied to each point of their domains, then the two functions are equal. The output of the interactive proof is presented in Figure 4 and continued in Figure 5. In the interactive prover, PVS presents a sequent, and the user inputs an inference rule transforming the sequent into a new sequent. In the PVS prover, the sequent is represented with a *turnstile* denoted by “|-----”, where antecedents are above the turnstile, labeled with negative integers, and consequents are below the turnstile, labeled with positive integers. The following inference rules are used to transform the sequents in this example:

- (**skolem! 1**) introduces *skolem constants* for the quantified variables in the formula number 1, where the skolem constant identifiers are constructed from the quantified variable identifiers by appending the identifier **!n**, where **n** is a natural number.
- (**flatten**) performs propositional simplification.
- (**apply-extensionality**) introduces the function extensionality rule for the consequent formula by introducing a new consequent $a!1(x!1) = b!1(x!1)$, where $x!1$ is an arbitrary constant in the domains of the functions $a!1$ and $b!1$.

```

extensionality :
  |-----
{1}  FORALL (a, b: set):
      (FORALL x: member(x, a) IFF member(x, b))
      IMPLIES (a = b)

Rule? (skolem! 1)
Skolemizing,
this simplifies to:
extensionality :
  |-----
{1}  (FORALL x: member(x, a!1) IFF member(x, b!1))
      IMPLIES (a!1 = b!1)

Rule? (flatten)
Applying disjunctive simplification to flatten sequent,
this simplifies to:
extensionality :

{-1} FORALL x: member(x, a!1) IFF member(x, b!1)
  |-----
{1}  (a!1 = b!1)

Rule? (apply-extensionality)
Applying extensionality,
this simplifies to:
extensionality :

[-1] FORALL x: member(x, a!1) IFF member(x, b!1)
  |-----
{1}  a!1(x!1) = b!1(x!1)
[2]  (a!1 = b!1)

```

FIGURE 4. Interactive Proof Session in PVS (Continued in Figure 5)

- (inst -1 "x!1") instantiates the quantified variable in formula -1.
- (expand "member") expands the definition of member.

```

Rule? (inst -1 "x!1")
Instantiating the top quantifier in - with the terms:
  x!1,
this simplifies to:
extensionality :

{-1} member(x!1, a!1) IFF member(x!1, b!1)
  |-----
[1]   a!1(x!1) = b!1(x!1)
[2]   (a!1 = b!1)

Rule? (expand "member" )
Expanding the definition of member,
this simplifies to:
extensionality :

{-1} a!1(x!1) IFF b!1(x!1)
  |-----
[1]   a!1(x!1) = b!1(x!1)
[2]   (a!1 = b!1)

Rule? (iff)
Converting top level boolean equality into IFF form,
Converting equality to IFF,
this simplifies to:
extensionality :

[-1] a!1(x!1) IFF b!1(x!1)
  |-----
{1}  a!1(x!1) IFF b!1(x!1)
[2]  (a!1 = b!1)

which is trivially true.
Q.E.D.

```

FIGURE 5. Interactive Proof Session in PVS (Continued from Figure 4)

- (iff) converts an equality operator = on boolean-typed terms to an “if and only if” operator.

Other sequences of inference rules can prove this theorem. In particular, PVS has more powerful commands that replace sequences of inference rules. For example, the sequence of inference rules (`grind`), (`apply-extensionality`), (`grind`) is sufficient to prove the same theorem. In fact, these rules can be combined into a single user-defined proof rule. More generally, the PVS prover can be programmed with *proof strategies*, programs for proving classes of theorems [79].

6.2. SRI’s Symbolic Analysis Laboratory (SAL). The *Symbolic Analysis Laboratory* (SAL) is a suite of automated verification tools developed by SRI, International [24,81]. SAL includes explicit-state, symbolic, and bounded model checkers, an interactive simulator, as well as other tools.

6.2.1. *Specification Language.* A single language serves as the input to the verification tools. The language is a simply-typed higher-order logic, just as in PVS. The predefined interpreted base types in SAL include booleans, the natural numbers, and the real numbers. Composite types may be built up in the same ways as in PVS. Types built from the naturals or reals are called *infinite types* to distinguish them from *finite types*. Both interpreted and uninterpreted constants may be specified. Variables may range over any defined type, but quantified variables may range over only finite types.

The language includes special constructs for building transition systems. A transition system is specified by a *module*. The transitions in a module are specified with a set of guarded transitions that are *enabled* if the guard is true. The guard may be any boolean combination of state variables and constants, including defined relations over the variables. Additionally, in a synchronous composition, described below, guards may reference next-state variable values, under certain conditions. When a guard is executed, some subset of the writable state variables are updated. If more than one guarded transition is enabled, exactly one is nondeterministically taken.

A module contains any combination of the following set of variables: input variables, output variables, global variables, and local variables. Output, global, and local variables can be both read and written; input variables can only be read. Modules may be composed to generate composite transition systems. Modules communicate via shared variables. Modules can be both synchronously and asynchronously composed. In a *synchronous composition*, each module simultaneously executes a guarded transition. Synchronously-composed modules cannot share global variables (this prevents concurrent read-write conflicts). If one module is *deadlocked* – i.e., no guarded transition is enabled – then the synchronous composition of modules is deadlocked. In an *asynchronous composition*, modules nondeterministically interleave their executions. However, in an asynchronous composition, if one module is deadlocked, then the other module must execute a transition, if it is not itself deadlocked.

Additionally, modules may be parameterized by some finite-range type, and then instantiated for each element of the type. Variables must be renamed in this case.

6.2.2. *ICS*. To reason about constraints containing constants and variables over infinite types, an automated solver is incorporated into SAL. The automated solver employed by SAL is the *Integrated Canonizer and Solver* (ICS), an automated solver for a quantifier-free, first-order theory of equality, the terms of which include uninterpreted functions, linear arithmetic, products, arrays, and fixed-sized vectors. [82]. Other solvers, such as UCLID [83] and CVC [84], may be plugged in to SAL.

6.2.3. *k-Induction*. SAL combines its bounded model-checking and ICS to prove that safety properties, stated as LTL formulas, hold in infinite-state transition systems. The invariants do not need to be strictly inductive; SAL supports *k-induction*, also known as *temporal induction*, a generalization of the ordinary induction principle (over transition systems) [85, 86]. Let $\langle S, S^0, \rightarrow \rangle$ be an *unlabeled transition system* where S is a set of states, $S^0 \subseteq S$ is a nonempty set of initial states, and $\rightarrow \subseteq S \times S$

is a transition relation. For a natural number k ($k \geq 0$), a k -trajectory is a sequence of states, s_0, s_1, \dots, s_k , such that for $0 \leq i < k$, $s_i \rightarrow s_{i+1}$ (a 0-trajectory (over the transition system) is a state s). Then the k -induction principle is as follows.

DEFINITION 2.2 (k -Induction Principle). Let k be a natural number, and let $P : S \rightarrow \text{bool}$ be some predicate defined over the states of S .

- *Base Case*: For each k -trajectory s_0, s_1, \dots, s_k such that $s_0 \in S^0$, $P(s_k)$ holds.
- *Induction Step*: For each k -trajectory s_0, s_1, \dots, s_k , if for all $0 \leq j < k$ $P(s_j)$ holds, then $P(s_k)$ holds.

Property P is a k -inductive property with respect to $\langle S, S^0, \rightarrow \rangle$ if there exists some $k \in \mathbb{N}^{0<}$ such that P satisfies the k -induction principle. The ordinary induction principle is the special case when $k = 1$. The benefit of k -induction is that as k increases, weaker invariants may be provable. The problem of discovering sufficiently strong inductive invariants can be difficult, and more often than not, a desired invariant is too weak to be proved with the ordinary induction principle. Discovering sufficiently strong inductive invariants is an active area of research [87, 88].

Furthermore, state invariants can be used as lemmas to support k -induction. An invariant has the effect of strengthening the antecedents in the base case and induction step so that only states satisfying the invariant are considered.

DEFINITION 2.3 (k -Induction Principle with Inductive Invariants). Let Q be an invariant over the states of S .

- *Base Case*: For each k -trajectory s_0, s_1, \dots, s_k such that $s_0 \in S^0$ holds and for each $0 \leq i \leq k$, $Q(s_i)$ holds, $P(s_k)$ holds.
- *Induction Step*: For each k -trajectory s_0, s_1, \dots, s_k such that $Q(s_i)$ holds for each $0 \leq i \leq k$, if for all $0 \leq j < k$ $P(s_j)$ holds, then $P(s_k)$ holds.

6.2.4. *Example: Bakery Algorithm.* The SAL specification in Figure 6 is an infinite-state specification of the Bakery Algorithm in SAL, as formulated by Rushby [1]. The algorithm is a scheduling algorithm for asynchronous processes. It exhibits many but not all of the syntactic elements of SAL. Its proof of correctness demonstrates the use of SAL’s infinite-state bounded model checker to prove safety properties.

The identifier `bakery` names the current context (a context is similar to a PVS theory). Types `phase` and `ticket` are type declarations where `ticket` is declared to be the infinite type of the natural numbers. The module `process` contains one input variable and two output variables. The output variables are initialized to constant values. Three guarded transitions are specified in the module, and they are asynchronously composed with the `[]` operator. In the first transition, if `pc = idle`, then `my_t` and `pc` are both updated. To avoid name clashes, `P1` is a module defined as `process` with the `pc` renamed; `P2` is likewise renamed. The module `system` is the asynchronous composition of these two processes. A theorem named `strong_prop` is stated. It states that in all states reachable when executing `system`, if `pc1 = critical` and `pc2 = trying`, then `my_t < other_t`, and likewise when `P2` is critical state.

The theorem `strong_prop` is proved by issuing the following command to SAL’s infinite-state bounded model-checker:

```
sal-inf-bmc -i -d 3 bakery strong_prop
```

The command `sal-inf-bmc` calls the infinite-state bounded model-checker. The argument `-i` commands a k -induction proof to be attempted, and the argument `-d 3` specifies the depth at which the k -induction is attempted. The last two arguments are the context name and theorem name, respectively.

```

bakery : CONTEXT =
BEGIN
  phase : TYPE = {idle, trying, critical};
  ticket: TYPE = NATURAL;

process : MODULE =
BEGIN
  INPUT other_t: ticket
  OUTPUT my_t: ticket
  OUTPUT pc: phase

  INITIALIZATION
  pc = idle;
  my_t = 0

  TRANSITION
  [pc = idle -->
    my_t' = other_t + 1;
    pc' = trying
  []
    pc = trying AND (other_t = 0 OR my_t < other_t) -->
    pc' = critical
  []
    pc = critical -->
    my_t' = 0;
    pc' = idle
  ]
END;

P1 : MODULE = RENAME pc TO pc1 IN process;

P2 : MODULE = RENAME other_t TO my_t,
           my_t TO other_t,
           pc TO pc2 IN process;

system : MODULE = P1 [] P2;

           :

strong_prop: THEOREM system |-
  G(((pc1 = critical AND pc2 = trying) => my_t < other_t) AND
    ((pc2 = critical AND pc1 = trying) => other_t < my_t));
END

```

FIGURE 6. Rushby's SAL Specification of the Bakery Algorithm [1]

7. Timing Models

We briefly describe the timing models of distributed systems used in this dissertation. Each chapter contains a more rigorous description of the timing model used therein.

The *untimed synchronous model* is so-called to emphasize that communication and computation are modeled as synchronous, instantaneous events. The granularity of time is with respect to a *round*. A round is comprised of two phases, a communication phase and a computation phase. In the *communication phase*, each node in the system instantaneously updates its outbound channels based on its current state. In the *computation phase*, each node instantaneously updates its state based on its current state and the messages it receives over its inbound channels.

In *partially-synchronous models*, the rate at which nodes execute (i.e., update their local state and send messages) may differ. However, this model is more restricted than the fully asynchronous model in which the differences may be unbounded. The partially-synchronous model closely models the implementation of real-time distributed protocols. In general, verifying a protocol in this model provides greater assurance of correct design than verifying it in the synchronous model does, since the latter model may abstract away subtle timing errors. Because it is more detailed, it is more complicated. Lynch states that our understanding of this model is nascent in comparison to synchronous and asynchronous models, and additional research in the area is needed [29, 89].

A kind of partially-synchronous model is the *time-triggered model*. In the time-triggered model, we constrain some of the behaviors allowed in the more general partially-synchronous model. The time-triggered model is an abstraction of *time-triggered implementations*. The behaviors of nodes are mostly governed by the passage of time as measured by their local clocks. Each node maintains a schedule stating

when it should take certain actions. A node will perform an action when its clock reaches the time that action is scheduled. This is opposed to *event-triggered implementations* in which the actions taken by nodes are determined by the occurrence of events in its environment. For example, a node may take some action immediately upon receiving a message from another node.

8. Timeout Automata: A Real-Time Model

The partially-synchronous model developed in Chapter 5 extends a real-time model developed for k -induction proofs in the SAL modeling language called *timeout automata* [53]. The development of timeout automata is motivated by Dutertre and Sorea’s experiments in using *timed automata* [90] for proofs by k -induction in SAL. Although they demonstrate that it is possible to specify timed automata in SAL via a shallow embedding – i.e., a timed automata is manually translated into a semantically-equivalent SAL specification – it proves to be unwieldy. The SAL language is rich, but it is a general-purpose tool for specifying composed state machines; neither the syntax nor the semantics of the language match those of timed automata well. Furthermore, the clock variables in timed automata may be updated in arbitrarily small increments leading to infinite trajectories in which the discrete state idles. This makes proof by k -induction difficult and sometimes impossible.

These difficulties motivated them to develop an explicit real-time model, timeout automata. In an explicit real-time model, the current time is tracked with a variable, and discrete events have real-time bounds on when they can occur [54, 91]. This contrasts with implicit, or clock-based, real-time models, such as timed automata. One attraction of explicit real-time models is the simplicity of their syntax – time constraints are modeled as inequalities over the reals, and they require no special semantics for verification.

The timeout automata¹ model is borrowed from the model of system execution used in discrete-event simulation [92]. The central idea behind timeout automata is that timeouts mark the point in the future when a system event occurs, and timeouts themselves may be nondeterministically updated. Time is always maximally updated to when the next system event occurs. Dutertre and Sorea provide the semantics of a timeout automaton in terms of a transition system [53, 54]. Fix a set of state variables V . An additional variable t , ranging over the nonnegative reals, records the current time. There is also a set of *timeout variables* T , ranging over the nonnegative reals. A *state* ρ in the transition system is a function mapping each variable to some value from the set over which it ranges. For any initial state ρ , $\rho(t) \leq \rho(x)$ for all $x \in T$. As in the definition of timed automata, there are two sorts of transitions: *time progress transitions* and *discrete transitions*. A time progress transition is enabled in a state if and only if for all $x \in T$, $\rho(t) < \rho(x)$. In this case, the state changes by updating $\rho(t)$ to the least-valued timeout (there may be multiple timeouts that are least-valued). Discrete transitions are enabled in a state if and only if there exists some timeout x such that $\rho(t) = \rho(x)$. Furthermore, the following conditions must hold for a discrete transition from state ρ to ρ' :

- $\rho'(t) = \rho(t)$;
- for all $x \in T$, $\rho'(x) \geq \rho(x)$;
- there exists $y \in T$ such that $\rho(y) = t$ and $\rho'(y) > \rho(y)$.

The third condition prevents infinite zero-delay state transitions. If multiple discrete transitions are enabled in a state, exactly one is nondeterministically applied. Note, too, that discrete transitions are instantaneous (i.e., the current time is not updated during their application).

¹We use ‘automata’ to refer to syntax, distinct from the semantics for automata.

An important distinction between timeout automata and formalisms like timed automata is that in a timed automaton, clocks measure how much time has elapsed since their last reset, whereas timeouts measure how much time will elapse until the next state transition.

CHAPTER 3

Synchronous Protocol Verification

This chapter presents a set of abstractions that serves as a framework for the specification and verification of synchronous fault-tolerant distributed protocols. Although the synchrony assumption simplifies reasoning about these protocols, modeling and verifying them is nevertheless difficult. Modeling and reasoning about the number of and kinds of faults, distributed control, complex fault-masking, nonfunctional behavioral requirements, and intimate protocol interactions contribute to the difficulty.

Although many fault-tolerant distributed systems and algorithms have been specified and verified, the models used have often been ad-hoc. Developing models at the appropriate level of detail is a difficult aspect of formal verification [41]. We present a generic model of common aspects relevant to the verification of fault-tolerant distributed systems.

The abstractions presented are in same spirit as ones to model digital hardware, as developed by Thomas Melham [93,94] insofar as they are intended to make specifications and their proofs of correctness less tedious, less error-prone, and more uniform. Furthermore, they are formulated for easy specification in a mechanical theorem-prover, as are Melham's. Although the abstractions we describe are quite general, we intend for them to be accessible to the working verification engineer.

These abstractions are largely the outcome of verifying the fault-tolerant distributed protocols of SPIDER in an synchronous model. Their development and formulation in a mechanical theorem-prover is joint work by this author, Paul Miner, Alfons Geser, and Jeffrey Maddalon [95]. Our work is used to specify and verify the

SPIDER Distributed Diagnosis, Interactive Consistency, and Clock Synchronization protocols [11] in PVS [27].¹

We introduce four fundamental abstractions in the domain of fault-tolerant distributed systems. Message abstractions address the correctness of individual messages sent and received. Fault abstractions address the kinds of faults possible as well as their effects in the system. Fault-masking abstractions address the kinds of local computations nodes make to mask faults. Finally, communication abstractions address the kinds of data communicated and the properties required for communication to succeed in the presence of faults.

These formal expressions are stated in the language of higher-order logic along with a small algebraic datatype. The abstractions have been formulated in PVS. They are available on-line [13]. There, some additional properties not described herein are stated and proved.

1. The Synchronous Model

We present a generic synchronous model for modeling synchronous protocols. The synchronous model presented is a *functional model* of a protocol's behavior: a protocol's behavior is specified as a recursive function. The abstractions described in Sections 2 through 4 assume an underlying functional model. A *relational model* is a more abstract specification of protocol behavior that simply states assumptions and requirements. The abstractions in Section 5 are relational abstractions. The functional models can then be shown to imply the relational assumptions; that is, the relational assumptions are shown to be invariants of the functional models.

¹The synchronous model formulated in PVS has minor syntactical differences with that presented in Section 1.

1.1. Syntax. The signature of a synchronous protocol comes from Rushby’s definition [10], which adapts Lynch’s definition [29] for the purpose of formulation in a mechanical theorem-prover.

We begin by fixing a set of messages, $mess$. A distinguished element $null$ represents the absence of a message (it can also represent a “do not care” message). Let P be a nonempty set of node identifiers. For each $p \in P$, the following total functions are defined:

- A set of node identifiers, out_nbrs_p , identifying the *outbound neighbors*; i.e., the nodes to which p is connected by outbound channels. A set of node identifiers, in_nbrs_p , identifying the *inbound neighbors*, can be defined from the outbound neighbors:

$$in_nbrs_p \stackrel{\text{df}}{=} \{q \in P \mid p \in out_nbrs_q\} .$$

- A set of states, $states_p$. A distinguished component of the state, r , keeps track of the current round. The set $init_states_p$ is a nonempty subset of $states_p$ containing the *initial states*.
- A *message-generation function* $msg_p : states_p \times out_nbrs_p \rightarrow mess$ that returns the message p sends to each node to which it is connected by an outbound channel; $null$ is returned if no message is sent.
- A higher-order *state-transition function*

$$trans_p : states_p \times IN_p \rightarrow states_p$$

that returns the new state of p based on the current state and inputs IN_p generated by its inbound neighbors.

Not every set or function needs to be interpreted in a protocol specification. For example, some communication protocols can be used for arbitrary data values, so the set $mess$ need not be interpreted.

Sometimes we omit the node-identifier subscript from a set or function to denote a global representation of the system. For example, we define the *global state* to be the function $states \stackrel{\text{df}}{=} \lambda p. states_p$.

1.2. Semantics. The semantics can be given by a transition system. A synchronous protocol proceeds in rounds. In a round, nodes synchronously and instantaneously update their outbound channels (in the communication phase) and then their local state (in the computation phase) [29]. The communication phase is modeled by each node applying its *msg* function, and the computation phase is modeled by each node applying its *trans* function.

A synchronous specification has a simple operational semantics given by a recursive function. It takes the number of rounds of execution, the global initial state, and returns the final global state. Thus, a protocol can be defined as the function $run(init_rnd, init_s)$, where

$$run(r, s) \stackrel{\text{df}}{=} \begin{cases} s & \text{if } r = 0 \\ \lambda p. trans_p(run_p(r - 1, s), \lambda q. msg_q(run_q(r - 1, s), p)) & \text{else} \end{cases}$$

The protocols modeled execute for only a finite number of rounds. However, a protocol may be scheduled to execute an infinite number of times.

Proving fault-tolerance requires the specification of not only the behavior of non-faulty nodes in the model described above but also the behavior of faulty nodes. How to model faulty behavior is in the synchronous model addressed in Section 3.

2. Abstracting Messages

2.1. Abstraction. Messages communicated in a distributed system are abstracted according to some correctness criteria agreed upon by the nodes. We distinguish between *benign messages* and *accepted messages*. The former are messages that a non-faulty receiving node recognizes as incorrect; the latter are messages that a non-faulty receiving node does not recognize as incorrect. Note that an accepted message may be incorrect: the receiving node just does not *detect* that the message is incorrect.

Benign messages abstract various sorts of misbehavior. A message that is sufficiently garbled during transmission may be caught by an error-checking code [45] and deemed benign. Benign messages also abstract the absence of a message: a receiver expecting a message but detecting the absence of one takes this to be the ‘reception’ of a benign message. In synchronized systems with global communication schedules, they abstract messages sent and received at unscheduled times.

2.2. Formulation. Let the set MSG be a set of messages of a given type. MSG is the base set of elements over which the datatype is defined. The set of all possible datatype elements is denoted by $ABSTRACT_MSG[MSG]$. In specifying a protocol in the synchronous model using the syntax described in Section 1.1, $ABSTRACT_MSG[MSG]$ is an interpretation of the set of messages, $mess$.

The datatype has two constructors, *accepted_msg* and *benign_msg*. The former takes an element $m \in MSG$ and creates the datatype element *accepted_msg*[m]. The constructor also has an associated extractor *value* such that

$$value(accepted_msg[m]) = m .$$

Constructors	Extractors	Recognizers
$accepted_msg[m]$	$value$	$accepted_msg?$
$benign_msg$	$none$	$benign_msg?$

FIGURE 1. Abstract Messages Datatype

The other constructor, $benign_msg$, is a constant datatype element; it is a constructor with no arguments. All benign messages are abstracted as a single message; thus, the abstracted incorrect message cannot be recovered, and distinct inbound messages considered to be benign are treated identically. Finally, we define two recognizers, $accepted_msg?$ and $benign_msg?$ with the following definitions. Let $a \in ABSTRACT_MSG[MSG]$.

$$accepted_msg?(a) \stackrel{\text{df}}{=} \exists m. m \in MSG \text{ and } a = accepted_msg[m] ,$$

and

$$benign_msg?(a) \stackrel{\text{df}}{=} a = benign_msg .$$

We summarize this datatype in Fig. 1.

3. Abstracting Faults

There are two closely-related abstractions with respect to faults. The first abstraction, *error types*, partitions the possible locations of faults. The second abstraction, *fault types*, partitions faults according to the manifestation of the errors caused by the faults.

3.1. Abstracting Error Types. Picking the right level of abstraction and the right components to which faults should be attributed is a modeling issue that has been handled in a variety of ways. The choice made is a particularly good example

of the extent to which modeling choices can affect the ease of specification and proof efficacy.

Both nodes and channels can suffer faults [29], but reasoning about node and channel faults together is tedious. Such reasoning is redundant – channel faults can be abstracted as node faults. A channel between a sending node and a receiving node can be abstracted as being an extension either of the sender or of the receiver. For instance, a lossy channel abstracted as an extension of the sender is modeled as a node failing to send messages.²

Even if we abstract all faults to ones affecting nodes and not channels, we are left with the task of abstracting how the functionality of a node – sending, receiving, or computing – is degraded. One possibility is to consider a node as an indivisible unit so that a fault affecting one of its functions is abstracted as affecting its other functions, too. Another possibility is to abstract all faults to ones affecting a node’s ability to send and receive messages [10, 26]. Finally, some models implicitly abstract node faults as being ones affecting *only* a node’s ability to send messages [17, 57]. So even if a fault affects a node’s ability to receive messages or compute, the fault is abstractly propagated to a fault affecting the node’s ability to send messages.

All three models above are *conservative*, i.e., the abstraction of a fault is at least as severe as the fault. Conservation holds for the first model in which the whole node is considered to be degraded by any fault, and it holds for the second model, too. Even though it is assumed that a node can always compute correctly, its computed values are inconsequential if it can neither receive nor send correct values. As for the

²The reader may recall that in the theory of distributed systems, an impossibility result known as the “Coordinated Attack Problem” holds for consensus in models with faulty communication channels that does not necessarily hold in models with node faults [29]. This abstraction does not contradict this result. Coordinated attack is impossible in this model, too, because a node’s messages mimic the effects of a lossy channel.

third model, the same reasoning applies – even if a faulty node can receive messages *and* compute correctly, it cannot send its computations to other nodes.

The model we choose is one in which all faults are abstracted to be ones degrading send functionality, and in which channels are abstracted as belonging to the sending node. There are two principal advantages to this model, both of which lead to simpler specifications and proofs. First, the model allows us to disregard faults when reasoning about the ability of nodes to receive and compute messages. Second, whether a message is successfully communicated is determined solely by a node’s send functionality; the faultiness of receivers need not be considered.

3.2. Abstracting Fault Types. Faults result from innumerable occurrences including physical damage, electromagnetic interference, and “slightly-out-of-spec” communication [48]. We collect these fault occurrences into *fault types* according to their effects in the system.

We adopt the *hybrid fault model* of Thambidurai and Park [38]. A node is called *benign*, or *manifest*, if it sends only benign messages, as described in Sect. 2. A node is called *symmetric* if it sends every receiver the same message, but these messages may be incorrect. A node is called *asymmetric*, or *Byzantine* [39], if it sends different messages to different receivers. All non-faulty nodes are also said to be *good*.

Other fault models exist that provide more or less detail than the hybrid fault model above. The least detailed fault model is to assume the worst case scenario, that all faults are asymmetric. The fault model developed by Azadmanesh and Kieckhafer [96] is an example of a more refined model. All such fault models are consistent with the other abstractions in this paper.

3.3. Formulation. We formulate fault types first. Let S and R be sets of nodes sending and receiving messages, respectively, in a round of communication. Let *asym*,

sym, *ben*, and *good* be constants representing the fault types asymmetric, symmetric, benign, and good, respectively.

As mentioned, we abstract all faults to ones that affect a node's ability to send messages. To model this formally, we modify the message-generation function described in Section 1.1 according to the fault-status of the sender. The range of the function is the set of abstract messages, which are elements of the datatype defined in Sect. 2. *MSG* is a set of messages, and *ABSTRACT_MSG[MSG]* is the set of datatype elements parameterized by *MSG*. Let $p, q \in P$ be a sending and receiving node, respectively, let $m \in MSG$, and let s_p be p 's state. Let *sender_status* be a function mapping senders to their fault partition. The function outputs the abstract message p sends to q :

$$msg_p(s_p, q) \stackrel{\text{df}}{=} \begin{cases} \text{accepted_msg}[m] & \text{if } sender_status(p) = good \\ \text{benign_msg} & \text{if } sender_status(p) = ben \\ \text{sym_msg}(s_p) & \text{if } sender_status(p) = sym \\ \text{asym_msg}(s_p, q) & \text{if } sender_status(p) = asym . \end{cases}$$

If p is good, then q receives an accepted abstract message from p . If p is benign, then q receives a benign message. In the last two cases – in which p suffers a symmetric or asymmetric fault – uninterpreted functions are returned. Applied to their arguments, *sym_msg* and *asym_msg* are uninterpreted constants of the abstract message datatype. The function *asym_msg* models a node suffering an asymmetric fault by taking the receiver as an argument: for receivers q and r , *asym_msg*(s_p, q) is not necessarily equal to *asym_msg*(s_p, r). On the other hand, the function *sym_msg* does not take a receiver as an argument, so all receivers receive the same arbitrary abstract message from a particular sender.

4. Abstracting Fault-Masking

4.1. Abstraction. Some of the information a node receives in a distributed system may be incorrect due to the existence of faults as described in Sect. 3. A node must have a means to mask incorrect information generated by faulty nodes. Two of the most common are (variants of) a majority vote or a middle-value selection, as defined in the following paragraph. These functions are similar enough to abstract them as a single fault-masking function.

A majority vote returns the majority value of some multiset (i.e., a set in which repetition of values is allowed), and a default value if no majority exists. A middle-value selection takes the middle value of a linearly-ordered multiset if the cardinality of the multiset is odd. If the cardinality is an even integer n , then the natural choices are to compute one of (1) the value at index $\lfloor n/2 \rfloor$, (2) the value at index $\lceil n/2 \rceil$, or (3) the average of the two values from (1) and (2). Of course, these options may yield different values; in fact, (3) may yield a value not present in the multiset.

For example, for the multiset $\{1, 1, 2, 2, 2, 2\}$, the majority value is 2, and the middle-value selection is also 2 for any of the three ways to compute the middle-value selection. For any multiset that can be linearly-ordered, if a majority value exists, then the majority value is equal to the middle-value selection (for any of the three ways to compute it mentioned above).

The benefit of this abstraction is that we can define a single fault-masking function (we call it a *fault-masking vote*) that can be implemented as either a majority vote or a middle-value selection (provided the data over which the function is applied is linearly-ordered).

The abstraction allows us to model distinct fault-tolerant distributed algorithms uniformly. Concretely, this abstraction, coupled with the other abstractions described in this paper, allow certain clock synchronization algorithms (that depend

on a middle-value selection) and algorithms in the spirit of an Oral Messages algorithm [17, 39] (that depend on a majority vote) to share the same underlying models [27].

4.2. Formulation. The formulation we describe models a majority vote and a middle-value selection over a multiset. A lemma stating their equivalence follows. Definitions of standard and minor functions are omitted. The formulation is useful for interpreting the state-transition function when specifying a protocol in the language presented in Section 1.1. A node executing a fault-tolerant protocol may make some fault-making computation as part of its state-transition.

Based on the NASA Langley Research Center PVS *Bags* Library [77], a multiset is formulated as a function from values to the natural numbers that determines how many times a value appears in the multiset (values not present are mapped to 0). Thus, let V be a nonempty finite set of values,³ and let $ms : V \rightarrow \mathbb{N}$ be a multiset.

To define a majority vote, we define the cardinality of a multiset ms to be the summation of value-instances in it:

$$|ms| \stackrel{\text{df}}{=} \sum_{v \in V} ms(v) .$$

The function maj_set takes a multiset ms and returns the set of majority values in it.

$$maj_set(ms) \stackrel{\text{df}}{=} \{v \mid 2 \times ms(v) > |ms|\} .$$

This set is empty if no majority value exists, or it is a singleton set. Thus, we define *majority* to be a function returning the special constant *no_majority* if no majority

³If V is finite, then multisets are finite. Fault-masking votes can only be taken over finite multisets.

value exists and the single majority value otherwise.

$$majority(ms) \stackrel{\text{df}}{=} \begin{cases} no_majority & : \quad maj_set(ms) = \emptyset \\ \epsilon(maj_set(ms)) & : \quad \text{otherwise} . \end{cases}$$

The function ϵ is the choice operator that takes a set and returns an arbitrary value in the set if the set is nonempty. Otherwise, an arbitrary value of the same type as the elements in the set is returned [93].

Now we formulate a middle-value selection. Let V have the linear order \preceq defined on it. The function mid_val_set takes a multiset and returns the set of values at index $\lfloor n/2 \rfloor$ when the values are ordered from least to greatest (the ordering is arbitrary). The set is always a singleton set.

$$mid_val_set(ms) \stackrel{\text{df}}{=} \left\{ v \mid \begin{array}{l} 2 \times |lower_filter(ms, v)| > |ms| \\ \text{and } 2 \times |upper_filter(ms, v)| \geq |ms| \end{array} \right\} .$$

The function $lower_filter$ filters out all of the values of ms that are less than or equal to v , and $upper_filter$ filters out the values greater than or equal to v . The function $lower_filter$ is defined as follows:

$$lower_filter(ms, v) \stackrel{\text{df}}{=} \lambda i. \begin{cases} ms(i) & : \quad i \preceq v \\ 0 & : \quad \text{otherwise} . \end{cases}$$

Similarly,

$$upper_filter(ms, v) \stackrel{\text{df}}{=} \lambda i. \begin{cases} ms(i) & : \quad v \preceq i \\ 0 & : \quad \text{otherwise} . \end{cases}$$

The relation $mid_val_set(ms)$ is guaranteed to be a singleton set, so using the function ϵ mentioned above, we can define $middle_value$ to return the middle value of a multiset:

$$middle_value(ms) \stackrel{\text{df}}{=} \epsilon(mid_val_set(ms)) .$$

The following theorem results.

THEOREM 3.1 (Middle Value is Majority).

$$\begin{aligned} &majority(ms) \neq no_majority \\ &\text{implies } middle_value(ms) = majority(ms) . \end{aligned}$$

5. Abstracting Communication

We identify two abstractions with respect to communication. First, we abstract the kinds of data communicated. Second, we identify the fundamental conditions that must hold for communication to succeed.

We distinguish between a *functional model* and a *relational model* of communication. In the former, communication is modeled computationally (e.g., using functions like *send* from Sect. 3). In the latter more abstract model, conditions on communication are stated such that if they hold, communication succeeds. This section presents a relational model of communication.

5.1. Abstracting Kinds of Communication. Some kinds of information can be modelled by real valued, uniformly continuous functions. Informally, a function is uniformly continuous if small changes in its argument produce small changes in its result; see e.g., Rosenlicht [97]. For example, the values of analog clocks and of thermometers vary with time, and the rate of change is bounded. In a distributed system, a node may *sample* such a function, i.e., determine an approximation of the function's value for a given input. We call such functions *inexact functions* and the communication of their values *inexact communication*. We call discrete functions, such as an array sorting algorithm, *exact functions* and communication involving them *exact communication*.

5.2. Abstracting Communication Conditions. Communication in a fault-tolerant distributed system is successful if *validity* and *agreement* hold. For exact communication, their general forms are:

Exact Validity: A good receiver’s fault-masking vote is equal to the value of the function good nodes compute.

Exact Agreement: All good nodes have equal fault-masking votes.

For inexact communication we have similar conditions:

Inexact Validity: A good receiver’s fault-masking vote is bounded above and below by the samples from good nodes, up to a small error margin.

Inexact Agreement: All good nodes differ in their fault-masking votes by at most a small margin of error.

A validity property can thus be understood as an agreement between senders and receivers, whereas an agreement property is an agreement between the receivers. We limit our presentation to guaranteeing validity. Agreement is treated similarly, and complete PVS formulations and proofs for both are located on-line [13].

We specifically present conditions that guarantee validity holds after a single broadcast communication round in which each node in a set of senders sends messages to each node in a set of receivers (a degenerate case is when these are singleton sets modeling point-to-point communication between a single sender and receiver). A functional model of a specific communication protocol can then be shown to satisfy these conditions.

First we describe how a single round of exact communication satisfies exact validity, provided that the three conditions *Majority Good*, *Exact Function*, and *Function Agreement* hold. The three conditions state, respectively, that the majority of the values over which a vote is taken come from good senders, that good senders compute

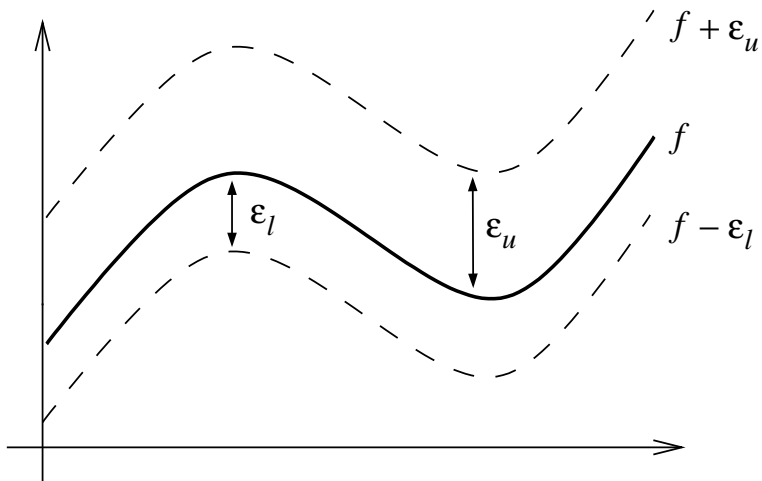


FIGURE 2. The Inexact Function Condition for Inexact Communication

functions exactly (i.e., there is no approximation in sampling an exact function), and that every good sender computes the same function.

For a single round of inexact communication, we have inexact validity if the two conditions Majority Good and Inexact Function hold. Majority Good is the same as above. The *Inexact Function* condition bounds the variance allowed between the sample of an inexact function computed by a good node for a given input and the actual value of the function for that input. That is, let e_l and e_u be small positive constants representing the lower and upper variation, respectively, allowed between an inexact function f and potential samples of it as depicted in Fig. 2. The sample computed by a good node is bounded by $f - e_l$ and $f + e_u$. We do not present an analog to the Function Agreement condition in the inexact case since nodes often compute and vote over possibly different functions. For example, each node might possess a distinct local sensor that it samples. It is assumed, however, that the functions are related, e.g., each sensor measures the same aspect of the environment.

Clock synchronization [29] is an important case of inexact communication. Clocks distributed in the system need to be synchronized in order to avoid drifting too far apart. In this case, sampling a local clock yields an approximation of time.

5.3. Formulation for Exact Communication. First we present the model of a round of exact communication. For a single round of communication, let S be the set of senders sending in that round. Let $good_senders \subseteq S$ be a subset of senders that are good. This set can change as nodes become faulty and are repaired, so we treat it as a variable rather than a constant. For an arbitrary receiver,⁴ let $eligible_senders \subseteq S$ be the set of senders trusted by the receiver (we assume that receivers trust all good senders). Then the condition *Majority Good* is defined

$$\begin{aligned} majority_good(good_senders, eligible_senders) &\stackrel{\text{df}}{=} \\ &2 \times |good_senders| > |eligible_senders| \\ &\text{and } good_senders \subseteq eligible_senders . \end{aligned}$$

The condition stipulates that a majority of the senders in $eligible_senders$ are in $good_senders$.

Next we describe the values sent and received. Let MSG be the range of the function computed – these are the messages communicated. The variable $ideal : S \rightarrow MSG$ maps a sender to the exact value of a function to be computed by the sender, for a given input. Doing so frees us from representing the particular function computed. Similarly, $actual : S \rightarrow MSG$ maps a sender to the value that sender computes for the same function and input. Good senders compute exact functions exactly:

$$\begin{aligned} exact_function(good_senders, ideal, actual) &\stackrel{\text{df}}{=} \\ &\forall s. s \in good_senders \text{ implies } ideal(s) = actual(s) . \end{aligned}$$

⁴The receiver can be *any* receiver, good or faulty. The abstractions described in Sect. 3 allow us to ignore the fault status of receivers in formal analysis.

Function Agreement states that the functions computed by any two good senders is the same (i.e., they send the same messages).

$$\begin{aligned} & \text{function_agreement}(\text{good_senders}, \text{ideal}) \stackrel{\text{df}}{=} \\ & \forall s_1, s_2. s_1 \in \text{good_senders} \text{ and } s_2 \in \text{good_senders} \\ & \text{implies } \text{ideal}(s_1) = \text{ideal}(s_2) . \end{aligned}$$

Before stating the validity result, we must take care of a technical detail with respect to forming the multiset of messages over which a receiver takes a fault-masking vote. For an arbitrary receiver, let the function *make_bag* take as arguments a nonempty set *eligible_senders* and a function mapping senders to the message the receiver gets. It returns a multiset of received messages from senders in *eligible_senders*.

$$\begin{aligned} & \text{make_bag}(\text{eligible_senders}, \text{actual}) \stackrel{\text{df}}{=} \\ & \lambda v. \left| \{s \mid s \in \text{eligible_senders} \text{ and } \text{actual}(s) = v\} \right| . \end{aligned}$$

For exact messages, validity is the proposition that for any good sender, the exact value of the function it is to compute is the value computed by the receiver's fault-masking vote. The proposition is defined as follows:

$$\begin{aligned} & \text{exact_validity}(\text{eligible_senders}, \text{good_senders}, \text{ideal}, \text{actual}) \stackrel{\text{df}}{=} \\ & \forall s. s \in \text{good_senders} \\ & \text{implies } \text{ideal}(s) = \text{majority}(\text{make_bag}(\text{eligible_senders}, \text{actual})) . \end{aligned}$$

We use *majority* for the fault-masking vote, but middle-value selection is acceptable given Thm. 3.1 (provided the data is ordered). Using *majority*, the Exact Validity Theorem reads:

THEOREM 3.2 (Exact Validity).

$majority_good(good_senders, eligible_senders)$
 and $exact_function(good_senders, ideal, actual)$
 and $function_agreement(good_senders, ideal)$
 implies $exact_validity(eligible_senders, good_senders, ideal, actual)$.

5.4. Formulation for Inexact Communication. Next we model a round of inexact communication. The *Majority Good* condition is formulated in the same way as for exact communication. To define *Inexact Function*, we now assume that the elements of MSG have at least the structure of an additive group linearly ordered by \preceq . *Inexact Function* is defined as the conjunction of two conditions, *Lower Function Error* and *Upper Function Error*. These two conditions specify, respectively, the maximal negative and positive error between the exact value of an inexact function and a good sender's approximation of the inexact function, for a given input.

$lower_function_error(good_senders, ideal, actual) \stackrel{\text{df}}{=} \\ \forall s. s \in good_senders \text{ implies } ideal(s) - e_l \preceq actual(s) ;$

$upper_function_error(good_senders, ideal, actual) \stackrel{\text{df}}{=} \\ \forall s. s \in good_senders \text{ implies } actual(s) \preceq ideal(s) + e_u ;$

$inexact_function(good_senders, ideal, actual) \stackrel{\text{df}}{=} \\ lower_function_error(good_senders, ideal, actual) \\ \text{and } upper_function_error(good_senders, ideal, actual) .$

For inexact communication, validity is the proposition that for a fixed receiver, the value determined by a fault-masking vote is bounded both above and below by the messages received from good senders, modulo error values e_l and e_u . Each sender

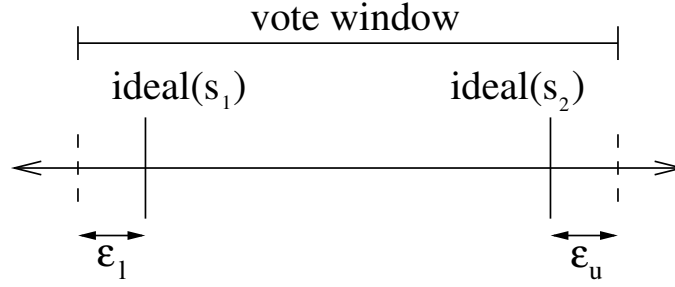


FIGURE 3. Inexact Validity

may be computing a different inexact function, so the vote window depends on both the functions computed as well as the errors in approximating them. Inexact validity is illustrated in Fig. 3, where s_1 and s_2 are good senders.

$$\begin{aligned}
 \text{inexact_validity}(\text{eligible_senders}, \text{good_senders}, \text{ideal}, \text{actual}) &\stackrel{\text{df}}{=} \\
 \exists s_1. \quad &s_1 \in \text{good_senders} \\
 \text{and } &\text{ideal}(s_1) - e_l \\
 &\preceq \text{middle_value}(\text{make_bag}(\text{eligible_senders}, \text{actual})) \\
 \text{and } \exists s_2. \quad &s_2 \in \text{good_senders} \\
 \text{and } &\text{middle_value}(\text{make_bag}(\text{eligible_senders}, \text{actual})) \\
 &\preceq \text{ideal}(s_2) + e_u .
 \end{aligned}$$

The Inexact Validity Theorem then reads:

THEOREM 3.3 (Inexact Validity).

$$\begin{aligned}
 &\text{majority_good}(\text{good_senders}, \text{eligible_senders}) \\
 &\text{and } \text{inexact_function}(\text{good_senders}, \text{ideal}, \text{actual}) \\
 &\text{implies } \text{inexact_validity}(\text{eligible_senders}, \text{good_senders}, \text{ideal}, \text{actual}) .
 \end{aligned}$$

6. Summary

The abstractions, in the language of higher-order logic, pertain to messages, faults, fault-masking, and communication in the formal specification and verification of fault-tolerant distributed protocols.

These abstractions describe a means by which to systematically specify and verify synchronous protocols using a mechanical theorem-prover. Because these abstractions model synchronous protocols, timing abstractions have been ignored.⁵ In the next chapter, we describe how temporal abstractions for a class of partially-synchronous protocols called time-triggered protocols.

⁵They have not been completely ignored, however. For example, the abstractions of faults described in Section 3 pertain to timing faults as well as data faults.

CHAPTER 4

Time-Triggered Protocol Verification

In the previous chapter, a set of abstractions for the specification and verification of synchronous distributed protocols was described. These abstractions form the basis for verifying the time-triggered protocols implemented in SPIDER [27]. The synchrony assumption makes the verification feasible, but not trivial.

As shown by Rushby [10], the class of time-triggered protocols can be systematically shown to implement their synchronous specifications. Rushby develops a particular timed model called the time-triggered model. It is a restriction of the partially-synchronous model in which there are lower and upper bounds on the duration of communication and computation. The essential feature of this model is that while the local clocks of individual nodes may not be perfectly synchronized, their disharmony is bounded. Good reasons exist for why safety-critical real-time protocols should satisfy these constraints, as described in Section 2.2, and many do, including those implemented in TTA and SPIDER [7, 11].

Rushby's presentation suffers two shortcomings. First, despite his formal verification of the time-triggered model in the mechanical theorem-prover PVS, three of the four system assumptions (or axioms) not only fail to model the actual behavior of time-triggered systems, but are in fact inconsistent [28]. We mend these assumptions as well as remove redundant axioms in the theory in Appendix A.

Second, the model is too constrained to model many protocol implementations of time-triggered systems. We generalize the theory for the specification and verification of implemented time-triggered systems. The generalization aims to give a

concrete answer to the question, “What are the most general constraints under which the synchrony assumption is satisfied?” The theory is extended along the following dimensions:

- event-triggered behavior,
- communication delays,
- reception windows,
- non-static clock skew, and
- pipelined rounds.

Some protocols occasionally manifest event-triggered behavior. A typical example is a clock synchronization protocol such as Davies and Wakerly’s protocol [98] or Srikanth and Toueg’s protocol [99]. Some of the messages sent in the protocol may be determined by the global schedule, but others are event-triggered: when a node receives some number of messages over its inbound channels, it sends a synchronization (or *echo*) message. Our theory should be rich enough to model this behavior; see Section 3.2 for an application.

Provisions for reasoning about non-static clock skew have two benefits. First, they allow protocols that satisfy the assumptions of the time-triggered model but nevertheless directly affect the system’s timing characteristics (e.g. clock synchronization, self-stabilization, and startup protocols) to be specified in a synchronous model. Second, they allow for formal reasoning about schedule optimizations. Time-triggered system schedules (also known as *task-descriptor lists* [8]) are usually designed with respect to the maximum possible clock skew during the normal operation of the system. When clocks are not resynchronized, the maximum possible clock skew increases as a function of time. If the difference between possible clock skew at different points in the system’s execution is significant, then a schedule can be tightened at those points that the clock skew is small.

It is possible to pipeline the communication and computation rounds of a single protocol or of multiple protocols for better throughput; we call pipelining of this sort *round-based pipelining*. Embedded control systems often have hard real-time deadlines that may require aggressive schedules. Extensions to reason about pipelined communication allows these sort of designs to be modeled and explored for non-pipelined systems.

In the second half of this chapter, we demonstrate how to formally verify that an implemented protocol schedule for a realized system satisfies the constraints necessary to abstract the protocol synchronously. The verification is highly automated using SAL. Verifying that a schedule satisfies the constraints is accomplished by the k -induction proof technique. This methodology allows one to mechanically check that a schedule satisfies the time-triggered model constraints and to optimize the schedule. Not only does this provide for the possibility of implementing a tighter schedule, but in a fault-tolerant system, a tighter schedule can increase a system's ability to detect timing faults [48].

In Section 1, we revise the formal theory of time-triggered systems developed by Rushby [10]. In Section 2, extensions are given to the theory as well as proofs that the extended theory satisfies the synchrony assumption. The modeling and verification of an implementation schedule in SAL is described in Section 3. Two case-studies applying these techniques are described. In the first case-study, a schedule for the SPIDER Distributed Diagnosis Protocol is verified; in the second, a schedule for the SPIDER Clock Synchronization Protocol is verified. The specifications and proofs in PVS and SAL associated with this chapter can be found on-line [13].

1. The Time-Triggered Model

We review Rushby’s model [10]. In this presentation, we have mended the inconsistent axioms and modified the proofs accordingly, removed redundant axioms, and formalized the presentation slightly (by clearly distinguishing the syntax and the semantics of the model). Theorem 4.15 demonstrates that a simulation relation exists between this time-triggered model and an untimed synchronous model.

This theory of time-triggered systems does not describe the effects of faulty nodes and communication channels. Abstractions for modeling these in the synchronous model are described in Chapter 3. The intention of this theory is to demonstrate the schedule constraints nonfaulty nodes must satisfy – indeed, these constraints characterize nonfaulty temporal behavior in the time-triggered model. The theory is not used to verify the correctness of protocols specified in the theory. Therefore, in the remainder of this chapter, all nodes and communication channels are considered to be nonfaulty.

1.1. Syntax. The signature of a time-triggered protocol extends the signature for a protocol specification in the synchronous model from Section 3.1. We define *real time* to be the set of real numbers \mathbb{R} and *clock-time* to be the set of integers \mathbb{Z} . Real time is measured in some arbitrary unit of time (e.g. milliseconds), and clock-time is measured in ticks, where one tick is one unit on the integral number line.

The synchronous model in Section 1 is augmented with the following declarations:

- An *inverse clock* function $C_p : \mathbb{R} \rightarrow \mathbb{Z}$ that takes a real time as an input and returns a clock-time, for each node p .
- A *schedule* function $sched : \mathbb{N} \rightarrow \mathbb{Z}$ from rounds to clock-time. It determines the clock-time at which nodes should execute some instruction.

- A *communication offset* function $D : \mathbb{N} \rightarrow \mathbb{Z}$ from rounds to a clock-time offset. It determines the clock-time at which nodes send messages in each round.
- A *computation offset* function $P : \mathbb{N} \rightarrow \mathbb{Z}$ from rounds to a clock-time offset. It determines the clock-time at which nodes begin computation in each round.
- A *clock drift rate* $\rho \in \mathbb{R}$ such that $0 < \rho < 1$. This is the maximum rate at which a clock may drift.
- A maximum *clock skew* $\Sigma \in \mathbb{N}$ that is a nonnegative clock-time denoting the maximum difference between the clocks of individual nodes (described below).
- A maximum *communication delay* $\delta \in \mathbb{R}^{0 \leq}$ denoting a nonnegative real number that is the maximum communication delay between when messages are sent and received by nodes.

We constrain the interpretations that can be given to the syntax when defining a time-triggered system with the following axioms. These axioms are divided into two groups, *system assumptions* and *schedule constraints* (or simply constraints). The system assumptions describe the assumed behavior of the underlying system, most notably, the behavior of the local clocks. The schedule constraints ensure the schedule of time-triggered events, given the system assumptions, gives rise to a time-triggered realization.

1.1.1. *System Assumptions.* Axiom 4.1 states the maximum clock drift as a function of the maximum drift rate, ρ . Axiom 4.2 ensures the skew between clocks is no greater than the maximum clock skew, Σ . Axiom 4.3 ensures that messages are received within the communication delay of when they are sent and that messages received were not “spontaneously generated.”

AXIOM 4.1 (Clock Drift Rate). *Let $t_1 \geq t_2$. Then $\lfloor (1 - \rho) \cdot (t_1 - t_2) \rfloor \leq C_p(t_1) - C_p(t_2) \leq \lceil (1 + \rho) \cdot (t_1 - t_2) \rceil$.*

AXIOM 4.2 (Clock Synchronization). $|C_q(t) - C_p(t)| \leq \Sigma$.

AXIOM 4.3 (Maximum Communication Delay). *There exists some $0 \leq d \leq \delta$ such that $\text{sent}_p(q, m, t)$ implies $\text{recv}_q(p, m, t + d)$, and there exists some $0 \leq d' \leq \delta$ such that $\text{recv}_q(p, m, t)$ implies $\text{sent}_p(q, m, t - d')$.*

LEMMA 4.4. $t_1 < t_2$ implies $C_p(t_1) \leq C_p(t_2)$.

PROOF. By Axiom 4.1, $C_p(t_2) \geq C_p(t_1) + \lfloor (1 - \rho)(t_2 - t_1) \rfloor$. □

1.1.2. *Schedule Constraints.* Axiom 4.5 orders the communication phase and the computation phase for each round. Axiom 4.6 ensures that the communication offset is strictly greater than the maximum skew. Axiom 4.7 ensures that the computation phase is sufficiently greater than the communication offset.

AXIOM 4.5 (Offset Constraint). *Let $\text{dur}(r) \stackrel{\text{df}}{=} \text{sched}(r + 1) - \text{sched}(r)$. Then $0 < D(r) < P(r) < \text{dur}(r)$.*

This axiom is illustrated in Figure 1.

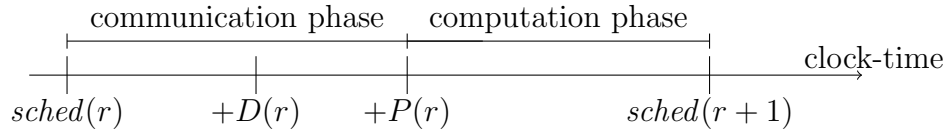


FIGURE 1. Axiom 4.5

AXIOM 4.6 (Communication Constraint). $D(r) \geq \Sigma$.

AXIOM 4.7 (Computation Offset Constraint). $P(r) > D(r) + \Sigma + \lceil 1 + \rho \rceil \cdot \delta$.

1.2. Semantics. The semantics for the time-triggered model is a transition system in which states are pairs of the form $\langle s, t \rangle$, where s is a global state of the system together with the current real time, t . The transitions between states are determined by an axiomatization. First, some uninterpreted functions and relations are stated to aid in this axiomatization. Although these functions and relations are uninterpreted, we state their intended meaning that is provided by the axiomatization that follows.

- A relation $sent_p \subseteq out_nbrs_p \times mess \times \mathbb{R}$, the tuples of which consist of a node q (that is an outbound neighbor of p), a message m , and a real time t and holds if p sent message m to q at real time t .
- A relation $recv_p \subseteq out_nbrs_p \times mess \times \mathbb{R}$, the tuples of which consist of a node q (that is an inbound neighbor of p), a message m , and a real time t and holds if p received message m from q at real time t .
- A function $sendtime_p : \mathbb{N} \rightarrow \mathbb{R}$ from rounds to real times denoting the real time p broadcast messages in each round.
- A *time-triggered system state* function $ttss_p(init_s)(T)$ that takes a global initial state $init_s$, a clock-time T , and returns node p 's state after executing for T clock ticks from the initial state.
- A *time-triggered inbound messages* function $ttin_p : \mathbb{Z} \times out_nbrs_p \rightarrow mess$ that maps a clock-time T and an inbound neighbor q to the message p receives from q at T .
- A *time-slice* function $gs : \mathbb{N} \rightarrow \mathbb{R}$ from rounds to real times. Its purpose is to provide real times at which the system state of the time-triggered model of a protocol is the same as the untimed model of the protocol, for each round.

Axiom 4.8 constrains the $sendtime_p$ function by ensuring that at the real time that p broadcasts its message in round r , its clock-time is at the communication offset into that round. Axioms 4.9 and 4.10 primarily constrain the behavior of the $sent_p$

function by first stating the sufficient conditions for it to hold and then the necessary conditions for it to hold. Axiom 4.9 ensures that the message p sends to q at the real time $sendtime_p(r)$ is the message generated by its message-generation function using its time-triggered state at the beginning of round r . Axiom 4.10 ensures that if the $sendtime_p$ relation is satisfied, then it is satisfied by a message generated by the message-generation function in some round and by the real time at the communication delay into the round. Axiom 4.11 constrains the behavior of the $ttin_p$ function by ensuring that for any clock-time T in a computation phase of round r , $ttin_p(T, q)$ is the message p receives from q in round r before the computation phase begins (if a message is in fact received). Axioms 4.12 and 4.13 constrain the $ttss_p$ function. Axiom 4.12 determines p 's time-triggered state at the clock-time $sched(r)$, for each round r to be either the initial state for round 0, or by the state-transition function being applied to p 's time-triggered state in the previous round and the time-triggered inbound messages it received during the communication phase in the previous round. Axiom 4.13 ensures that outside of the computation phase, p 's time-triggered state does not spontaneously change. Finally, Axiom 4.14 constrains the real time $gs(r)$ to be the real time at which the process with the slowest clock has reached $sched(r)$.

In the following, let the *pre-computation phase* relation take a real time and a round such that $pre_comp_phase_p(t, r)$ if and only if $sched(r) \leq C_p(t) < sched(r) + P(r)$. Intuitively, the *pre-computation phase* relation holds for a real time t and round r if t is in the round, but before the computation phase begins.

AXIOM 4.8. $C_p(sendtime_p(r)) = sched(r) + D(r)$.

AXIOM 4.9. $sent_p(q, msg_p(ttss_p(init_s)(sched(r))), q, sendtime_p(r))$.

AXIOM 4.10. $sent_p(q, m, t)$ implies there exists a round r such that $t = sendtime_p(r)$ and $m = msg_p(ttss_p(init_s)(sched(r)))$.

AXIOM 4.11. $sched(r) + P(r) \leq T < sched(r + 1)$ implies

$$ttin_p(T, q) = \epsilon(\{m \in mess \mid \exists t \in \mathbb{R}. pre_comp_phase_p(t, r) \text{ and } recv_p(q, m, t)\}) .^1$$

AXIOM 4.12. Let $T = sched(r - 1) + P(r - 1)$. Then

$$\begin{aligned} ttss_p(init_s)(sched(r)) = \\ \text{if } r = 0 \text{ then } init_s_p \\ \text{else } trans_p(ttss_p(init_s)(T), \lambda q. ttin_p(T, q)) . \end{aligned}$$

AXIOM 4.13. $sched(r) \leq T \leq sched(r) + P(r)$ implies

$$ttss_p(init_s)(T) = ttss_p(init_s)(sched(r)) .$$

AXIOM 4.14.

$$\begin{aligned} \forall q : C_q(gs(r)) \geq sched(r) \\ \text{and } \exists p : C_p(gs(r)) = sched(r) . \end{aligned}$$

The transition relation over the states satisfies these axioms.

This axiomatization ensures a simulation relation exists between a synchronous protocol and its time-triggered implementation, as stated in Theorem 4.15.

THEOREM 4.15. $ttss_p(init_s)(C_p(gs(r))) = run_p(r, init_s)$.

PROOF. See [10] for a proof sketch and [13] for a mechanized proof in PVS. \square

2. Extending The Axiomatization

The goal in extending the axiomatization is to generalize the class of time-triggered implementations that can be faithfully abstracted to the synchronous model. Syntax and constraints are added to model the following:

¹The function ϵ is the *choice operator* that takes a set and returns an arbitrary value in the set if the set is nonempty. Otherwise, an arbitrary value of the same type as the elements in the set is returned [93].

- event-triggered behavior where the time at which some action is taken varies between nodes;
- the nominal expected communication delay, as well as lower and upper bounds on the difference between the nominal delay and the actual delays in communication;
- the existence of a *reception window* during which nodes accept incoming messages;
- changes in clock skew over time (the maximum possible skew increases when clocks continue to drift after some initial synchronization, and it may decrease after a resynchronization protocol is executed);
- the pipelining of communication and computation phases.

2.1. Generalized Syntax. The following generalizes the syntax for specifying a time-triggered protocol.

Event Triggering: Rather than a global schedule that marks the beginning of each round, the schedule $sched_p : \mathbb{N} \rightarrow \mathbb{Z}$ is a function from rounds to clock-times, for each node p . It parameterizes the communication and computation phase schedules, defined as offsets from the beginning of the round. We take a more general view of the schedule function now: the schedule function may *determine* the time at which some event occurs, for a time-triggered action, or it may simply denote the clock-time at which an event occurs for an event-triggered action. An application is the SPIDER Clock Synchronization Protocol in Section 3.2. The uninterpreted function $\Lambda : \mathbb{N} \rightarrow \mathbb{Z}$ is a function from rounds to clock-times denoting the maximum clock-time discrepancy between the schedule functions for that round.

Communication Delay: The real-time constant $\delta_{nom} > 0$ denotes the expected nominal delay between when a message is sent and when it is received. The small real-time constants $e_l > 0$ and $e_u > 0$ denote the maximum delays such that a receiver may receive a message at time t , where $\delta_{nom} - e_l \leq t \leq \delta_{nom} + e_u$. We require $e_l < \delta_{nom}$ and $e_u < \delta_{nom}$.

Reception Window: The function $R : \mathbb{N} \rightarrow \mathbb{Z}$ is a function from rounds to a *reception window offset*. It marks the clock-time at which a node accepts inbound messages. In round r , the reception window closes at $P(r)$.

Dynamic Skew: In the execution of a time-triggered system, the maximum skew varies. For example, a clock synchronization protocol resynchronizes clocks and reduces the skew. A small skew is preferable as it allows for a tighter schedule. In the original analysis, a single constant skew is assumed, and this must be set to the largest possible skew.

This can be generalized. In each round $\Sigma(r) \geq 0$ is the greatest clock-time skew occurring between a sender and receiver during the duration of round r .

Pipelined Rounds: In some rounds, the messages sent depends on new state computed in the previous round. In these rounds, no node should send its messages until it completes its computation from the previous round. In rounds where this dependency does not hold, the communication phase of round r can begin before the computation phase of round $r + 1$ ends. We call this *round-based pipelining*, or simply *pipelining*, since this is the only sort of pipelining addressed.

A binary relation, *independent*, over rounds holds if messages to be sent in $r + 1$ do not depend on the computation that occurs during round r . We leave the relation uninterpreted, but constrain its behavior with Axiom 4.22.

The following axioms relax those from Section 1 as well as constrain the possible interpretations of the additional syntax described above. Axioms 4.16 and 4.17 in Section 2.1.1 are generalized system assumptions. Axiom 4.22 is an axiom in the untimed model that describes pipelined behavior. Axioms 4.19 through 4.21 in Section 2.1.3 are generalized schedule constraints, Axiom 4.18 in Section 2.1.2 is a new system assumption, and Axioms 4.23 through 4.25 in Section 2.1.4 are new schedule constraints. The clock drift rate axiom (Axioms 4.1) from Section 1.1 is not generalized below but is assumed to hold.

2.1.1. *Generalized System Assumptions.* Axiom 4.16 generalizes Axiom 4.2. For nodes p and q and real time t , if the node with the faster clock at time t has reached clock-time $\min(\text{sched}_p(r), \text{sched}_q(r)) + D(r)$ and the node with the slower clock has not surpassed the clock-time $\max(\text{sched}_p(r), \text{sched}_q(r)) + P(r)$, then their clocks differ by at most $\Sigma(r)$. Clock-time $\text{sched}_p(r) + D(r)$ is the clock-time at which p sends its messages in round r , and $\text{sched}_p(r) + P(r)$ is the clock-time at which it begins the computation phase of round r . Thus, the clock synchronization assumption ensures that if any clock is in the communication phase of round r , all of the clocks are synchronized within the skew of that round. Because clock synchronization affects only inter-node communication, the computation phase does not need to be similarly constrained. Depending on the schedule, the clocks may be constrained by the skew values for multiple rounds.

AXIOM 4.16 (Clock Synchronization).

$$\begin{aligned} & \left(\max(C_p(t), C_q(t)) \geq \min(\text{sched}_p(r), \text{sched}_q(r)) + D(r) \right. \\ & \quad \left. \text{and } \min(C_p(t), C_q(t)) \leq \max(\text{sched}_p(r), \text{sched}_q(r)) + P(r) \right) \\ & \text{implies } |C_q(t) - C_p(t)| \leq \Sigma(r) . \end{aligned}$$

2.1.2. *New System Assumptions.* Axiom 4.17 is a straightforward generalization of Axiom 4.3 that takes into account communication delay.

AXIOM 4.17 (Maximum Communication Delay). *There exists some $\delta_{nom} - e_l \leq d \leq \delta_{nom} + e_u$ such that $sent_p(q, m, t)$ if and only if $recv_q(p, m, t+d)$, and there exists some $\delta_{nom} - e_l \leq d' \leq \delta_{nom} + e_u$ such that $recv_q(p, m, t)$ if and only if $sent_p(q, m, t-d')$.*

Axiom 4.18 constrains the maximum discrepancy permitted between the schedule functions of two nodes for a given round with respect to the value of the uninterpreted function Λ . For a particular implementation, whether this constraint is met depends on the constraints for the event-triggered behavior of the individual nodes.

AXIOM 4.18. $0 \leq |sched_p(r) - sched_q(r)| \leq \Lambda(r)$.

2.1.3. *Generalized Schedule Constraints.* Axiom 4.5 is weakened by 4.19. Notice that $D(r)$ no longer needs to be positive, so nodes may send messages for round r before that round begins.

AXIOM 4.19 (Offset Constraint). $0 < P(r) < dur(r)$.

Axiom 4.20 weakens Axiom 4.6 insofar as (1) $D(r)$ may be negative, and (2) if the nominal delay is substantially greater than the skew, the skew has little bearing on when messages must be sent (this is often the case in time-triggered systems, which are generally highly-synchronized).

AXIOM 4.20 (Communication Constraint).

$D(r) \geq \Sigma(r) + \Lambda(r) - \lfloor (1 - \rho) \cdot (\delta_{nom} - e_l) \rfloor$.

Axiom 4.21 generalizes Axiom 4.7 by taking into account the nominal communication delay and upper error bound on communication delay.

AXIOM 4.21 (Computation Offset Constraint).

$P(r) > D(r) + \Sigma(r) + \Lambda(r) + \lceil (1 + \rho) \cdot (\delta_{nom} + e_u) \rceil$.

2.1.4. *New Schedule Constraints.* First, we state an axiom in the synchronous model (Section 3.1) that describes the effects of pipelining. The schedule constraints follow.

Axiom 4.22 describes the behavior of pipelined communication and computation phases by stating that in the synchronous model, the execution of rounds $r + 1$ and r can be transposed. Formally,

AXIOM 4.22 (Pipelining).

$$\begin{aligned} & \neg \text{independent}(0) \\ \text{and } & (\text{independent}(r) \\ & \text{implies } (\forall q \in \text{out_nbrs}_p : \\ & \quad \text{msg}_p(\text{run}_p(r, \text{init}_s), q) \\ & \quad = \text{msg}_p(\text{run}_p(r - 1, \text{init}_s), q))) . \end{aligned}$$

The following are new schedule constraints that ensure pipelining works correctly. Axiom 4.23 ensures that pipelining only occurs when the messages to be sent do not depend on the computations from the previous round. Axiom 4.24 restricts pipelining to consecutive rounds. The effect of pipelining is illustrated in Figure 2.

AXIOM 4.23. $\neg \text{independent}(r)$ implies $D(r) \geq 0$.

AXIOM 4.24. $r > 0$ implies $D(r) \geq P(r - 1) - \text{dur}(r - 1)$.

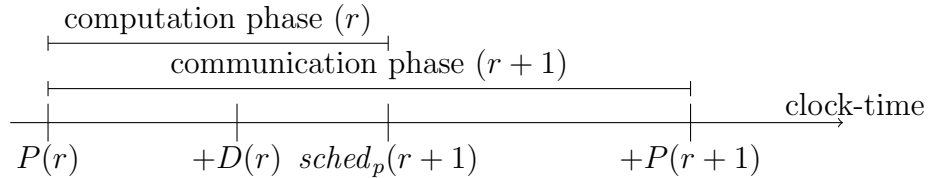


FIGURE 2. Pipelined Communication Phase (Axiom 4.24)

We consider only the case in which the communication phase in a round may begin during the computation phase of the immediately preceding round. More aggressive pipelining is possible. For example, if multiple communication phases overlap, messages sent in the communication phase of one round may arrive after the messages sent in a succeeding phase arrive. In this case, messages should be tagged with their round number, or some other mechanism should be provided to discriminate these inbound messages. Inbound messages also need to be buffered.

Axiom 4.25 constrains the scheduling for when the reception window is opened. The axiom is illustrated in Figure 3. The reception window must be opened soon enough so that any message sent by a node is received within the window. The formula $\lfloor (1 - \rho) \cdot (\delta_{nom} - e_l) \rfloor$ gives a lower bound on the minimum message delay. We add $D(r)$ to take into account the clock-time offset at which the message is sent. The skew for the round, $\Sigma(r)$, is subtracted to account for the case where the receiver's clock is maximally faster than the sender's. A constant of one tick is added to the upper bound on $R(r)$ because the reception window is opened on a clock edge, but messages arrive asynchronously. A message that arrives strictly less than one clock tick before the reception window is opened will be latched on the clock edge when the window is opened. In defining the semantics below, an axiom ensures that messages received during this window are latched appropriately.

AXIOM 4.25 (Reception Window Constraint).

$$0 \leq R(r) \leq D(r) + \lfloor (1 - \rho) \cdot (\delta_{nom} - e_l) \rfloor - \Sigma(r) - \Lambda(r) + 1 .$$

2.2. Generalized Semantics. The semantics are generally the same as those given in Section 1.2, once the corresponding axioms are generalized. Axiom 4.8 is generalized by Axiom 4.26 in which the schedule function is parameterized:

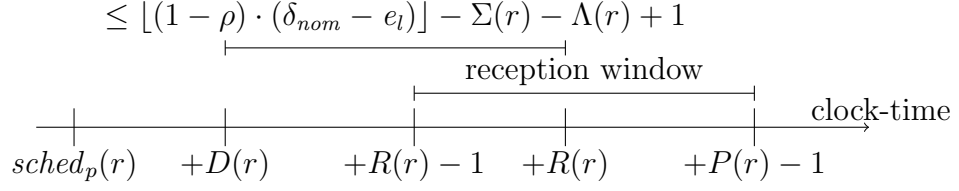


FIGURE 3. The Reception Window (Axiom 4.25)

AXIOM 4.26. $C_p(\text{sendtime}_p(r)) = \text{sched}_p(r) + D(r)$.

Axioms 4.9 and 4.10 are generalized by Axioms 4.27 and 4.28, respectively, to take into account the communication offset:

AXIOM 4.27. $\text{sent}_p(q, \text{msg}_p(\text{tss}_p(\text{gs})(\text{sched}_p(r) + D(r)), q), \text{sendtime}_p(r))$.

AXIOM 4.28. $\text{sent}_p(q, m, t)$ implies there exists a round r such that $t = \text{sendtime}_p(r)$ and $m = \text{msg}_p(\text{tss}_p(\text{init}_s)(\text{sched}_p(r) + D(r)))$.

The relation recv_win_open_p takes a real time t and a round r and is true if the real time falls within p 's reception window for round r .

DEFINITION 4.1 (*Reception Window Open*).

$$\text{recv_win_open}_p(t, r) \stackrel{\text{df}}{=} \text{sched}_p(r) + R_p(r) - 1 \leq C_p(t) < \text{sched}_p(r) + P(r).$$

Recall that the constraint Axiom 4.25 places on $R_p(r)$ is weak enough so that messages from senders may arrive strictly less than one clock tick before $R_p(r)$ is reached, but these messages are latched at $R_p(r)$. Therefore, $\text{recv_win_open}_p(t, r)$ holds for any real time t that is mapped to a clock-time strictly greater than $R_p(r) - 1$ (and strictly less than the beginning of the computation phase).

AXIOM 4.29. $\text{sched}_p(r) + P(r) \leq T < \text{sched}_p(r + 1)$ implies $\text{tin}_p(T, q) = \epsilon(\{m \in \text{mess} \mid \exists t \in \mathbb{R}. \text{recv_win_open}_p(t, r) \text{ and } \text{recv}_p(q, m, t)\})$.

Axiom 4.29 restricts Axiom 4.11 insofar as the time-triggered message received must be received within the reception window.

Axioms 4.12 through 4.14 are generalized by Axioms 4.30 through 4.32, respectively, by parameterizing the schedule functions:

AXIOM 4.30. *Let $T = sched_p(r - 1) + P(r - 1)$. Then*

$$\begin{aligned} ttss_p(init_s)(sched_p(r)) = \\ \text{if } r = 0 \text{ then } init_s_p \\ \text{else } trans_p(ttss_p(init_s)(T), \lambda q. ttin_p(T, q)) . \end{aligned}$$

AXIOM 4.31. *$sched_p(r) \leq T \leq sched_p(r) + P(r)$ implies*

$$ttss_p(init_s)(T) = ttss_p(init_s)(sched_p(r)) .$$

AXIOM 4.32. *For all nodes l ,*

$$\begin{aligned} \forall q : C_q(gs(r)) \geq sched_l(r) \\ \text{and } \exists p : C_p(gs(r)) \geq sched_l(r) . \end{aligned}$$

Furthermore, an additional axiom constrains the effects of pipelining. Axiom 4.33 ensures that while a node is in its computation phase, its state is either the state it has before applying its state-transition function in that round or the updated state resulting from its application (in this model, the state is updated at some nondeterministic time during the computation phase, but the entire state is updated instantaneously).

AXIOM 4.33. *$sched_p(r) + P(r) \leq T < sched_p(r + 1)$ implies either $ttss_p(init_s)(T) = ttss_p(init_s)(sched_p(r))$ or $ttss_p(init_s)(T) = ttss_p(init_s)(sched_p(r + 1))$.*

The transition system is constructed according to these axioms.

2.3. Simulation Proof. To demonstrate an equivalence between the time-triggered and untimed models, a simulation relation is shown to hold between the transition systems constructed for each model. There are two possible simulations that can be demonstrated, a *real-time simulation* or a *clock-time simulation*. A real-time simulation is one such that for each round, there exists a real time where the state of each node in the time-triggered model is the same as the states of the nodes in the synchronous model. A clock-time simulation is one in such that for each round, there exists a clock-time where the state of each node in the time-triggered model is the same as the states of the nodes in the synchronous model. In a synchronized system, such as a time-triggered system, a clock-time simulation implies a real-time simulation. We therefore prove a clock-time simulation exists first, and then we use this to demonstrate the existence of a real-time simulation.

A primary characteristic of a time-triggered system – and one upon which the simulation proof depends – is that messages sent by nodes are received at the appropriate time. Our first obligation is to show the schedule constraints are sufficient to ensure that if p is a node that sends a message to node q at real time t , then

$$C_p(t) = sched_p(r) + D(r)$$

$$\text{implies } recv_win_open(sendtime_q(r) + d, r)$$

where $\delta_{nom} - e_l \leq d \leq \delta_{nom} + e_u$. By the definition of *recv_win_open* (Definition 4.1), this expands into two inequalities. The first inequality is proved in Lemma 4.35 and the second is proved in Lemma 4.36. First, we prove a small supporting result in Lemma 4.34.

LEMMA 4.34. *Let $C_p(t) = sched_p(r) + D(r)$, and let $\delta_{nom} - e_l \leq d \leq \delta_{nom} + e_u$.*

Then

$$\min(C_p(t + d), C_q(t + d)) \leq \max(sched_p(r), sched_q(r)) + P(r) .$$

PROOF. By Axiom 4.1,

$$(1) \quad C_p(t+d) - C_p(t) \leq \lceil (1+\rho) \cdot d \rceil ,$$

so

$$C_p(t+d) \leq \text{sched}_p(r) + D(r) + \lceil (1+\rho) \cdot d \rceil .$$

The result follows from the preceding inequality and by Axiom 4.21. \square

LEMMA 4.35. *Let $C_p(t) = \text{sched}_p(r) + D(r)$. Then $\text{sched}_q(r) + R(r) - 1 < C_q(t+d)$.*

PROOF. By the hypothesis, Axiom 4.16 and Lemma 4.34,

$$|C_q(t+d) - C_p(t+d)| \leq \Sigma(r) ,$$

which implies that

$$(2) \quad C_p(t+d) \leq C_q(t+d) + \Sigma(r) .$$

By Axiom 4.1,

$$(3) \quad \lfloor (1-\rho) \cdot d \rfloor + C_p(t) \leq C_p(t+d) .$$

Inequalities 2 and 3 imply

$$\lfloor (1-\rho) \cdot d \rfloor + C_p(t) \leq C_q(t+d) + \Sigma(r) ,$$

and since $C_p(t) = \text{sched}_p(r) + D(r)$ by assumption,

$$\lfloor (1-\rho) \cdot d \rfloor + \text{sched}_p(r) + D(r) \leq C_q(t+d) + \Sigma(r) .$$

By Axiom 4.25 and since

$$\lfloor (1-\rho) \cdot (\delta_{nom} - e_i) \rfloor \leq d ,$$

it follows that

$$(4) \quad \text{sched}_p(r) + R(r) + \Lambda(r) - 1 < C_q(t+d) .$$

The result follows from Equation 4 and Axiom 4.18. \square

LEMMA 4.36. *Let $C_p(t) = sched_p(r) + D(r)$. Then $C_q(t + d) < sched_q(r) + P(r)$.*

PROOF. By Axiom 4.16 and Lemma 4.34,

$$|C_q(t + d) - C_p(t + d)| \leq \Sigma(r) ,$$

which implies

$$(5) \quad C_q(t + d) \leq C_p(t + d) + \Sigma(r) .$$

From inequality 1 of Lemma 4.34 and our assumption that $C_p(t) = sched(r) + D(r)$,

$$(6) \quad C_p(t + d) \leq sched_p(r) + D(r) + \lceil (1 + \rho) \cdot d \rceil .$$

From Inequalities 5 and 6,

$$C_q(t + d) \leq sched_p(r) + D(r) + \Sigma(r) + \lceil (1 + \rho) \cdot d \rceil .$$

The result follows from the preceding inequality and Axiom 4.21. \square

Lemma 4.37 ensures p 's time-triggered state at $sched_p(r) + D(r)$, when p generates its message for round r , is equal to its untimed state in round r if r is not pipelined; otherwise, it is equal to either its untimed state in round r or $r - 1$.

LEMMA 4.37. *Suppose that for all rounds $j \leq r$,*

$$(7) \quad ttss_p(init_s)(sched_p(j)) = run_p(j, init_s) .$$

Let $s = ttss_p(init_s)(sched_p(r) + D(r))$. Then if $independent(r)$, then either $s = run_p(r, init_s)$ or $s = run_p(r - 1, init_s)$; otherwise, $s = run_p(r, init_s)$.

PROOF. We consider the cases of whether $independent(r)$ holds.

- (1) If $independent(r)$ is false, then by Axiom 4.23, $D(r) \geq 0$. By Axiom 4.21, $D(r) < P(r)$, and so by Axiom 4.31, $s = ttss_p(init_s)(sched_p(r))$. Thus, by Equation 7, $s = run_p(r, init_s)$.

- (2) If $independent(r)$ holds, and $D(r) \geq 0$, then the result holds by the above argument. Otherwise, by Axiom 4.24,

$$D(r) + sched_p(r) \geq P(r - 1) + sched_p(r - 1) .$$

The result follows immediately from Axiom 4.33.

□

Lemma 4.38 shows that if each node's time-triggered state is the same as its untimed state, then in the time-triggered model, the messages a node p has received when it begins its computation phase in round r are the messages sent to p by the other nodes in round r , generated from their time-triggered states at the beginning of that round.

LEMMA 4.38. *Suppose that for all processes $q \in in_nbrs_p$ and all rounds $j \leq r$,*

$$(8) \quad ttss_q(init_s)(sched_q(j)) = run_q(j, init_s) .$$

Then

$$(9) \quad ttin_p(sched_p(r) + P(r), q) = msg_q(ttss_q(init_s)(sched_q(r)), p) .$$

PROOF. By Axiom 4.29, $ttin_p(sched_p(r) + P(r), q)$ is the message p receives from q in the communication phase of round r , if a message is in fact received. First we show a message is received (Item 1), and then we show that Equation 9 holds, so the message sent is the message received (Item 2).

- (1) Axioms 4.26 and 4.27 ensure that q sends p a message at clock-time $sched_q(r) + D(r)$. Axiom 4.17 ensures this message is received by p at real time $sendtime_q(r) + d$, where $\delta_{nom} - e_l \leq d \leq \delta_{nom} + e_u$. Lemmas 4.35 and 4.36 ensure that the reception window is open at this time.

(2) We must show that the message generated by q from its state at this clock-time is the same as the message generated by q at $sched_q(r)$. Consider the cases of whether r is independent or not.

- If r is not independent, then by Axiom 4.23, the communication offset is nonnegative. By Axiom 4.31, q 's state is invariant during its communication phase, so q 's state at clock-time $sched_q(r) + D(r)$ is the same as its state at clock-time $sched_q(r)$. Thus, q 's message is generated from its state, $ttss_q(init_s)(sched_q(r))$, and the result follows immediately.
- If r is independent, then by Lemma 4.37, q 's time-triggered state at $sched_q(r) + D(r)$ is equal to either its untimed state in round $r - 1$ or round r . In the first case, the message generated is the same as the one generated by q 's untimed state in round r by Axiom 4.22 which, by assumption (Equation 8), is the same message as that generated by q 's time-triggered state at $sched_q(r)$. The second case holds by the same reasoning as when r is not independent, described above.

□

Lemma 4.39 demonstrates that for node p , when its clock-time is $sched_p(r)$, its time-triggered state is the same as its untimed state in round r .

LEMMA 4.39 (Clock-Time Simulation). $ttss_p(init_s)(sched_p(r)) = run_p(r, init_s)$.

PROOF. By induction on the rounds.

Base Case: : In the base case, $r = 0$, and both the time-triggered system and the untimed system are in their initial states, $init_s$.

Induction Step: : The induction hypothesis is

$$ttss_p(init_s)(sched_p(r)) = run_p(r, init_s),$$

and we show that

$$ttss_p(\text{init}_s)(\text{sched}_p(r+1)) = \text{run}_p(r+1, \text{init}_s)(r+1) .$$

Unfolding $ttss_p$ and run_p gives us

$$\begin{aligned} & \text{trans}_p(ttss_p(\text{init}_s)(T), \lambda q. \text{tin}_p(T, q)) \\ &= \text{trans}_p(\text{run}_p(r, \text{init}_s), \lambda q. \text{msg}_q(\text{run}_q(r, \text{init}_s), p)) \end{aligned}$$

where $T = \text{sched}_p(r-1) + P(r-1)$. Substituting the induction hypothesis gives us

$$\begin{aligned} & \text{trans}_p(\text{run}_p(r, \text{init}_s), \lambda q. \text{tin}_p(T, q)) \\ &= \text{trans}_p(\text{run}_p(r, \text{init}_s), \lambda q. \text{msg}_q(\text{run}_q(r, \text{init}_s), p)) . \end{aligned}$$

Thus, we must show that

$$\text{tin}_p(T, q) = \text{msg}_q(\text{run}_q(r, \text{init}_s), p)$$

which holds immediately from the induction hypothesis and Lemma 4.38. □

THEOREM 4.40 (Real-Time Simulation). $ttss_p(\text{init}_s)(C_p(\text{gs}(r))) = \text{run}_p(r, \text{init}_s)$.

PROOF. First, we know that for all nodes p , $\text{pre_comp_phase}_p(\text{gs}(r), r)$. This follows from the clock synchronization axiom, Axiom 4.16, and the communication and computation offset constraints, Axioms 4.20 and 4.21.

Second, by Axiom 4.31 and Lemma 4.39, $ttss_p(\text{init}_s)(C_p(t)) = \text{run}_p(r, \text{init}_s)$ for any real time t such that $\text{pre_comp_phase}_p(\text{gs}(r), r)$.

The theorem follows immediately from these two results. □

3. Schedule Verification

To verify the schedules for time-triggered protocol implementations, we show its schedule satisfies the six schedule constraints, Axioms 4.19 through 4.21 and Axioms 4.23 through 4.25.

The system assumptions and the axiomatization of the transition system must also be satisfied to ensure that a time-triggered protocol implements its synchronous specification. We assume the system parameters are fixed, and that the system assumptions hold for the physical system. The axioms of the transition system are used to provide a semantics to the formalism and are not “satisfied” by the physical implementation.

This verification is carried out in SAL using its infinite-state bounded model checker. Because the languages of PVS and SAL are similar, the schedule constraints have nearly identical formulations in the respective languages. In particular, PVS and SAL share the same type theory. By ensuring the variables and constants specified in SAL have the same type declarations as the corresponding ones in PVS, we implicitly rely on the PVS typechecker to ensure their type-correctness.

The technique is demonstrated by verifying the schedules in the following three Sections. The schedules verified are taken from the VHDL coded by Wilfredo Torres-Pomales and Mahyar Malekpour of the NASA Langley Research Center, the implementors of the latest prototype [11]. The VHDL is being used to generate a FPGA-based implementation of SPIDER. The schedules were generated using Matlab[®] according to the analysis of the timing requirements [11]. These verifications provide an independent check that the by-hand analysis and the scripts generating the VHDL schedules are error-free. Some low-quality bugs (i.e., “typos”) were found in the generated schedules. More importantly, the verification provides a formal mapping from the synchronous specification of these protocols [27] to the time-triggered implementation.

3.1. Distributed Diagnosis Protocol Schedule Verification. The SPIDER Distributed Diagnosis Protocol ensures nodes maintain a consistent assignment of the faultiness of the other nodes. Nodes may individually accuse one another of being

faulty. During the diagnosis protocol, if enough nodes accuse a node, the accusations are promoted to an agreed-upon *conviction*. When a node has been convicted, the other nonfaulty nodes ignore the convicted node until it proves itself to be nonfaulty.² The protocol is described in detail by Torres-Pomales et. al. [11], and a formal verification of an early version of the protocol is described by Geser and Miner [57].

The verification of this protocol’s schedule is straightforward. The protocol has four rounds. The schedule offsets D , P , and R do not vary from round to round. We verify the protocol with respect to the maximum possible skew for the duration of the protocol. Furthermore, none of the rounds are pipelined, and there are no event-triggered actions.

3.1.1. *Type and Constant Declarations.* The type and constant declarations are straightforward in SAL. All system constants are interpreted to be concrete values taken from the system parameters for the targeted prototype. The SAL specification of the declarations are given in Figure 4. The schedule constraints require taking the floor and ceiling of the minimum and maximum communication delay, respectively; we do this by hand and set them equal to constants.

3.1.2. *Variables.* In the axiomatization of time-triggered systems described in Section 2.1, the following are functions, the domains of which are the set of rounds: $sched$, D , P , R , Λ , Σ , $independent$, and R . In the SAL model, we replace these functions with state variables that range over their respective ranges (e.g., a state variable replaces D that ranges over the integers). Then, in the state machine model described, these state variables are updated in each state according to which round is current in that state. This gives us a “state machine perspective” of each function’s behavior.

²The mechanism for doing involves executing a reintegration protocol; please see Section 5.2.

```

REALTIME      : TYPE = REAL;
CLOCKTIME     : TYPE = INTEGER;
OFFSET        : TYPE = {T: CLOCKTIME | T >= 0};
RND           : TYPE = NATURAL;
rho           : REALTIME = 1/10000;
d_nom         : {t: REALTIME | t >= 0} = 5;
ERROR         : TYPE = {t: REALTIME | t >= 0
                       AND t < d_nom};

e_l           : ERROR = 5/10000;
e_u           : ERROR = 5/10000;
% floor((1 - rho) * (d_nom - e_l))
fl_d_min      : CLOCKTIME = 4;
% ceiling((1 + rho) * (d_nom + e_u))
cd_d_max      : CLOCKTIME = 6;

```

FIGURE 4. Type and Constant Declarations

The state variables Σ and Λ are special cases. The other state variables are schedule constraints chosen by the implementer. However, the values of these variables are determined by the behavior of the system and the time that has elapsed.

3.1.3. *Schedule Constraint Specification.* The schedule constraints stated in Section 2.1 are stated in SAL as shown in Figure 5. Some of these constraints compare the schedule between successive rounds (e.g., Axiom 4.19). Because we have transcribed the functions over the rounds to variables that are updated in the state machine at each round, these relations may take as arguments the values of these variables in a round and compare them to the values in the next round (e.g., `constraint1` takes `pre_sched` and `sched` as arguments, denoting the values for $sched(r-1)$ and $sched(r)$, respectively). The SPIDER Distributed Diagnosis Protocol is completely time-triggered; therefore, for all rounds, the schedule skew Λ is zero. We therefore omit it from the constraints and the state-machine model described below.

3.1.4. *Specifying a Round-Based Schedule.* We create a state machine representation of how the schedule constraints evolve through the rounds of execution as shown

```

constraint1(P: OFFSET, pre_sched: CLOCKTIME,
            sched: CLOCKTIME): BOOLEAN =
  0 < P AND P < sched - pre_sched;

constraint2(D: CLOCKTIME, S: OFFSET): BOOLEAN =
  D >= S - fl_d_min;

constraint3(P: OFFSET, D: CLOCKTIME, S: OFFSET): BOOLEAN =
  P > D + S + cd_d_max;

constraint4(r: RND, D: CLOCKTIME): BOOLEAN =
  (NOT independent?(r)) => D >= 0;

constraint5(pre_P: OFFSET, D: CLOCKTIME, pre_sched: CLOCKTIME,
            sched: CLOCKTIME): BOOLEAN =
  D >= pre_P - sched + pre_sched;

constraint6(D: CLOCKTIME, R: CLOCKTIME, S: OFFSET): BOOLEAN =
  R - 1 <= D + fl_d_min - S;

```

FIGURE 5. SAL Specification of the Generalized System Assumptions

in Figure 6. The state machine is specified with a single module. Because the module does not need to communicate with other modules via shared variables, all the state variables are declared to be local. In addition to schedule variables, for each constraint, a boolean variable is declared. The value of the variable is determined by whether its associated schedule constraint is satisfied in the present round. The state machine includes a counter r that records the current round in the synchronous abstraction of the protocol. In each initial state, this counter is set to 0. Each transition from a state in round r is to a state in round $r + 1$. In general, we check the schedule constraints for the next-state values of the variables. For constraints that compare the values between rounds, the current-state variable values and the next-state variable values are compared in the constraint. Because there are no previous

state assignments in round 0, those state variables associated with constraints that compare values between rounds are declared to be true upon initialization.

Not every state variable needs to be updated in each transition. If a state variable is not reassigned in a guarded transition, its value remains the same in the next state. In the schedule verified for the SPIDER Distributed Diagnosis Protocol, the offsets do not change through the execution of the protocol. Thus, in the guarded transition shown in Figure 6, only the `rnd` and `sched` variables are updated, and the values of the other variables remain the same.

3.1.5. *Verification.* The property stating that in all reachable states, each constraint holds can then be specified by the following state invariant (the \mathbf{G} operator is a temporal logic operator denoting that the property is global; i.e., it holds in all reachable states [100]).

The property is verified by executing SAL’s infinite-state bounded model checker. The lemma `constraints` is verified when $k = 2$.

```
sal-inf-bmc -i -d 2 spider_diag_sched.sal constraints
```

The SAL language is typed, which can lead to deadlocked state machines. For example, a variable `x` of type `[0..2]` can be assigned the values 0, 1, and 2. If a transition in a state machine assigns `x` the value 3, for example, the state machine deadlocks. In a deadlocked state machine, state invariants hold vacuously. Therefore, it is a good idea to state and prove a “poor man’s liveness condition.” In an infinite-state system in SAL, only state invariants may be stated and proved. Therefore, to check for deadlock, one can express a state invariant that should be false in the model and

```

SYSTEM: MODULE =
BEGIN
LOCAL
  r                : RND,
  sched, D         : CLOCKTIME,
  P, S, R          : OFFSET,
  ind              : BOOLEAN,
  c1, c2, c3, c4, c5, c6 : BOOLEAN
INITIALIZATION
  r                = 0;
  sched            = 2;
  D                = 1;
  P                = 13;
  S                = 4;
  R                = 2;

  c1               = TRUE;
  c2               = constraint2(D, S);
  c3               = constraint3(P, D, S);
  c4               = constraint4(r, D);
  c5               = TRUE;
  c6               = constraint6(D, R, S) % no "pre" here
TRANSITION
[
  TRUE
  -->
  r' = IF r = 3 THEN 0 ELSE r + 1 ENDIF;
  sched' = sched+14;
  c1' = constraint1(P', sched, sched');
  c2' = constraint2(D', S');
  c3' = constraint3(P', D', S');
  c4' = constraint4(r', D');
  c5' = constraint5(P, D', sched, sched');
  c6' = constraint6(D', R', S')
]
END;

```

FIGURE 6. A Round-Based State Machine

attempt to prove it by k -induction. This is only a positive test for deadlock, however. If the invariant proves, then a deadlock exists. If the invariant fails to prove,

```
constraints: LEMMA SYSTEM |-
  G(c1 AND c2 AND c3 AND c4 AND c5 AND c6);
```

a deadlock may nevertheless exist; it may be the case that k is not sufficiently large for the deadlock to be discovered. Furthermore, state invariants taken together may produce a deadlocked system (often times, numerous invariants are proved about a single system).

In any event, proving the model is deadlock-free is relatively straightforward in this case because the state machine is constructed by a single SAL module and there is only one invariant to prove. The SPIDER Distributed Diagnosis Protocol is a four-round protocol. In the possible final states, the round counter $r = 3$ (assuming the initial state models a round, and the round counter is initialized to 0). Therefore, the following invariant should be false:

```
liveness: LEMMA SYSTEM |- G(r <= 2);
```

An attempted proof for it should fail, as does the following (for $k = 4$):

```
sal-inf-bmc -i -d 4 spider_diag_sched.sal liveness
```

This ensures the system is not deadlocked through the last round of the protocol.

3.2. Clock Synchronization Protocol Schedule Verification. The SPIDER Clock Synchronization Protocol ensures the local clocks of the nodes remained synchronized within a small skew and within a linear envelope of real time [11]. It is executed periodically to resynchronize the clocks due to drift. The protocol is designed to be fault-tolerant to ensure correct behavior in the presence of faulty nodes and clocks.

The verification of the protocol is described by Miner et. al. [27]. Therein, the protocol is abstracted synchronously. That this abstraction is justified is not altogether obvious. The protocol both adjusts the local clocks of the nodes (which in turn adjusts the skew of these clocks), and as well, it contains event-triggered actions. Initial synchronization messages are sent at a prescribed clock-time. Receivers actively monitor for these messages, and when they receive a sufficient number of them, they broadcast a message in return. The broadcast of these echo messages is time-triggered.

Much of the SAL model for the SPIDER Clock Synchronization Protocol is the same as for the SPIDER Distributed Diagnosis Protocol described above. The main differences are that the skew changes between rounds and that we must now model the event-triggered behavior of the protocol. The event-triggered behavior may cause Λ to be nonzero, and is therefore included as an argument to the schedule constraints specified in SAL. Furthermore, in the second through fourth rounds of the protocol, the behavior of the nodes is event-triggered. The event is the reception of a certain number of messages during a node's reception window. A node begins the next round a constant clock-time duration from this occurrence.

We therefore cannot assume that the nodes begin rounds two through four at the same clock-time. Therefore, the `sched` variable is parameterized by the nodes, and is declared to be an array with possibly different clock-time readings for each node. It is of type `SCHEDS`:

```
SCHEDS: TYPE = ARRAY NODES OF CLOCKTIME;
```

where the type `NODES` is a finite set of indices of nodes. In event-triggered rounds, the schedules are updated nondeterministically; for example, in one round, the following update may occur:

```
scheds' IN {s: SCHEDULES | scheds_update(s, scheds,
                                         S', R, P, 70)};
```

The relation `scheds_update` takes as arguments an updated schedule array `s`, the previous schedule array `scheds`, the updated clock skew, the reception window and computation phase offsets, and another nonnegative constant `C`. In the protocol, a sufficient number of messages may be received at any time within a node's reception window, so the updated schedule time may be a constant offset from any clocktime therein. Furthermore, the difference between the triggering events for each node is bounded by the maximum skew (in the succeeding round). The relation is as follows and its effect for the schedule update of a single node is illustrated in Figure 7.

```
scheds_update(s: SCHEDULES, scheds: SCHEDULES, S: OFFSET,
              R: OFFSET, P: OFFSET, C: OFFSET): BOOLEAN =
  FORALL (i: NODES):
    s[i] >= scheds[i] + R + C
  AND s[i] < scheds[i] + P + C
  AND FORALL (j: NODES): s[i] - s[j] <= S;
```

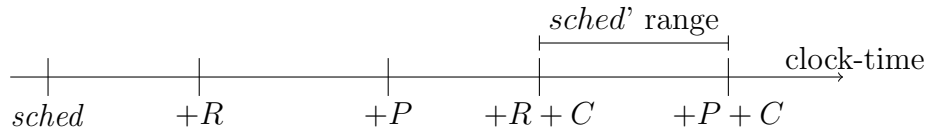


FIGURE 7. Clock Synchronization Event-Triggered Schedule Update

3.3. Schedule Optimization. This methodology provides a “push-button” means for ensuring that modifications to a schedule still satisfy the schedule constraints. First, we explore the use of this technology to verify a schedule that pipelines the rounds of two protocols. In particular, we pipeline the SPIDER Interactive Consistency Protocol with the SPIDER Distributed Diagnosis Protocol [11]. The latter protocol was briefly described in Section 3, the former protocol is a fault-tolerant protocol to reliably pass data from a single source BIU to the other BIUs (and ultimately, from a source PE to the other PEs). This is the protocol that provides the bus capabilities of the ROBUS. The other protocols, including the distributed diagnosis and clock synchronization protocols, are periodically executed to maintain the health of the ROBUS. The time required to execute these protocols can reduce the throughput of interactive consistency messages.

The SPIDER Interactive Consistency Protocol is a two-round protocol, whereas the SPIDER Distributed Diagnosis Protocol is a four-round protocol. The pipelined schedule will therefore contain six rounds. We alternate rounds between the two protocols, so the interactive consistency protocol is executed twice for every one time the distributed diagnosis protocol is executed. Therefore, in the SAL model, the communication and computation of each round is independent of the preceding round:

```
independent?(r: RND): BOOLEAN = r > 0;
```

We can then assign a negative value to the communication offset, so that messages are sent before the computations of the previous round are complete.

Not only can optimizations such as pipelining be explored, but we can also do parameterized verification. Suppose, for example, that the time at which the pipelined protocols are scheduled to execute is event-triggered, so we cannot exactly fix the

maximum clock skew (i.e., we cannot fix how much time has elapsed since the clocks were last synchronized). Thus, we may initialize the maximum skew variable non-deterministically:

```
S IN {T: OFFSET | T <= 3};
```

Both system parameters and schedule offsets can be similarly parameterized.

4. Summary

The first part of this chapter reviews and extends a formal theory of time-triggered protocols. In particular, the theory describes a set of system assumptions and a set of schedule constraints such that if they are met by a time-triggered protocol, the protocol faithfully implements its synchronous specification. The second half of this chapter describes a methodology to prove that an implementation satisfies the schedule constraints. The methodology is demonstrated by verifying schedules of the SPIDER Distributed Diagnosis Protocol and the SPIDER Clock Synchronization Protocol taken directly from a current VHDL implementation. We also describe how this method can be used to optimize the scheduling of protocols.

Most protocols can be specified and verified in the synchronous model, and then their time-triggered implementation can be proved to meet the synchronous specification. Nevertheless, some protocols execute when the system is unsynchronized, and cannot be modeled by the time-triggered model presented here. A strategy for specifying and verifying unsynchronized protocols is developed in the following chapter.

Partially-Synchronous Protocol Verification

Time-triggered systems – and their protocols – are designed to satisfy the synchrony assumption the majority of the time during which the system executes. Nevertheless, sometimes they do not satisfy this assumption. For instance, in a distributed system in which nodes are independently-clocked, synchronization must be achieved upon power-up by executing a *startup protocol*. A startup protocol takes the nodes from an unsynchronized state to a synchronized one. A similar protocol may be executed if the system must be reset.

The probability of the system failing to satisfy the synchrony assumption increases if it must execute in the presence of faults. For example, during a massive failure in the system, the individual nodes undergo a self-test and then execute a restart protocol to regain consistent state with one another. The synchrony assumption may not be satisfied in this case. Another example is when a single node suffers a *transient fault* causing it to lose its volatile state but suffer no permanent damage. In this case, the node executes a *reintegration protocol*.

Although the specification and verification strategy described in Chapters 3 and 4 is suitable for time-triggered protocols, it is not suitable for the protocols that execute in the boundary-cases described above. A verification of these sort of protocols requires their real-time behavior to be modeled. One very recent strategy for these sort of real-time verifications are by infinite-state bounded model checking as implemented in SAL [101]. We develop the use of this technique for easy verification of real-time fault-tolerant protocols.

1. Synchronizing Timeout Automata (STA)

The following definitions build upon the timeout automata semantics described in Section 8, Chapter 2. We call this the *Synchronizing Timeout Automata* (STA) model. The STA model provides a succinct specification and simple semantics for systems that synchronize both with respect to events (e.g., message passing) and with respect to time. For example, the train-gate-controller (TGC), is a standard example of such a real-time system [90]. In a timeout automata model, the TGC is modeled as the asynchronous composition of timeout automata in which synchronous communication is modeled by the sequential application of transitions [53]. We provide a simpler timeout automata model.

As noted, timeout automata were motivated by Dutertre and Sorea’s desire to specify timed systems amenable to k -induction in SAL. Proofs by k -induction have a complexity that is exponential with respect to k , by solving the equivalent boolean satisfaction problem. The initial timeout automata models of the SPIDER Reintegration Protocol required k -induction at infeasible depths: proofs by k -induction for $k > 4$ were often infeasible for even a small number of modeled nodes. By allowing both synchronous and asynchronous composition, the depth required for proofs by k -induction can be reduced since in a synchronous composition, multiple transitions may be applied simultaneously.

We use the train-gate-controller (TGC) to illustrate this. Dutertre and Sorea prove a simple safety property using k -induction, for $k = 14$, when the TGC is modeled with the (asynchronous) timeout automata semantics described in Section 8, Chapter 2 [53]. In Section 1.4, we prove the same property with $k = 9$ in a synchronous timeout automata model described below. When the optimization in Sec 1.5 is also applied, the property is provable for $k = 5$. The optimization allows more

complex systems to be verified via k -induction without having to strengthen the invariant, and it is necessary to complete the verification of the reintegration protocol described in Section 2.

1.1. Syntax.

DEFINITION 5.1 (STA Syntax). A *synchronizing timeout automaton STA* is a tuple $\langle V, M, I, E \rangle$, where

- V is a nonempty finite set of state variables. fV is the set of all possible total *assignment functions* that assign values from the respective sets over which the variables range to these variables. These functions are called *states*. We use variables f, g, h , and i to denote states.
- $M \subseteq 2^V$ is a nonempty set of subsets of state variables that cover V (i.e., for each $v \in V$, there exists $m \in M$ such that $v \in m$). For $m \in M$, the set $fV_m = \{f \upharpoonright m \mid f \in fV\}$ is the set of states restricted to variables in m . An element $f_m \in fV_m$ is the m *timeout component* or m -*component* of state f .
- I is a set of initial states and associated timeouts. A *timeout* is associated with each $m \in M$. A timeout ranges over the set of nonnegative reals, denoted by $\mathbb{R}^{0 \leq}$. The set of all possible *timeout vectors* is $TO = \{\alpha \mid \alpha : M \rightarrow \mathbb{R}^{0 \leq}\}$ (we use lowercase Greek letters to denote timeout vector variables). The relation $I \subseteq fV \times TO$ relates initial states to initial timeout vectors.
- E is a set of edges for each timeout component. For $m \in M$, let $TO_m = \{\alpha \upharpoonright m \mid \alpha : M \rightarrow \mathbb{R}^{0 \leq}\}$ be the set of possible timeout vectors restricted to subsets of m (we use subscripted lowercase Greek letters to denote restricted timeout vector variables). An element $\alpha_m \in TO_m$ is an m -*timeout vector*.

For each $m \in M$, $E_m \subseteq fV_m \times TO_m \times fV_m \times TO_m$ is an *edge relation*. E_m relates a current m -component and m -timeout vector to an updated m -component and m -timeout vector. An edge $\langle f_m, \alpha_m, g_m, \beta_m \rangle$ is called an *m -edge* or an *edge for m* .

REMARK 5.1 (Timeouts and Timeout Components). A timeout component represents a portion of the state that updates synchronously. The notion of a timeout component is independent of a single state machine in a composition. For example, if two composed machines are synchronized and share a time-triggered schedule (see Chapter 4), the state variables of the two machines are in a shared timeout component. A completely synchronous distributed system can be represented by letting M be a singleton set containing V .

REMARK 5.2 (Edges). In general, if an edge updates a timeout nondeterministically, it is updated to some value over a continuous interval on the nonnegative reals.

1.2. Semantics. We require that a STA satisfy the following property to provide a semantics. It ensures that if edges for different timeout components are simultaneously applied, they agree on how to update shared variables.

DEFINITION 5.2 (Synchronous Update Property). For all $m, n \in M$ where $m \neq n$ and $m \cap n \neq \emptyset$, if there exist edges $E_m(f_m, \alpha_m, g_m, \beta_m)$ and $E_n(f_n, \alpha_n, h_n, \gamma_n)$, then $g_m \upharpoonright n = h_n \upharpoonright m$, and $\beta_m \upharpoonright n = \gamma_n \upharpoonright m$.

DEFINITION 5.3 (STA Semantics). Let $STA = \langle V, M, I, E \rangle$ be a timeout automaton that satisfies the Synchronous Update Property. Its semantics is an unlabeled transition system \mathcal{S}_{STA} . The context distinguishes whether we speak of the states in fV or the constructed states of the transition system. A *state* of \mathcal{S}_{STA} is a tuple

$\langle f, \alpha, t \rangle$ consisting of a state $f \in fV$, a timeout vector $\alpha \in TO$, and a clock, $t \in \mathbb{R}^{0\leq}$.

The tuple $\langle f, \alpha, t \rangle$ is an initial state of \mathcal{S}_{STA} if and only if $\langle f, \alpha \rangle \in I$ and $t = 0$.

Let $\langle f, \alpha, t \rangle$ and $\langle g, \beta, t' \rangle$ be states. There is a *time progress transition* $\langle f, \alpha, t \rangle \xrightarrow{t} \langle g, \beta, t' \rangle$ if and only if $t < \min(\alpha)$, $t' = \min(\beta)$, $g = f$, and $\beta = \alpha$.

The following terminology is useful for describing *discrete transitions*. In the state $\langle f, \alpha, t \rangle$, $E_m(h_m, \gamma_m, i_m, \delta_m)$ is an *enabled edge* if and only if $h_m = f \upharpoonright m$, $\gamma_m = \alpha \upharpoonright m$, and $\alpha(m) = t$. An *m-component* is an *enabled timeout component* in $\langle f, \alpha, t \rangle$ if and only if there exists an *m-edge* enabled in that state. If $\langle f, \alpha, t \rangle$ and $\langle g, \beta, t' \rangle$ are states, then $E_m(h_m, \gamma_m, i_m, \delta_m)$ is *applied* in the discrete transition $\langle f, \alpha, t \rangle \xrightarrow{E} \langle g, \beta, t' \rangle$ if and only if it is an enabled edge in $\langle f, \alpha, t \rangle$, $i_m = g \upharpoonright m$, and $\delta_m = \beta \upharpoonright m$.

The discrete transition $\langle f, \alpha, t \rangle \xrightarrow{E} \langle g, \beta, t' \rangle$ holds if and only if for every $m \in M$ such that m is an enabled *m-component* in $\langle f, \alpha, t \rangle$, there exists some *m-edge* that is applied, and $t' = t$.

REMARK 5.3 (Minimum Timeouts). An edge $E_m(f_m, \alpha_m, g_m, \beta_m)$ will never be applied if $\alpha_m(m) \neq \min(\alpha_m)$.

REMARK 5.4 (Nonzeroness and NonZenoness). Additional properties are required for executability. The Nonzero Property ensures that timeouts are never updated to values in the past, and at least one timeout is updated to some time in the future. This prevents infinite discrete state transitions with no time progress. For all edges $E_m(f_m, \alpha_m, g_m, \beta_m)$, $\min(\beta_m) \geq \min(\alpha_m)$, and there exists $n \in M$ such that $\beta_m(n) > \min(\alpha_m)$. Note that this does not prevent an edge from updating a timeout to some time sooner than its current value.

The *nonZeno Property* [102] ensures that an infinite number of transitions are not enabled within a finite interval of time. This property must be satisfied for a specification to be implementable.

1.3. Composition. Two STA are composed by taking the union of their state variables, timeout components, and edges. The initial states of the composition is defined as the set of states satisfying the initial conditions of each automata.

DEFINITION 5.4 (Composition). Let $STA^1 = \langle V^1, M^1, I^1, E^1 \rangle$ and $STA^2 = \langle V^2, M^2, I^2, E^2 \rangle$. Their composition, denoted $STA^1 \parallel STA^2$, is the STA $\langle V^1 \cup V^2, M^1 \cup M^2, I, E^1 \cup E^2 \rangle$, where $\langle f, \alpha \rangle \in I$ if and only if $\langle f \upharpoonright V^1, \alpha \upharpoonright M^1 \rangle \in I^1$, and $\langle f \upharpoonright V^2, \alpha \upharpoonright M^2 \rangle \in I^2$.

REMARK 5.5 (Compositional Specifications). The specification of a STA is independent of the notion of composed state machines. Because timeout components include state variables from communicating state machines, in practice, state machines are not specified separately as STAs and then composed.

1.4. Example: STA Model of the Train-Gate-Controller. The train-gate-controller (TGC) models the interaction of a train, a gate, and a gate controller at a railroad crossing. Assume there is one train on a circular track that may repeatedly approach the crossing. Initially, the train is out of the crossing, and the gate is up. The train signals its approach to the controller, and after a delay of exactly one unit of time, the controller signals the gate to lower. Once the gate has been signaled, it takes no more than 1 unit of time to lower. It takes from three to five units of time from the time the train signals its approach until it enters the crossing. Furthermore, it must exit the crossing within 5 units of time from when it signals its approach. When the train signals its exit to the controller within one unit of time of receiving

this signal, the controller signals the gate to raise. The gate takes at least one and no more than two units of time from when it is signaled to raise until it is completely up. As soon as the train has exited, it may approach the crossing again.

This behavior is modeled as composed timed automata by Alur [90] and is shown in Figure 1. The train, gate, and controller state machines each begin in states t_0 , g_0 , and c_0 , respectively. Their clock variables are x , y , and z , respectively, and they are assumed to be synchronous. Clock constraints at the vertices denote the time by which the state must be left. Clock constraints at the edges constrain when the edge is enabled, and clocks may also be reset when a transition is taken. A transition is nondeterministically taken at some time satisfying the constraints. Edges are labeled. If edges from distinct state machines share a label, transitions on these edges must be synchronized. For example, when the train state machine is in state t_0 and the controller state machine is in state c_0 , they must transition to states t_1 and c_1 , respectively, simultaneously.

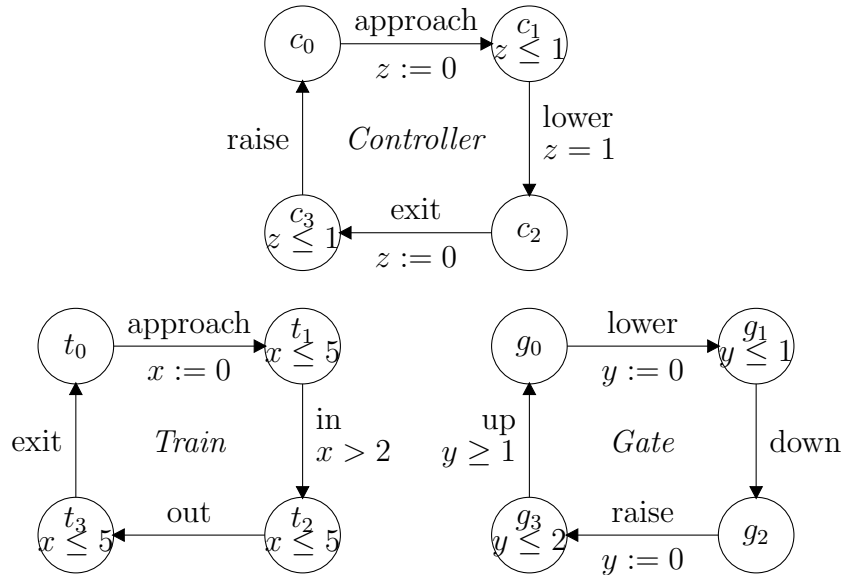


FIGURE 1. The Train-Gate-Controller

1.4.1. *STA Model of the TGC.* Following Definition 5.1, the TGC is modeled as a STA $\langle V, M, I, E \rangle$, informally described as follows. A presentation in the language of SAL can be found on-line [13].

- V : There are five main state variables. The variable s_t ranges over the state labels for the train (t_0, t_1 , etc.); variables s_g and s_c similarly range over the labels for the gate and controller, respectively. The variable msg_t ranges over $\{approach, exit, null\}$, the messages the train sends to the controller (the *null* message denotes the lack of a message being sent). Likewise, the variable msg_g ranges over $\{lower, raise, null\}$, the messages the controller sends the gate. All of the messages that are not sent between machines are irrelevant in the STA model).
- M : The set M contains three elements, m_t , m_c , and m_g . Each of these sets contain the state-label variables and message variables for a machine, and if that machine outputs messages to another one, it contains the state-label variables for that machine, too. Thus, $m_t = \{s_t, msg_t, s_c\}$, $m_g = \{s_g\}$, and $m_c = \{s_c, msg_c, s_g\}$.
- I : The state-label variables are initially set to t_0 , g_0 , and c_0 , respectively. The messages variables are initially set to *null*. Initially, timeouts may have any value, but note that some initial states lead to deadlock (e.g., if m_g initially has the strictly least-valued timeout).
- E : For each timeout component, the edges update the state labels and timeouts in that component according to the constraints described. Consider, for example, an edge for the m_t -component in which the train and controller synchronize on the *approach* message. For such an edge $E_{m_t}(f_{m_t}, \alpha_{m_t}, g_{m_t}, \beta_{m_t})$, $f_{m_t}(s_t) = t_0$ and $f_{m_t}(s_c) = c_0$ (msg_t may have any value). In the updated state, $g_{m_t}(s_t) = t_1$, $g_{m_t}(s_c) = c_1$, and $g_{m_t}(msg_t) = approach$. The updated

timeouts are those associated with m_t and m_c ; they are nondeterministically updated to satisfy the constraints $\alpha_{m_t}(m_t) + 2 < \beta_{m_t}(m_t) \leq \alpha_{m_t}(m_t) + 5$ and $\beta_{m_t}(m_c) = \alpha_{m_t}(m_c) + 1$, respectively.

REMARK 5.6 (Timeout Vs. Timed Automata). Unlike in the timed automata formalization, clocks are not reset. Timeouts continue to increase indefinitely, but they are required to satisfy the constraints given the current time. For example, if t is the current time, upon entering state t_1 , the timeout for m_{tc} is nondeterministically updated to some value greater than $t + 2$ and less than or equal to $t + 5$.

1.5. Clockless STA Semantics. The clock in an STA can be conservatively removed. By applying this optimization, we are able to reduce the depth at which k -induction must be applied to prove safety properties about timeout automata. For example, for the TGC, a basic safety property of the model is that whenever the train is in the railroad crossing, the gate is down. In the original timeout automaton model, this is proved in SAL by k -induction at depth 14 [53]. After applying the optimization described here, this depth is reduced to $k = 5$. The STA model of the TGC applying the optimization is available on-line [13]. This optimization helped to complete the verification of the reintegration protocol.

In a timeout automaton, the purpose of the clock is to record the least-valued timeouts of the automata. That is, the clock is either equal to the least-valued timeout(s), or it is equal to the least-valued timeout(s) in the next state. However, this information can be obtained from the timeouts themselves; the clock variable is unnecessary. Removing the clock variable reduces the state space. Each time the timeouts are updated so that no timeout is equal to the current clock time, a transition is taken in which only the clock variable is updated. In the worst case, this can double the value of k required to prove a safety property via k -induction.

Finally, removing the time transitions simplifies the semantics insofar as only one kind of transition need be considered. In most formalisms for specifying real-time systems, the semantics included both time and state transitions.

DEFINITION 5.5 (Clockless STA Semantics). Let $STA = \langle V, M, I, E \rangle$ be a timeout automaton that satisfies the Synchronous Update Property. Its semantics is an unlabeled transition system $\mathcal{S}_{STA}^{\neg cl}$. A state of \mathcal{S}_{STA} is a pair $\langle f, \alpha \rangle$ consisting of a state $f \in fV$ and a timeout vector $\alpha \in TO$. A state $\langle f, \alpha \rangle$ is an initial state of $\mathcal{S}_{STA}^{\neg cl}$ if and only if $\langle f, \alpha \rangle \in I$.

In the state $\langle f, \alpha \rangle$, the edge $E_m(h_m, \gamma_m, i_m, \delta_m)$ is an *enabled edge* if and only if $h_m = f \upharpoonright m$, $\gamma_m = \alpha \upharpoonright m$, and $\alpha(m) = \min(\alpha)$. An *m-component* is an *enabled timeout component* in $\langle f, \alpha \rangle$ if and only if there exists an *m-edge* enabled in the state. Furthermore, if $\langle f, \alpha \rangle$ and $\langle g, \beta \rangle$ are states, the edge $E_m(h_m, \gamma_m, i_m, \delta_m)$ is *applied* in the transition $\langle f, \alpha \rangle \rightarrow \langle g, \beta \rangle$ if and only if it is an enabled edge in $\langle f, \alpha \rangle$, $i_m = g \upharpoonright m$, and $\delta_m = \beta \upharpoonright m$.

The transition $\langle f, \alpha \rangle \rightarrow \langle g, \beta \rangle$ holds if and only if for every $m \in M$ such that m is an enabled *m-component* in $\langle f, \alpha \rangle$, there exists some *m-edge* that is applied in the transition.

The following proposition asserts that the same states are reachable under both semantics, modulo the clock variable values.

PROPOSITION 5.1 (Clockless Simulation). *Fix a STA $\langle V, M, I, E \rangle$. Let its semantics from Definition 5.3 be the transition system \mathcal{S}_{STA} , and let its clockless semantics be the transition system $\mathcal{S}_{STA}^{\neg cl}$. If $\langle f, \alpha, t \rangle$ is a reachable state of \mathcal{S}_{STA}^{cl} , then $\langle f, \alpha \rangle$ is a reachable state of $\mathcal{S}_{STA}^{\neg cl}$, and if $\langle f, \alpha \rangle$ is a reachable state of $\mathcal{S}_{STA}^{\neg cl}$, then there exists a t such that $\langle f, \alpha, t \rangle$ is a reachable state of \mathcal{S}_{STA}^{cl} .*

PROOF. We prove both conjuncts by induction. To prove the first conjunct, assume $\langle f, \alpha, t \rangle$ is an initial state. Then $\langle f, \alpha \rangle$ is an initial state of \mathcal{S}_{STA}^{-cl} , by definition. For the induction step, suppose that if $\langle f, \alpha, t \rangle$ is a reachable state in \mathcal{S}_{STA}^{cl} , then $\langle f, \alpha \rangle$ is a reachable state in \mathcal{S}_{STA}^{-cl} . In \mathcal{S}_{STA}^{cl} , there are two kinds of transitions, time progress transitions and discrete transitions. In a time progress transition, only the clock variable updates. By definition, a discrete transition $\langle f, \alpha, t \rangle \xrightarrow{E} \langle g, \beta, t' \rangle$ holds in \mathcal{S}_{STA}^{cl} exactly when the transition $\langle f, \alpha \rangle \rightarrow \langle g, \beta \rangle$ holds in \mathcal{S}_{STA}^{-cl} .

To prove the other conjunct, suppose that $\langle f, \alpha \rangle$ is an initial state of \mathcal{S}_{STA}^{-cl} . Then $\langle f, \alpha, 0 \rangle$ is an initial state of \mathcal{S}_{STA}^{cl} . For the induction step, suppose that if $\langle f, \alpha \rangle$ is a reachable state in \mathcal{S}_{STA}^{-cl} , then there exists a t such that $\langle f, \alpha, t \rangle$ is a reachable state in \mathcal{S}_{STA}^{cl} . Suppose there is a transition $\langle f, \alpha \rangle \rightarrow \langle g, \beta \rangle$ in \mathcal{S}_{STA}^{-cl} . Then by definition, there is a discrete transition $\langle f, \alpha, t \rangle \xrightarrow{E} \langle g, \beta, t' \rangle$ in \mathcal{S}_{STA}^{cl} , such that $t' = t$.

□

COROLLARY 5.2. *If P is a safety property that does not mention the clock variable, then P holds in \mathcal{S}_{STA}^{cl} if and only if P holds in \mathcal{S}_{STA}^{-cl} .*

EXAMPLE 5.1 (TGC with Clockless STA Semantics). Removing the clock is straightforward. In SAL, this amounts to removing the module that specifies the global clock, as described in Section 1.4. The specifications of the train, gate and controller must then be modified: rather than comparing timeouts against the global clock to determine whether an edge is enabled, timeouts are explicitly compared with one another. The SAL specification is available on-line [13].

REMARK 5.7 (k -Induction in Clockless Semantics). By removing the global clock, we are able to decrease the depth of k -induction to prove the safety property described in Section 1.4 from 14 under the original timeout automata semantics to $k = 9$ in the STA semantics to $k = 5$ in the clockless STA semantics.

1.6. Timeout Automata Specification and Verification in SAL. Because SAL is a general-purpose specification and verification environment, it does not automatically generate the semantics of an STA from its syntax. Therefore, we describe a *shallow embedding* of the STA semantics in the language of SAL.

Consider the TGC example described in Section 1.4. Modules are specified for the train, gate, and controller. Each of these contains output variables for their state labels and outgoing messages, and if a module receives messages from another, those are specified as input variables. Each module also has an output timeout variable against which the other modules can compare their own timeouts. Because edges may simultaneously update variables from multiple modules (e.g., in a synchronized transition between the train and controller), the train, gate, and controller are synchronously composed. In each of the guards for the transitions in the train, gate, and controller modules is a condition that the relevant timeout is equal to the current time. If two machines synchronize on a message, the sender’s timeout is a guard for both the sender and receiver. For example, the train sends the controller an *exit* message. When this message is sent is guarded by the train’s timeout in the train module. In the gate module, its action that depends on receiving the message is guarded by the train’s timeout and the reception of the message.

2. Case-Study: The SPIDER Reintegration Protocol

Distributed fault-tolerant systems such as SPIDER and TTA are designed to withstand both *permanent faults* and *transient faults*, two means by which to classify *fault persistence* [8, 40]. A permanent fault is caused by physical disruptions that damage the system and affect the system until it is repaired or replaced off-line. A transient fault affecting a node may cause it to lose its volatile state but suffer no permanent damage. This can be caused by high-intensity radiation, for example. Although a

transiently-faulty node may be fault-free, its state no longer is coordinated with that of the *operational clique*, the set of fault-free nodes with coordinated states allowing them to provide the requested services of the system. The operational clique and the set of non-faulty nodes are not necessarily equivalent: for example, a reintegrator is a non-faulty node not in the operational clique. This distinction can be subtle and in fact, a misunderstanding of it was partially responsible for a subtle error in the previous design of another SPIDER protocol [16]. Nodes in the operational clique are called *operational nodes*.

If too many nodes become uncoordinated with the operational clique, the system degrades and becomes more susceptible to new faults. Too many simultaneous faults will lead the system to violate its *maximum fault assumption* (MFA), the maximum kind and number of faults the system is designed to withstand yet maintain correct operation. If the MFA is violated, no guarantees can be made about the system's behavior.

The reliability requirements for these busses coupled with the potential for a high number of transient faults in the environments in which they operate have led to the development of reintegration mechanisms for these systems. For a transiently-faulty node to regain correct state, it may execute a reintegration protocol. In a synchronized fault-tolerant distributed system, the reintegrating node (called the *reintegrator*) executes the protocol to resynchronize its local clock with those of the nodes in the operational clique. As well, it may need to regain diagnostic data consistent with the operational clique. A node's diagnostic data are its view of which other nodes are faulty (messages from faulty nodes should be ignored). Other state may also be regained via the protocol; for example, if the system supports dynamic scheduling, this needs to be obtained, too.

We present the first formal verification of a reintegration protocol. Rushby and Pfeifer respectively describe the formal verification of TTA, one of the most mature and formally-verified busses in development, and therein state that the formal analysis of reintegration remains important future work [25, 26]. The work presented here should be extensible to other fault-tolerant systems that employ reintegration protocols, especially given that this verification is architecture-independent, as elaborated in Section 3.

The protocol described here abstracts the reintegration protocol being designed for the latest SPIDER prototype [11]. The most significant abstraction is that we model only the portion of the protocol in which the reintegrator resynchronizes its local clock with the clocks of the nodes in the clique. We omit that portion of the protocol in which the reintegrator regains diagnostic data consistent with the operational clique. This portion of the protocol is a slight modification of the SPIDER Distributed Diagnosis Protocol. The main difference is that the reintegrator simply listens but does not broadcast messages as in the full distributed diagnosis protocol. The Distributed Diagnosis Protocol has been formally verified in PVS [27].

From a pragmatic standpoint, resynchronization during reintegration is the most complex portion of the protocol and stands to benefit the most from formal analysis. Once reintegration is achieved, the remainder of the protocol can be modeled synchronously, substantially easing its analysis.

Other minor simplifications include, for example, not modeling timers signaling massive failure (e.g., where there is no clique with which to reintegrate) that triggers the reintegrator to stop executing the reintegration protocol and begin executing a reset protocol. The protocol, as it is described in the remainder of this section, is fully modeled and verified using SAL.

During the reintegration protocol, the reintegrator monitors its communication links for *echo messages* (or simply *echos*) sent by the other nodes. Echos are messages sent by nodes during the SPIDER Clock Synchronization Protocol. The clock synchronization protocol must be executed periodically by all operational nodes because clock drift is inevitable, even in operational nodes. The period beginning at the conclusion of one execution of the synchronization protocol lasting until its next execution is called a *resynchronization frame* or simply a *frame*.

We verify the correctness of the reintegration protocol with respect to a single reintegrating node. During the reintegration protocol, the reintegrator sends no messages. If multiple reintegrators are executing the protocol, they receive no messages from each other, assuming they are non-faulty. Although a reintegrating node may be non-faulty, it will be considered faulty by other nodes simultaneously reintegrating. In particular, a reintegrating node will diagnose another as suffering a *fail-silent* fault, since it receives no messages from it. This issue is discussed in more detail in Section 3.

The reintegrator is designed to tolerate the full range of faulty behaviors, including Byzantine faults [39], manifested as arbitrary behavior to respective observers. However, because the reintegration protocol is not a distributed protocol (i.e., only a single node executes it), the only fault manifestations detectable by the reintegrator are *benign faults*, detectable in point-to-point communication [95].

Finally, note that the ability of the reintegrator to reintegrate successfully with the operational clique depends on the behavior of the nodes in the operational clique as well. In particular, the reintegrator executes the reintegration protocol after suffering a transient fault and resetting. During this period, the operational nodes have likely determined the reintegrator to be faulty. So long as the operational nodes believe the reintegrator to be faulty, they will ignore it, even if it resynchronizes and regains

correct local state. Thus, to allow for reintegration, the operational nodes must periodically purge their diagnostic data to allow nodes a chance to reintegrate. In the current SPIDER prototype, the non-faulty nodes purge their diagnostic data at the end of each resynchronization frame. This allows a node that has suffered a fault in one resynchronization frame to successfully reintegrate in another.

2.1. System Assumptions. Before describing the behavior of the protocol, a preliminary understanding of the system assumptions is required. The system assumptions are invariants assumed to hold to demonstrate the correctness of the protocol. These properties are stated in terms of *accusations* made by the reintegrator. The reintegrator accuses a node when it believes the communication from the node is inappropriate (e.g., the reintegrator does not receive an echo message when one is expected or it receives one unexpectedly).

The first property constrains the behavior of the operational nodes during each resynchronization frame. It is guaranteed by the correctness of the clock resynchronization protocol [27] and the high-level scheduling of the protocols. It is illustrated in Figure 2.

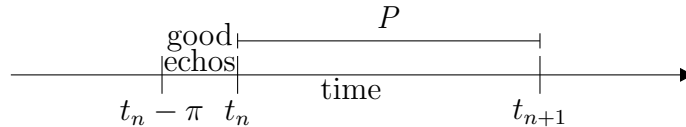


FIGURE 2. The Frame Property

DEFINITION 5.6 (Frame Property). Let $\{t_n\}_0^\infty$ be a sequence of nonnegative reals (denoting real time) assumed to have the following properties: for all $n \in \mathbb{N}$, $t_{n+1} > t_n$ and $t_{n+1} - t_n = P$. The constant P is called the *frame length*, and for each n , the interval $[t_n, t_{n+1})$, closed on the left and open on the right, is the n th *frame*. P is constrained as follows: let l be the number of faulty nodes not accused by the

reintegrator during the preliminary diagnosis and frame synchronization modes (to be described shortly). Then $P > l\pi + 2\pi$. The constant $\pi \in \mathbb{R}^{0<}$ and is called the *skew constant*. The reintegrator receives exactly one echo message from each operational node during each open interval $(t_n - \pi, t_n)$ (in this model, we include communication error in the skew) and no more than one echo message in each frame.

The next property ensures that enough of the monitored nodes that have not been accused are non-faulty for the protocol to work.

DEFINITION 5.7 (Majority Property). Of the nodes that have not been accused by the reintegrator during the entire protocol, the majority are operational.

2.2. Protocol Description. The reintegration protocol is comprised of three modes of operation: *preliminary diagnosis*, *frame synchronization*, and *synchronization capture*. These modes are executed sequentially in as shown in Figure 3. We itemize the global and local state variables of the modes, and then we describe the behavior of the protocol during each mode.

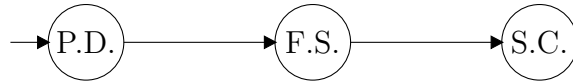


FIGURE 3. State Machine Model of the Protocol Mode Control

2.2.1. *State Description.* The following state variables of the reintegrator determine the state of the reintegrator during the execution of the protocol. In the following, let i range over the indices of the nodes the reintegrator monitors.

- *accs* is an array of boolean values such that for each node i , $accs[i]$ is true if the reintegrator *accuses* node i of being faulty and it is false otherwise. The reintegrator ignores echos from nodes it has accused.
- *clock* is the current time of the reintegrator's local clock.

- fs_finish ranges over the nonnegative reals and is a timer variable used in the frame synchronization mode.
- $mode$ records the current mode being executed. It ranges over the set $\{prelim_diag, frame_synch, synch_capture\}$, denoting the three modes, respectively.
- pd_finish ranges over the nonnegative reals and denotes the time at which the preliminary diagnosis mode completes.
- $seen$ is an array of natural numbers such that for each node i , $seen[i]$ records the number of times a message has been received from i .

The following state variables are initialized at the beginning of the reintegration protocol.

```

for each  $i$ ,  $accs[i] := false$ ;
 $mode := prelim\_diag$ ;
for each  $i$ ,  $seen[i] := 0$ ;

```

2.2.2. Protocol Behavior. When the reintegrator begins executing the reintegration protocol, it has no diagnostic data to use in deciding which nodes are faulty and which are not. Trusting too many faulty nodes may lower the probability that it will successfully reintegrate with the operational clique. The purpose of preliminary diagnosis is to acquire preliminary diagnostic data to attempt to recognize faulty nodes early in the protocol. This is achieved by monitoring echo messages for the duration $P + \pi$. The reintegrator expects to receive at least one and no more than two echo messages from i .

In the following pseudo code, a **when** statement is a guarded action. The guard $echo(i)$ is true precisely when the reintegrator receives an echo message from node i .

```

pd_finish := clock + P +  $\pi$ ;
while clock < pd_finish do {
  for each i, when echo(i) do {
    if (seen[i] < 2 and not accs[i])
      then seen[i] := seen[i] + 1
      else accs[i] := true;
  };
};
for each i, if seen[i] = 0 then accs[i];
mode := trans;

```

The purpose of the frame synchronization mode is to determine a time such that all operational nodes have already issued an echo message in some frame and before any operational node issues an echo in the next frame. An interval satisfying this property is referred to as a *frame gap*. This provides the reintegrator with a coarse-grained synchronization with the operational clique: a reintegrator is assumed to separate echo messages (from operational nodes) arriving in different resynchronization frames.

The mode relies on echo messages from operational nodes being separated by no more than π units of time. Therefore, the mode begins monitoring for echos, and it exits when π units of time have elapsed such that no echo is observed from a node that has not yet been accused. If an echo is observed within that time from a node that has not be accused, then the timer is reset.

Acquiring this course-grained level of synchronization is a precondition for the actual resynchronization that occurs in the next mode.

```

for each  $i$ ,  $seen[i] := 0$ ;
 $fs\_finish := clock$ ;
while  $clock - fs\_finish < \pi$  do {
  for each  $i$ , when  $echo(i)$  do {
    if ( $seen[i] = 0$  and not  $accs[i]$ )
    then {
       $fs\_finish := clock$ ;
       $seen[i] := seen[i] + 1$ ;
    };
    else  $accs[i] := true$ ;
  };
};
 $mode := synch\_capture$ ;

```

The synchronization capture mode is the final mode of the reintegration protocol. Its purpose is to allow the reintegrator to determine a time during which some operational node issues an echo message. It does so by synchronizing when it has received echos from at least half of the nodes it has not accused (or has not already seen in this mode). To ensure that it is synchronizing with an operational node, the Majority Property (Definition 5.7) must hold. If so, the reintegrator will have become resynchronized with the operational clique, within the accepted skew, π .

Let $trusted$ be the total number of nodes the reintegrator has not accused: $trusted = |\{i \mid \text{not } accs[i]\}|$. Let $seen_cnt$ be the number of nodes seen (that have not been accused in previous frames): $seen_cnt := |\{i \mid seen[i] > 0\}|$.

```

for each  $i$ ,  $seen[i] := 0$ ;
while  $seen\_cnt \leq trusted/2$  do {
  for each  $i$ , when  $echo(i)$  do {
    if ( $seen[i] = 0$  and not  $accs[i]$ )
    then  $seen[i] := seen[i] + 1$ ;
  };
};
 $clock := 0$ ;

```

2.3. Modeling. We now describe the modeling of the reintegration protocol as a STA with clockless semantics. We describe the model in the language of SAL. The

shallow embedding of the semantics of the reintegration protocol’s STA model in SAL is similar to the TGC example described in Section 1.6. The full model can be found on-line [13].

2.3.1. *Timeouts.* The model contains the following timeout variables: `reint_to`, which is primarily associated with the reintegrator; `frame_to`, which is primarily associated with the operational nodes; and each faulty node has its own timeout. The timeouts for the operational and faulty nodes essentially exist for modeling purposes. In modeling the reintegrator’s execution of the protocol, we require a model of the entire system’s behavior. A naïve model would fix the behavior of the monitored nodes over multiple resynchronization frames *a priori*. However, the state space required to do so makes this infeasible. Rather, we model the behavior of the monitored nodes one frame at a time. The frame in which the reintegrator is presently in is modeled, and if the reintegrator passes into another frame by updating `reint_to`, then the monitored nodes simultaneously change to the same frame (of course, the reintegrator is modeled to have no knowledge of which frame in which it actually resides).

This model admits a few simplifications. The behavior of the reintegration protocol depends on that of the observed nodes, but not vice versa. Thus, the model can be constructed so that `reint_to` is always the minimum of the other timeouts, which is proved by *k*-induction. This ensures the issuing of echo messages are always future events observable by the timeout model of the reintegrator.

2.3.2. *Monitored Nodes.* To verify the correctness of the protocol, we must model both the reintegrator and the monitored nodes. In the model, we distinguish between nodes in the operational clique and faulty nodes (as discussed in Section 2, non-faulty nodes not in the operational clique are considered faulty by the reintegrator, and their

behaviors are subsumed by the modeled behavior of the faulty nodes). We describe the model of the two kinds of nodes in turn.

To model the operational nodes, we begin by defining a module that keeps track of the resynchronization frames, as presented in Figure 4. The timeout `frame_to` serves as an abstract global clock shared by the synchronized operational nodes. The timeout keeps track of the values of t_n marking the end of a frame, as described in Section 2.1. There is a single transition, updating the timeout `frame_to` on transitions when the timeout `reint_to` has been updated so that its value is in the next resynchronization frame. This can be determined by comparing the next state's value of `reint_to` (denoted by `reint_to'`) to the end of the current resynchronization frame. The variable `new_frame` is a boolean value that is true if and only if the transition just taken was one in which the frame has been updated.

```

P_update: MODULE =
    :
    TRANSITION
    [
        frame_to <= reint_to'
        -->
        frame_to' = frame_to + P;
        new_frame' = TRUE
    []
    ELSE -->
        new_frame' = FALSE
    ]

```

FIGURE 4. Synchronization Frame Module

The operational nodes themselves are specified by an `op_node` module, parameterized by the indices of operational nodes, presented in Figure 5. The timeout for an operational node is `frame_to`. Whenever the frame updates, it nondeterministically updates its echo variable, `op_echo` (ranging over the nonnegative reals), to a new

value satisfying the Frame Property (Definition 5.6). This is a conservative model insofar as an operational node may update its echo to any time satisfying the constraints, so the difference between the echos it issues in adjoining frames may be up to $P + \pi$. In reality, the clock of an operational node would not drift so dramatically.

To ensure the correctness of the model, when the reintegrator moves from one frame to the next, its timeout `reint_to'` must never be updated so far into the future that it is beyond when operational nodes issue echos in the next frame. An invariant is proved about the model that demonstrates that this does not occur.

```

op_node[i: OP_IDS]: MODULE =
    :
    TRANSITION
    [
        frame_to <= reint_to'
        -->
        op_echo' IN {t: TIME |      frame_to' > t
                        AND t > frame_to' - pi}
    []
    ELSE -->
    ]

```

FIGURE 5. Operational Node Module

Finally, the instances of `op_node` are synchronously composed, and this composition is synchronously composed with the `P_update` module as shown in Fig 6.

```

op_nodes: MODULE =
    WITH OUTPUT op_echos: OP_ECHOS
    (|| (i: OP_IDS): RENAME op_echo TO op_echos[i]
        IN op_node[i]);

clique: MODULE = op_nodes || P_update;

```

FIGURE 6. Operational Clique Module

Faulty nodes are also specified by a module parameterized by the indices of nodes that may exhibit faulty behavior. The model is slightly more complicated so that all possible faulty behaviors are modeled, yet k -induction proofs are feasible over the transition system. In a naïve model of the entire system, the reintegrator would make a transition whenever it receives an echo from a node it is actively monitoring. This would amount to updating its timeout to be equal to the timeout of the first echo it receives and updating its state accordingly. It would then reset its timeout to the next echo and so on. In this model, the reintegrator’s transitions are event-triggered; they depend on echo events. However, because a faulty node may issue multiple echos before being ignored by the reintegrator, this model can quickly lead the reintegrator to make a large number of transitions for even a relatively small number of faulty nodes. For k -induction to succeed, a more sophisticated model is required.

A preferable model is one in which the reintegrator’s transitions are essentially time-triggered. This amounts to the reintegrator updating its timeout irrespective of the states and timeouts of the monitored nodes. Ideally, a time-triggered model of the reintegrator would make a small number of time-triggered transitions at regular intervals and update its state based on all of the echos received during the intervals rather than updating its state upon receiving each echo.

Care must be taken to make a time-triggered model conservative. Because timeouts record when future events occur, when the reintegrator makes a state transition, it can only “observe” those echos that come after its current timeout and no later than the time at which it sets its next timeout. For example, in a naïve model, suppose the reintegrator were to update its state in a time-triggered fashion as illustrated in Figure 7. Suppose `reint_to` denotes the reintegrator’s current timeout, which is also the least of all timeouts. Suppose that for some monitored node, it issues an echo message at time *echo*. The reintegrator observes this echo message, and updates its

timeout to `reint_to'`. Once the current time reaches `echo`, however, that node could issue another echo message at `echo'`, which will go undetected by the model of the reintegrator.

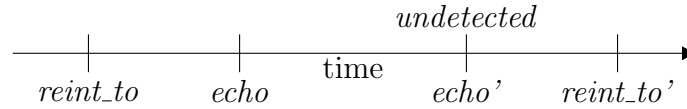


FIGURE 7. The Reintegrator TA Misses Echo Messages

Therefore, we allow the reintegrator to behave in a time-triggered fashion (in part), but faulty nodes are able to issue multiple echo messages in a single transition. The model of a faulty node contains a state variable `bad_echo`, as shown in Figure 8, that is an array of echos (nonnegative reals). The array has three indices. This is because the greatest number of echos that must be observed from any node in a mode is three before the node is accused. The echo in the first index also serves as the timeout for a faulty node, and the remaining echos in the array are guaranteed to always be greater than the timeout by the `ascending?` predicate. If the reintegrator updates its timeout in a time-triggered manner, there is the possibility it will observe all three echos during the update.

Nevertheless, there is no upper bound on how large any of the values in the array may be. If the echos are too far ahead of the reintegrator's timeout, they will be beyond the time to which it updates its timeout in a time-triggered transition and will never be observed. Thus, the module also models faulty nodes that are fail-silent. As well, note that the behavior of a faulty node so modeled may also be indistinguishable from that of an operational one. A faulty node may issue echos such that the first echo in the array consistently satisfies the Frame Property (Definition 5.7), and the other echos are beyond the observation window of the reintegrator.

```

bad_node[i: BAD_IDS]: MODULE =
    :
    TRANSITION
    [
        bad_echo[1] <= reint_to'
        -->
        bad_echo' IN {be: BAD_ECHO_ARRAY
                      | ascending?(be, reint_to')}
    ]
    ELSE -->
]

```

FIGURE 8. Faulty Node Module

The precondition for the transition to update the echos is similar to that described for the frame synchronization module described in Section 2.3.2. Here, if the next state's value of `reint_to` ever surpasses the first echo from a faulty node, all of the faulty nodes echos are updated. Thus, all of the faulty node's echos are always observed by the reintegrator (i.e., the values of each echo is greater than `reint_to`). This is also provable in the model by k -induction.

Each of the three modes of the reintegration protocol is specified as a separate module. Additionally, another module handles mode control, which we describe first. Each mode has a binary control signal to determine whether it is active. Only one mode may be active at any time. The module specified in Figure 9 ensures the correct flow of control through the modes. It is synchronously composed with the modules specifying the modes themselves.

We specify mode control for a number of reasons. Making mode control explicit simplifies the analysis of counterexamples generated by SAL when attempting to verify properties of the formal model; knowing in which mode the counterexample occurs simplifies the search for the error. The mode control is part of the protocol as it was designed. Mode exit points mark locations in the execution of the protocol

```

modes: MODULE =
    :
    TRANSITION
    [
        mode = pd_mode
        -->
        mode' = IF pd_cntrl=active
                THEN mode ELSE fs_mode
            ENDIF
    []
        mode = fs_mode
        -->
        mode' = IF fs_cntrl=active
                THEN mode ELSE sc_mode
            ENDIF
    []
    ELSE -->
    ]

```

FIGURE 9. Mode Control Module

where certain invariants are supposed to be reached. An invariant guaranteed upon the completion of a mode serves as an assumption in demonstrating the succeeding mode behaves correctly. Demonstrating that each mode guarantees the appropriate invariants is sufficient to demonstrate the entire protocol behaves correctly. Thus, modes serve as both a conceptual and formal decomposition to model and verify the protocol. Because the module is synchronously composed with the mode modules, it does not affect the trajectory-length required for k -induction proofs.

The three modes of the reintegration protocol are specified by separate SAL modules. Because of the distinct way in which operational and faulty nodes are modeled, it is simpler to specify distinct *accs* and *seen* variables for each kind. For example, in the SAL model, the reintegrator contains variables *op_accs* and *bad_accs* to record accusations. Nevertheless, care is taken to ensure that the reintegrator has no *a priori* knowledge about which nodes are in fact operational and which are faulty.

In addition, in proving invariants, we found it simpler to specify separate *seen* variables for each mode rather than resetting the *seen* variable at the conclusion of each mode.

In the preliminary diagnosis mode, there are two principle transitions, as shown in Figure 10. The first transition models the behavior during the mode, and the second models exiting the mode. Our model of the reintegrator during the preliminary diagnosis mode is essentially time-triggered. The variable `pd_finish` marks the time at which the mode exits. The effect of a transition is to move the reintegrator's timeout from the beginning of frame n to the beginning of frame $n+1$. As it does so, it records the echos observed in that frame and updates its state variables recording how many echos are seen from each node and whether they should be accused, respectively. When the reintegrator's timeout is updated to the beginning of the next frame, the `P_update` module simultaneously updates `frame_to` to prepare the reintegrator to observe the echos in the next frame. If the mode should exit before the termination of the frame, the reintegrator's timeout is updated to the time at which the mode should end, and only those echos in the interium are recorded by the reintegrator.

The purpose of the frame synchronization mode is to allow the reintegrator to discover some time during which no echos have been observed for π units of time (from nodes it does not know to be faulty). Thus, as shown in Figure 11, we define the relation `none_in_pi?` that determines whether this holds. If the relation is not satisfied, the reintegrator's timeout is updated to the greatest echo not known to be from a faulty node within π units of time of the reintegrator's current timeout within the current frame. If no such echo exists within the current frame, `reint_to` is updated to the beginning of the next frame, allowing the operational echos to be updated (see Section 2.3.2). When the relation does hold, the reintegrator's timeout is simply updated to be π units of time greater than its current value.

```

preliminary_diagnosis_mode: MODULE =
    :
TRANSITION
[
    mode' = pd_mode
  AND frame_to < pd_finish
  -->
    reint_to' = frame_to;
    :
[]
    mode' = pd_mode
  AND frame_to >= pd_finish
  -->
    pd_cntrl' = deactive;
    reint_to' = pd_finish;
    :
]

```

FIGURE 10. Preliminary Diagnosis Module

In the last mode, shown in Figure 12, we allow the reintegrator to behave in an event-triggered fashion. The reintegrator's timeout is updated from its current value to the time at which the soonest echo message occurs (that does not come from a node known to be faulty), or if no such echo exists in the current frame, it updates to the beginning of the next frame. The function `sc_seen_total` records how many echo messages have been seen so far. The mode exits when more than half of the nodes that have not been accused have been observed – that is, when

$$\begin{aligned}
 & \text{sc_seen_total}(\text{sc_op_seen}, \text{sc_bad_seen}) \\
 & > \text{not_accd}(\text{op_accs}, \text{bad_accs})/2.
 \end{aligned}$$

This also marks the termination of the reintegration protocol.

The three mode modules are composed asynchronously, in the `base_modes` module:

```

frame_synchronization_mode: MODULE =
    :
    TRANSITION
    [
        mode' = fs_mode
        AND NOT none_in_pi?(reint_to, op_echos, bad_echos,
                            fs_op_seen, fs_bad_seen,
                            op_accs, bad_accs)
        -->
        fs_cntrl' = active;
        reint_to' IN {t: TIME
                    | last_in_pi?(t, reint_to,
                                   op_echos, bad_echos,
                                   op_accs, bad_accs,
                                   fs_op_seen,
                                   fs_bad_seen,
                                   reint_to)};
    :
    []
        mode' = fs_mode
        AND none_in_pi?(reint_to, op_echos, bad_echos,
                        op_accs, bad_accs,
                        fs_op_seen, fs_bad_seen)
        -->
        fs_cntrl' = deactivate;
        reint_to' = reint_to + pi;
    :

```

FIGURE 11. Frame Synchronization Module

No two modes should be active simultaneously. This is enforced by ensuring that if one mode is active, the others are deadlocked. The reintegrator is then defined as the synchronous composition of the `base_modes` module and the `modes` module:

The entire system is the synchronous composition of the reintegrator module, the clique module, and the module of the composition of the faulty nodes:

```

synch_capture_mode: MODULE =
    :
    TRANSITION
    [
        mode' = sc_mode
        AND    sc_seen_total(sc_op_seen, sc_bad_seen)
              <= not_accd(op_accs, bad_accs)/2
        -->
        sc_cntrl' = active;
        reint_to' IN {t: TIME
                    | next?(t, reint_to,
                            op_echos, bad_echos,
                            op_accs, bad_accs,
                            sc_op_seen,
                            sc_bad_seen, frame_to)};
        :
    []
        mode' = sc_mode
        AND    sc_seen_total(sc_op_seen, sc_bad_seen)
              > not_accd(op_accs, bad_accs)/2
        -->
        sc_cntrl' = deactivate;
        :
    ]

```

FIGURE 12. Synchronization Capture Module

```

base_modes: MODULE =
    preliminary_diagnosis_mode
    []
    frame_synchronization_mode
    []
    synch_capture_mode;

```

FIGURE 13. The Composition of the Reintegrator's Modes

2.4. Verification. There are two main theorems to prove. First, we wish to show that the reintegrator accuses no operational nodes during the execution of the

```
reintegrator: MODULE = base_modes || modes;
```

FIGURE 14. Reintegrator Module

```
system: MODULE = reintegrator || clique || bad_nodes;
```

FIGURE 15. Full System Composition

reintegration protocol. Second, we wish to show that the reintegrator has successfully resynchronized with the operational nodes upon completion of the reintegration protocol.

THEOREM 5.3 (No Operational Accusations). *For all operational nodes i , $accs[i]$ does not hold during the reintegration protocol.*

THEOREM 5.4 (Synchronization Acquisition). *For all operational nodes i , $|clock - echo(i)| < \pi$ upon termination of the reintegration protocol.*

The proofs of these theorems via k -induction requires a number of supporting lemmas. Our strategy is to decompose the protocol verification into a verification of its constituent modes. Each mode should guarantee certain postconditions. The postconditions for a mode then serve as preconditions for succeeding modes. This strategy can be followed through the entire protocol making the proof of the above theorems straightforward.

This proof strategy is similar to the proof by abstraction technique used by Dutertre and Sorea [53, 54] and is a form of *disjunctive invariants*, an invariant composed from a set of disjuncts rather than the usual conjunctive form [88, 103]. Dutertre and Sorea manually construct the abstraction over the transition system to

be verified. Its construction appears to require a good of understanding of the protocol. While their abstraction technique is powerful, its construction is complicated; the mode abstraction used here is simply adopted from the protocol specification.

The proof of Thm 5.3 requires showing that no accusations are issued in any of the modes; accusations are not issued in the synchronization capture mode, so we need only be concerned with the first two modes. One challenge in doing so is that the reintegrator is unsynchronized with the operational nodes in these modes, so it may begin listening for echos at any time during a frame. In particular, it may begin listening for echos after some operational nodes have issued them and before others have done so. Thus, for example, in the preliminary diagnosis mode, we cannot state precisely how many echos messages the reintegrator should receive from operational node. Rather, the reintegrator should receive at least one and no more than two echos. Proving that this in fact happens requires some additional lemmas regarding the maximum and minimum length of time the mode is active, and the effects of the mode initializing at different points in a frame.

The proof of Thm 5.4 relies principally on two supporting lemmas, each of which provides preconditions for the mode. The first precondition is that no operational nodes have been accused (Thm 5.3). The second is that the time at which the synchronization capture mode initializes and the reintegrator begins listening for echos is such that either all operational nodes in that frame have already issued echo messages or no operational node in the frame has issued one; that is, the frame synchronization mode has successfully located a frame gap.

2.4.1. *Architectures Verified.* In the prototypical design of SPIDER, the reintegrator monitors no more than three nodes. The architecture of the SPIDER bus is a bipartite graph of six nodes (i.e., there are two disjoint sets of nodes, and any two nodes from distinct sets have interconnects and no two nodes from the same set have

interconnects) [104], and this architecture with six nodes is designed to tolerate up to two simultaneous value faults.

The protocol has been verified for up to four monitored nodes, where one node may be faulty, (without increasing the number of non-faulty nodes, a greater number of faulty nodes would violate the Definition 5.7, the Majority Property). The proofs took on the order of seconds (and occasionally minutes) to complete on a typical modern desktop CPU with a gigabyte of memory. Although we did not attempt it, it may be possible to verify these properties for models containing a greater number of monitored nodes if proofs are allowed to run on the order of hours or on a more powerful machine. Furthermore, strengthening the invariants would allow larger architectures to be verified.

Because of the way in which we have modeled the protocol, for most lemmas, the size of k required to prove a lemma is invariant to the number of monitored nodes modeled. The size of k is dependent upon the duration of a mode (i.e., for how many resynchronization frames it is active) rather than on how many echos are received in the mode. For the architectures verified, all lemmas are proved by k -induction for $k \leq 4$.

SAL has the capacity to assist the user in discovering required lemmas. It has an option such that when enabled, SAL will return a counterexample to a failed proof by k -induction. Because the model is infinite, the counterexample is often symbolic. It shows a k -trajectory over which the constraints of the infinitely-typed variables do not satisfy the induction step (rarely does the base case fail). The onus is on the user to interpret how the constraints lead to a counterexample.

Clique avoidance is the property that there exists exactly one operational clique in the system [7, 105]. If more than one clique exists, the nodes in one clique will consider the nodes in the other to be either faulty or recovering, and the members

of each clique disregard the nodes in the other. This decreases the survivability of the system, since each clique is smaller than it would be if all the nodes were in the same clique. Worse though is that multiple cliques may lead the processors connected to the bus architecture to lose agreement about the status of the bus. The bus interface unit serving as the interface between a processor and the other nodes in the bus architecture can only communicate within the clique in which it is a member. Consequently, multiple cliques can violate processor-level fault-tolerance requirements the bus is supposed to guarantee for the attached processors.

The SPIDER architecture does not have a protocol to guarantee clique avoidance, unlike, e.g., TTP/C [26, 105]. However, the architecture is designed with the intent that if during the course of its operation the MFA (Section 1) is not violated, clique avoidance is guaranteed. The analysis of the reintegration protocol supports this claim by demonstrating that a necessary condition for clique avoidance is met.

Suppose the MFA is not violated and the protocols executed by the non-faulty nodes during startup and normal operation guarantee clique avoidance. Then the only opportunity for clique avoidance to be violated is after multiple nodes suffer transient faults and attempt to find a clique with which to reintegrate. If we assume clique avoidance holds while a node begins to reintegrate, it has only one clique to observe. By Theorem 5.3, such a node will not accuse the nodes in the single clique during reintegration and will therefore reintegrate into it by executing the reintegration protocol.

The two assumptions to the above argument are essential. First, it is necessary to assume the MFA is not violated. If the architecture suffers a massive failure that triggers a bus restart, scenarios exist in which clique avoidance is violated, although these scenarios have a low probability [11]. Although the essential protocols that

execute during startup and normal operation have been formally verified individually [27], there does not yet exist a cohesive argument to demonstrate formally that clique avoidance is preserved.

3. Summary

We have described a formal proof of the correctness of the SPIDER Reintegration Protocol in the SAL tool using k -induction. We have described improvements to a novel formalism for real-time system. Finally, we have described a means by which event-triggered behavior can be modeled as time-triggered behavior. The essential means by which we achieved our results were by introducing synchrony into the formalism and by (conservatively) modeling event-triggered actions with time-triggered behavior.

Modeling the reintegration protocol revealed two distinctions between this protocol and the other fault-tolerant protocols designed for SPIDER and similar systems. First, although the ROBUS is designed to withstand Byzantine faults, these sort of faults do not warrant special consideration when reasoning about reintegration. A node that suffers a Byzantine fault can send arbitrary messages to other nodes. The difficulty in designing distributed protocols to tolerate Byzantine faults is that different nodes may receive different messages from the same node. The reintegration protocol is not a distributed protocol; only the reintegrator executes a reintegration protocol, so only the messages the reintegrator receives are relevant when reasoning about the correctness of the protocol.

Second, the topology of the system does not need to be modeled. The verification is with respect to a single node, the reintegrator. The topology can be abstracted away so that only the reception of messages by the reintegrator from the other nodes in the system is modeled. If a communication link does not exist that allows a node

to send messages to the reintegrator, then that node is simply ignored in the formal model.

The formal specification and verification of the reintegration protocol reveals no flaws in the protocol. Nevertheless, it is of value since no hand proofs existed to demonstrate its correctness. Furthermore, the protocol was significantly different from the other SPIDER protocols and many other fault-tolerant distributed protocols [29]. As well, the formal verification not only demonstrates the correctness of reintegration but it strongly suggests that clique avoidance is preserved.

The formal verification does reveal that a more general assumption can be used to prove correctness: we require only that $P > l\pi + 2\pi$, where P is the duration of a resynchronization frame, π is the skew, and l is the number of faulty nodes not accused by the reintegrator during the first two modes. In the originally-stated assumption, the requirement was that $P > m\pi + \pi$, where m is the total number of monitored nodes. This latter requirement implies the former since Definition 5.7, the Majority Property, ensures there is at least one operational node. In the worst case – i.e., if the reintegrator trusts as many faulty nodes as possible for the protocol to work – they are equivalent.

A difficulty with k -induction is that properties can be proved vacuously if the system is deadlocked. Checking for deadlocks in a infinite-state systems is a difficult problem. This is exacerbated by the fact that SAL’s language is typed, and violating typing constraints can cause deadlocks as well. The heuristic used to check for deadlocks, just as described in Chapter 4, is to specify properties known to be false and attempt to prove them by k -induction. This is only a positive test for deadlock; a counterexample does not imply the system is not deadlocked.

Proving safety properties of parameterized real-time systems models using infinite-state bounded model-checking is promising. This technique has been used in other

real-time domains too, with much success. Parameterized models of two physical layer communication protocols – 8N1 (used in UARTS) and the Biphase Mark Protocol – have been verified [19, 106]. The verification of the former physical layer protocol reveals a significant error in a published technical note, and the verification of the latter is orders of magnitude easier than previous approaches using mechanical theorem-proving.

CHAPTER 6

Conclusion

A methodology for the formal verification of time-triggered systems is developed in the preceding chapters. For these kinds of systems, a variety of timing models are used to specify their protocols, and the ability to reason within and between these timing models is essential for their verification.

1. Limitations

A complete formal verification methodology for time-triggered systems has not been provided. The development of safety-critical systems is recent and formal verification research in this domain is less mature than in others such as hardware. Many open problems in time-triggered system verification described by Rushby and Pfeifer have not been addressed herein including, for example, formal analyses of application-level fault-tolerance, and never-give-up strategies [25, 26].

The following are limitations specific to the work presented in each of the technical chapters (Chapter 3 through 5).

1.1. Chapter 3. A set of abstractions for the formal verification of synchronous fault-tolerant distributed protocols is presented. Like any set of abstractions, modeling a system using them requires the system to satisfy various constraints, notably, they require the abstracted protocols to implement synchronous behavior. Although the abstractions have proved useful in the specification and verification of the SPIDER protocols, it remains to be seen how they extend to similar systems. Preliminary

work by the NASA Langley SPIDER Group and Holger Pfeifer to extend the specifications presented by Miner et. al. [27] to the specification of TTA, which are founded upon these abstractions, has been promising.

1.2. Chapter 4. One limitation we find with the approach outlined to verify implemented protocol schedules is that the timing analysis developed in the formal theory does not correspond exactly with that developed by the SPIDER engineers who produced the protocol schedules [11]. For example, offsets for the communication phase, computation phase, reception window, etc. are offsets from the beginning of the current round. In the engineers' specifications of the SPIDER prototypes, these offsets are calculated with respect to the scheduled start of the entire protocol. Another difference is that the analyses carried out by the engineers uses *clock functions* from clock-time to real-time whereas we use inverse clock functions, which are functions from real-time to clock-time. There are other minor differences. The differences require an informal mapping from the variables and constants in the VHDL schedules developed by the engineers to those of the formal theory. This is a result, in part, of beginning with Rushby's existing theory and formalized specifications. Ideally, the formal theory is developed in conjunction with the engineers' analysis (or even replaces that analysis) to ensure a simple mapping.

1.3. Chapter 5. Beyond the standard limitation of bounded model-checking – that the propositional satisfaction problem is exponential in the size of the formula – *infinite-state* bounded model-checking depends on powerful combinations of theory-specific SMT solvers. At the time of this writing, this is an active area of research. In fact, the model reported in Chapter 5 provided many of the benchmarks for the first SMT competition [107]. The use of this technique depends on the development of more powerful and better integrated solvers.

2. Future Work

The following describes natural extensions of the work presented in this dissertation.

2.1. Emergent Properties. As described in Chapter 1, some of the most difficult yet important challenges facing the formal verification of time-triggered systems are the specification and verification of system-level “emergent properties” [25]. This work addresses this challenge indirectly. In particular, one of the difficult aspects in verifying emergent properties is that separate protocols tend to be specified and verified in different timing models (see Pfeifer [26]). A systematic means for abstracting a wide range of protocols to the synchronous model (Chapter 4) and a systematic means for specification in this model (Chapter 3) should ease this challenge.

2.2. Design Derivation. In general, the level of abstraction this dissertation deals with is the protocol level. These specifications are of the global system behavior and the properties the distributed protocols must satisfy. A gap exists between this level of abstraction and the implementation, in either hardware or software. One of the greatest differences is that a protocol specification describes the coordinated behavior of some subset of the node operations. A specification of a single node describes its sequential behavior over all the protocols in which it plays some role. Roughly, the two aspects of specification are orthogonal: a protocol specification “cuts” the system temporally, and the node specification “cuts” the system spatially. A protocol specification describes the global system over some time interval, whereas a node specification describes the entire behavior over time of a single node. There are additional differences: a protocol-level specification models the environment, which

may include the effects of faults, timing assumptions, and so on. A protocol specification is necessary to verify the protocols behave correctly, but it is from a node-level specification from which an implementation can be built.

There are two possibilities for creating a formal connection between a protocol-level specification and a node-level specification. The first is to use a verification technology such as mechanical theorem-proving or model checking. For example, Bevier and Young use the ACL2 theorem-prover to verify a hardware realization of the Oral Messages protocol [108]. The drawbacks of using mechanical theorem-proving is that it is time-intensive, requires substantial expertise, and formulations and proofs are not easily modified in the face of changes to the specification or realization. Furthermore, many of the refinements made to obtain the realization from the specification may be routine, but each instance of the refinement requires a separate proof of correctness. More preferable would be a form of *interactive synthesis* or *interactive compilation*. Results reported demonstrate techniques for deriving hardware specifications from high-level functional specifications, and it may be possible to extend these to the derivation of hardware realizations of time-triggered systems [18, 109–111]. Another approach from the synchronous languages perspective [112] for generating a TTA implementation from Lustre [113] specifications is described by Caspi et. al. [114].

2.3. Verification Libraries. The success of modern programming languages is due in part to the vast libraries available to the programmer. No large-scale programming effort would be attempted in a language without good library support. Not only does the existence of libraries reduce the development time, but it also reduces programming bugs. Libraries have often been carefully designed by experts in the language, and they have undergone extensive testing.

Despite current practice, these remarks hold for formal specification and verification projects, too. A main criticism of formal methods, and particularly mechanical

theorem-proving, is that it is very time-intensive. Significant effort is required to formulate and prove background results needed in a mechanical-theorem-prover. Basic mathematical results are proved repeatedly in verification endeavors.

One approach is to provide domain-centric libraries. A set of specifications and mechanical proofs have been developed for SPIDER in PVS [27, 95]. Libraries generalizing these results are being developed. They are being generalized for the specification and verification of time-triggered systems other than SPIDER, such as the TTA. These compliment the NASA Langley PVS libraries [77]. In general, better libraries are needed, and libraries need to be integrated, both within a mechanical theorem-prover and between theorem provers [115].

2.4. Real-Time Verification by k -Induction. The k -induction verification technique for infinite-state systems used Chapter 5 is promising, and preliminary results suggest larger models can be verified using it than using timed-automata based verification for some problems [54]. More generally, a direct comparison between the specification and verification of real-time systems in SAL and in other tools specifically designed for real-time system verification (e.g., HyTech [116], Kronos [117], Uppaal [118], etc.) would be useful, although results reported by Brown and Pike suggest real-time verification by k -induction compares favorably [19]. More case-studies are needed to determine its how to use the technology effectively. The development of automated solvers is a very active area of research, being shaped by case-studies like the one from Chapter 5 [107]. The k -induction proof technique is sensitive to the maximum length of a trajectory in a transition system over which the induction step does not hold. Other methods to “flatten” the transition system, like the ones developed in Chapter 5, makes the technique more powerful.

2.5. Tool Integration. A thesis of this dissertation is that the economic application of formal methods to this domain requires the use of multiple tools. We use two tools, PVS and SAL, that are developed by a single laboratory, SRI, International. These tools are designed to be integrated in a future release [119]. Although they are stand-alone tools at this point, we are able to take advantage of their similar language constructs, particularly in Chapter 4.

In general, an industrial-sized formal verification endeavor requires a range of tools including mechanical theorem-proving, model-checking, and automated and interactive synthesis [66, 120, 121].

3. Concluding Remarks

Embedded digital systems are at once becoming more complex and more integrated into safety-critical systems. Assurance that their designs are error-free is essential. For systems with high reliability requirements, formal methods are recognized to provide the highest level of assurance of correctness. The feasibility of acquiring this level of assurance requires techniques, like the ones presented herein, for the systematic and economic formal verification of time-triggered systems.

Bibliography

- [1] John Rushby. Tutorial introduction to mechanized formal analysis using theorem proving, model checking and abstraction. Presentation Slides, May 2003. Available at <http://www.csl.sri.com/users/rushby/abstracts/fm-tut>.
- [2] M. Baleani, A. Ferrari, L. Mangeruca, A. Sangiovanni-Vincentelli, Maurizio Peri, and Saverio Pezzini. Fault-tolerant platforms for automotive safety-critical applications. In *CASES '03: Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, pages 170–177, 2003.
- [3] Philip Koopman, editor. *Critical Embedded Automotive Networks*, volume 22–4 of *IEEE Micro*. IEEE Computer Society, July/August 2002.
- [4] Pascal Traverse, Isabelle Lacaze, and Jean Souyris. Airbus fly-by-wire - a total approach to dependability. In *IFIP Congress Topical Sessions*, pages 191–212, 2004.
- [5] Ying C. (Bob) Yeh. Unique dependability issues for commercial airplane fly by wire systems. In *IFIP Congress Topical Sessions*, pages 213–220, 2004.
- [6] Markus Krug, Hermann Kopetz, Elmar Dilger, Lars-Ake Johansson, U. Panizza, Peter Lidn, G. McCall, P. Mortara, Bernd Mller, Stefan Poledna, Anton Schedl, J. Sderberg, M. Strmberg, and Thomas Thurner. Towards an architecture for safety related fault tolerant systems in vehicles. *ESREL '97, June 1997, Portugal*, Jun. 1997.
- [7] John Rushby. Bus architectures for safety-critical embedded systems. In Tom Henzinger and Christoph Kirsch, editors, *EMSOFT 2001: Proceedings of the First Workshop on Embedded Software*, volume 2211 of *Lecture Notes in Computer Science*, pages 306–323, Lake Tahoe, CA, October 2001. Springer-Verlag.
- [8] Hermann Kopetz. *Real-Time Systems*. Kluwer Academic Publishers, 1997.
- [9] Ricky W. Butler and George B. Finelli. The infeasibility of quantifying the reliability of life-critical real-time software. *Software Engineering*, 19(1):3–12, 1993. Available at <http://citeseer.nj.nec.com/butler93infeasibility.html>.

- [10] John Rushby. Systematic formal verification for fault-tolerant time-triggered algorithms. *IEEE Transactions on Software Engineering*, 25(5):651–660, September 1999.
- [11] Wilfredo Torres-Pomales, Mahyar R. Malekpour, and Paul Miner. ROBUS-2: A fault-tolerant broadcast communication system. Technical Report NASA/TM-2005-213540, NASA Langley Research Center, 2005.
- [12] NASA Formal Methods Group. SPIDER homepage. Website, 2004. Available at <http://shemesh.larc.nasa.gov/fm/spider/>.
- [13] Lee Pike. Dissertation artifacts: PVS and SAL specifications and proofs. Website, 2005. Available at <http://www.cs.indiana.edu/~lepike/phd.html>.
- [14] Tony Hoare. The verifying compiler: A grand challenge for computing research. *J. ACM*, 50(1):63–69, 2003.
- [15] J. Strother Moore. A grand challenge proposal for formal methods: A verified stack. In Bernhard K. Aichernig and Tom Maibaum, editors, *Formal Methods at the Crossroads. From Panacea to Foundational Support*, volume 2757 of *LNCS*, pages 161–172. Springer, 2003.
- [16] Lee Pike, Paul Miner, and Wilfredo Torres. Model checking failed conjectures in theorem proving: a case study. Technical Report NASA/TM-2004-213278, NASA Langley Research Center, November 2004. Available at http://www.cs.indiana.edu/~lepike/pub_pages/unproven.html.
- [17] Patrick Lincoln and John Rushby. The formal verification of an algorithm for interactive consistency under a hybrid fault model. In Costas Courcoubetis, editor, *Computer-Aided Verification, CAV '93*, volume 697 of *Lecture Notes in Computer Science*, pages 292–304, Elounda, Greece, June/July 1993. Springer-Verlag. Available at <http://www.csl.sri.com/papers/cav93-hybrid/>.
- [18] Steven D. Johnson. View from the fringe of the fringe. In Tiziana Margaria and Thomas Melham, editors, *11th Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, volume 2144 of *Lecture Notes in Computer Science*, pages 1–12. Springer-Verlag, 2001.

- [19] Geoffrey M. Brown and Lee Pike. Easy parameterized verification of biphasic mark and 8N1 protocols. In *The Proceedings of the 12th International Conference on Tools and the Construction of Algorithms (TACAS'06)*, pages 58–72, 2006. Available at http://www.cs.indiana.edu/~lepik/pub_pages/bmp.html.
- [20] Steven D. Johnson and Paul S. Miner. Integrated reasoning support in system design: Design derivation and theorem proving. In Hon F. li and David K. Probst, editors, *Advances in Hardware Design and Verification: IFIP WG10.5 International Conference on Correct Hardware Design and Verification Methods (CHARME)*, pages 255–272, October 1997.
- [21] Leonardo de Moura, Sam Owre, Harald Ruess, John Rushby, and Natarajan Shankar. Integrating verification components. In *Verified Software: Theories, Tools, Experiments*, October 2005.
- [22] Carl-Johan H. Seger, Robert B. Jones, John W. O’Leary, Tom Melham, Mark D. Aagaard, Clark Barrett, and Don Syme. An industrially effective environment for formal hardware verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(9):1381–1405, September 2005.
- [23] Sam Owre, John Rusby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of pvs. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.
- [24] Saddek Bensalem, Vijay Ganesh, Yassine Lakhnech, César Muñoz, Sam Owre, Harald Rueß, John Rushby, Vlad Rusu, Hassen Saïdi, N. Shankar, Eli Singerman, and Ashish Tiwari. An overview of SAL. In C. Michael Holloway, editor, *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, pages 187–196, Hampton, VA, jun 2000. NASA Langley Research Center. Available at <http://www.csl.sri.com/papers/lfm2000/>.
- [25] John Rushby. An overview of formal verification for the time-triggered architecture. In Werner Damm and Ernst-Rüdiger Olderog, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 2469 of *Lecture Notes in Computer Science*, pages 83–105, Oldenburg, Germany, September 2002. Springer-Verlag.
- [26] Holger Pfeifer. *Formal Analysis of Fault-Tolerant Algorithms in the Time-Triggered Architecture*. PhD thesis, Universität Ulm, 2003. Available at <http://www.informatik.uni-ulm.de/ki/Papers/pfeifer-phd.html>.

- [27] Paul Miner, Alfons Geser, Lee Pike, and Jeffery Maddalon. A unified fault-tolerance protocol. In Yassine Lakhnech and Sergio Yovine, editors, *Formal Techniques, Modeling and Analysis of Timed and Fault-Tolerant Systems (FORMATS-FTRTFT)*, volume 3253 of *Lecture Notes in Computer Science*, pages 167–182. Springer, 2004. Available at http://www.cs.indiana.edu/~lepike/pub_pages/unified.html.
- [28] Lee Pike. A note on inconsistent axioms in Rushby’s *Systematic Formal Verification for Fault-Tolerant Time-Triggered Algorithms*. *IEEE Transactions on Software Engineering*, 2006. To appear. Available at http://www.cs.indiana.edu/~lepike/pub_pages/time.triggered.html.
- [29] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.
- [30] Flaviu Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2), February 1991.
- [31] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *ACM Proceedings: Operating Systems Design and Implementation (OSDI)*, pages 173–186, February 1999.
- [32] Jean-Claude Laprie. Dependability—its attributes, impairments and means. In B. Randell, J.-C. Laprie, H. Kopetz, and B. Littlewood, editors, *Predictability Dependable Computing Systems*, ESPRIT Basic Research Series, pages 3–24. Springer, 1995.
- [33] Paul Miner. Private communication, 2005.
- [34] Paul S. Miner. Verification of fault-tolerant clock synchronization systems. Technical report, NASA Langley Research Center, 1993. Available at <http://techreports.larc.nasa.gov/ltrs/refer/1993/rdp3349.tex.refer.html>.
- [35] Ricky W. Butler. A survey of provably correct fault-tolerant clock synchronization techniques. Technical Report NASA-TM-100553, NASA Langley Research Center, February 1988.
- [36] Kenneth Hoyme and Kevin Driscoll. SAFEbus. In *11th AIAA/IEEE Digital Avionics Systems Conference (DASC)*, pages 68–73, October 1992.
- [37] Jaynarayan H. Lala, Richard E. Harper, and Linda S. Alger. A design approach for ultrareliable real-time systems. *Computer*, 24(5):12–22, 1991.
- [38] Philip Thambidurai and You-Keun Park. Interactive consistency with multiple failure modes. In *7th Reliable Distributed Systems Symposium*, pages 93–100, October 1988.

- [39] Lamport, Shostak, and Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4:382–401, July 1982. Available at <http://citeseer.nj.nec.com/lamport82byzantine.html>.
- [40] Elizabeth Ann Latronico. *Reliability Validation of Group Membership Services for X-by-Wire Protocols*. PhD thesis, Carnegie Mellon University, May 2005. Available at <http://www.ece.cmu.edu/~Ekoopman/thesis/latronico.pdf>.
- [41] John Rushby. Formal methods and digital systems validation for airborne systems. Technical Report CR-4551, NASA, December 1993.
- [42] John Rushby. Proof of Separability—A verification technique for a class of security kernels. In *Proc. 5th International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 352–367, Turin, Italy, April 1982. Springer-Verlag.
- [43] Radio Technical Commission for Aeronautics (RTCA). DO-178b: Software considerations in airborne systems and equipment certification, December 1992.
- [44] Radio Technical Commission for Aeronautics (RTCA). DO-254: Design assurance guidance for airborne electronic hardware, April 2000.
- [45] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.
- [46] G. Leen and D. Heffernan. Expanding automotive electronic systems. *IEEE Computer*, 0018-9162/02:88–93, Jan 2002.
- [47] Hermann Kopetz. TTP—a protocol for fault-tolerant real-time systems. *IEEE Computer*, 27(1):14–23, January 1994.
- [48] Kevin Driscoll, Brendan Hall, Håkan Sivencrona, and Phil Zumsteg. Byzantine fault tolerance, from theory to reality. In G. Goos, J. Hartmanis, and J. van Leeuwen, editors, *Computer Safety, Reliability, and Security*, Lecture Notes in Computer Science, pages 235–248. The 22nd International Conference on Computer Safety, Reliability and Security SAFECOMP, Springer-Verlag Heidelberg, September 2003.
- [49] Holger Pfeifer and Friedrich W. von Henke. Formal modelling and analysis of fault tolerance properties in the time-triggered architecture. In E. Schnieder and G. Tarnai, editors, *Proc. of the 5th Symposium on Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT 2004)*, pages 230–240. Technical University of Braunschweig, Institute for Traffic Safety and Automation Engineering, November 2004.

- [50] Holger Pfeifer and Friedrich W. von Henke. Modular formal analysis of the central guardian in the time-triggered architecture. In Maritta Heisel, Peter Liggesmeyer, and Stefan Wittmann, editors, *Proc. of the 23rd International Conference on Computer Safety, Reliability, and Security (SAFECOMP)*, volume 3219 of *Lecture Notes in Computer Science*, pages 240–253, Potsdam, Germany, September 2004. Springer-Verlag.
- [51] Holger Pfeifer and Friedrich W. von Henke. Formal Analysis for Dependability Properties: the Time-Triggered Architecture Example. In *8th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2001)*, pages 343–352, Antibes Juan-les-Pins, October 2001. IEEE.
- [52] Holger Pfeifer, Detlef Schwier, and Friedrich W. von Henke. Formal verification for time-triggered clock synchronization. In Charles B. Weinstock and John Rushby, editors, *Dependable Computing for Critical Applications – 7*, volume 12 of *Dependable Computing and Fault Tolerant Systems*, pages 207–226. IEEE Computer Society, January 1999.
- [53] Bruno Dutertre and Maria Sorea. Timed systems in SAL. Technical Report SRI-SDL-04-03, SRI, International, July 2004. Available at <http://www.sdl.sri.com/users/bruno/publis.html>.
- [54] Bruno Dutertre and Maria Sorea. Modeling and verification of a fault-tolerant real-time startup protocol using calendar automata. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 3253 of *Lecture Notes in Computer Science*, pages 199–214, Grenoble, France, September 2004. Springer-Verlag. Available at <http://fm.csl.sri.com/doc/abstracts/ftrtft04>.
- [55] Wilfried Steiner, John Rushby, Maria Sorea, and Holger Pfeifer. Model Checking a Fault-Tolerant Startup Algorithm: From Design Exploration To Exhaustive Fault Simulation. In *Proc. of the 2004 International Conference on Dependable Systems and Networks*, pages 189–198, Florence, Italy, June 2004. IEEE Computer Society.
- [56] John Rushby. Formal verification of transmission window timing for the time-triggered architecture. Technical report, SRI, International, March 2001.
- [57] Alfons Geser and Paul Miner. A formal correctness proof of the SPIDER diagnosis protocol. Technical Report NASA/CP-2002-211736, NASA Langley Research Center, Hampton,

- Virginia, August 2002. Technical Report contains the Track B proceedings from Theorem Proving in Higher Order Logics (TPHOLs).
- [58] John Rushby. Systematic formal verification for fault-tolerant time-triggered algorithms. In *Dependable Computing for Critical Applications—6*, volume 11, pages 203–222. IEEE Computer Society, March 1997.
- [59] Ben L. Di Vito and Ricky W. Butler. Formal techniques for synchronized fault-tolerant systems. In C. E. Landwehr, B. Randell, and L. Simoncini, editors, *Dependable Computing for Critical Applications—3*, volume 8 of *Dependable Computing and Fault-Tolerant Systems*, pages 85–97, Vienna, Australia, September 1992. Third IFIP International Working Conference on Dependable Computing for Critical Applications, Springer-Verlag.
- [60] Ricky Butler, James L. Caldwell, and Ben L. Di Vito. Design strategy for a formally verified reliable computing platform. In *Compass '91: 6th Annual Conference on Computer Assurance*, pages 125–134, Gaithersburg, Maryland, 1991. National Institute of Standards and Technology.
- [61] Ricky Butler and Ben Di Vito. Formal design and verification of a reliable computing platform for real-time control (phase 2 results). Technical Report NASA/TM-104196, NASA Langley Research Center, January 1992.
- [62] Jack Goldberg et. al. Development and analysis of the software implemented fault-tolerance (SIFT) computer. CR 172146, NASA, 1984.
- [63] C. J. Walter, R. M. Kieckhafer, and A. M. Finn. MAFT: A multicomputer architecture for fault-tolerance in real-time control systems. In *IEEE Real-Time Systems Symposium*, 1985.
- [64] Fred B. Schneider. Understanding protocols for Byzantine clock synchronization. Technical Report 87-859, Department of Computer Science, Cornell University, August 1987.
- [65] Natarajan Shankar. Mechanical verification of a generalized protocol for Byzantine fault-tolerant clock synchronization. In J. Vytupil, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 571 of *Lecture Notes in Computer Science*, pages 217–236. Springer-Verlag, 1992.
- [66] Paul S. Miner and Steven D. Johnson. Verification of an optimized fault-tolerant clock synchronization circuit. In Mary Sheeran and Satnam Singh, editors, *Designing Correct Circuits*. Electronic Workshops in Computing, Springer, September 1996.

- [67] D. Schwier and F. von Henke. Mechanical verification of clock synchronization algorithms. In *Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT)*, volume 1486 of *Lecture Notes in Computer Science*, pages 262–271. Springer-Verlag, 1998.
- [68] Shmuel Katz, Pat Lincoln, and John Rushby. Low-overhead time-triggered group membership. In Marios Mavronicolas and Philippas Tsigas, editors, *11th International Workshop on Distributed Algorithms (WDAG '97)*, volume 1320 of *Lecture Notes in Computer Science*, pages 155–169. Springer-Verlag, September 1997. Available at <http://www.csl.sri.com/papers/wdag97/>.
- [69] John Rushby. SAL tutorial: Analyzing the fault-tolerant algorithm OM(1). Technical Report CSL Technical Note, SRI International, 2004. Available at <http://www.csl.sri.com/users/rushby/abstracts/om1>.
- [70] Patrick Lincoln and John Rushby. Formal verification of an interactive consistency algorithm for the Draper FTP architecture under a hybrid fault model. In *Compass '94 (Proceedings of the Ninth Annual Conference on Computer Assurance)*, pages 107–120, Gaithersburg, MD, June 1994. IEEE Washington Section. Available at <http://www.csl.sri.com/papers/compass94/>.
- [71] William D. Young. Comparing verification systems: Interactive consistency in ACL2. *IEEE Transactions on Software Engineering*, 23(4):214–223, April 1997.
- [72] Patrick Lincoln and John Rushby. A formally verified algorithm for interactive consistency under a hybrid fault model. Technical Report SRI-CSL-93-2, Computer Science Laboratory, SRI International, mar 1993. Available at <http://www.csl.sri.com/papers/csl-93-2/>. Also available as NASA Contractor Report 4527, July 1993.
- [73] Gordon M.J.C. HOL: A proof generating system for higher-order logic. In G. BirtWistle and P.A. Subramanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 73–128. Kluwer, 1988.
- [74] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
- [75] Ben L. Di Vito. A PVS prover strategy package for common manipulations. Technical Report NASA/TM-2002-211647, NASA Langley Research Center, April 2002.

- [76] C. Muñoz and M. Mayero. Real automation in the field. Technical Report NASA/CR-2001-211271 Interim ICASE Report No. 39, ICASE-NASA Langley, ICASE Mail Stop 132C, NASA Langley Research Center, Hampton VA 23681-2199, USA, December 2001.
- [77] NASA Langley Formal Methods Group. NASA Langley PVS libraries. Website. Available at <http://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library/pvslib.html>.
- [78] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference*. SRI International, version 2.4 edition, December 2001. Available at <http://pvs.csl.sri.com/manuals.html>.
- [79] M. Archer, B. Di Vito, and C. Munoz, editors. *Proceedings of Design and Application of Strategies/Tactics in Higher Order Logics (STRATA 2003)*. NASA Technical Report CP-2003-212448, 2003.
- [80] S. Owre and N. Shankar. The PVS prelude library. Technical Report SRI-CSL-03-01, SRI, International, March 2003. Available at <http://pvs.csl.sri.com/documentation.shtml>.
- [81] SRI International. Symbolic analysis laboratory SAL, 2004. Available at <http://sal.csl.sri.com/>.
- [82] Leonardo de Moura, Sam Owre, Harald Ruess, John Rushby, and N. Shankar. The ICS decision procedures for embedded deduction. In *2nd International Joint Conference on Automated Reasoning (IJCAR)*, volume 3097 of *Lecture Notes in Computer Science*, pages 218–222, Cork, Ireland, July 2004. Springer-Verlag.
- [83] Shuvendu K. Lahiri and Sanjit A. Seshia. The UCLID decision procedure. In *CAV'04: Proceedings of the 16th International Conference on Computer Aided Verification*, pages 475–478, 2004.
- [84] Aaron Stump, Clark W. Barrett, and David L. Dill. CVC: A cooperating validity checker. In *CAV '02: Proceedings of the 14th International Conference on Computer Aided Verification*, pages 500–504, London, UK, 2002. Springer-Verlag.
- [85] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT-solver. In *FMCAD '00: Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*, pages 108–125. Springer-Verlag, 2000. Available at <http://www.cs.chalmers.se/~ms/>.

- [86] Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Computer Science*, 89(4), 2003.
- [87] Klaus Havelund and Natarajan Shankar. Experiments in theorem proving and model checking for protocol verification. In *Proceedings of Formal Methods Europe FME'96*, Lecture Notes in Computer Science. Springer, 1996.
- [88] John Rushby. Verification diagrams revisited: Disjunctive invariants for easy verification. In E. A. Emerson and A. P. Sistla, editors, *Computer-Aided Verification, CAV '2000*, volume 1855 of *Lecture Notes in Computer Science*, pages 508–520, Chicago, IL, July 2000. Springer-Verlag. Available at <http://www.csl.sri.com/users/rushby/abstracts/cav00>.
- [89] Dilsun K. Kaynar, Nancy Lynch, Roberto Segala, and Frits Vaandrager. *Synthesis Lectures on Computer Science*, chapter The Theory of Timed I/O Automata. Morgan Claypool Publishers, 2005. Available at <http://theory.lcs.mit.edu/tds/lynch-pubs.html>.
- [90] Rajeev Alur. Timed automata. In *11th International Conference on Computer-Aided Verification*, Lecture Notes in Computer Science, pages 8–22. Springer-Verlag, 1999. Available at <http://www.cis.upenn.edu/~alur/onlinepub.html>.
- [91] Leslie Lamport. Real time is really simple. Technical Report MSR-TR-2005-30, Microsoft Research, March 2005. Available at <ftp://ftp.research.microsoft.com/pub/tr/TR-2005-30.pdf>.
- [92] Jerry Banks and John S. Carson II. *Discrete-Event Simulation*. Prentice-Hall, 1984.
- [93] T. Melham. *Higher Order Logic and Hardware Verification*, volume 31 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1993.
- [94] T. F. Melham. Abstraction mechanisms for hardware verification. In G. Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification, Verification, and Synthesis*, pages 129–157, Boston, 1988. Kluwer Academic Publishers. Available at <http://citeseer.nj.nec.com/melham87abstraction.html>.
- [95] Lee Pike, Jeffery Maddalon, Paul Miner, and Alfons Geser. Abstractions for fault-tolerant distributed system verification. In Konrad Slind, Annette Bunker, and Ganesh Gopalakrishnan, editors, *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 3223 of *Lecture Notes in Computer Science*, pages 257–270. Springer, 2004. Available at http://www.cs.indiana.edu/~lepik/pub_pages/abstractions.html.

- [96] Mohammad H. Azadmanesh and Roger M. Kieckhafer. Exploiting omissive faults in synchronous approximate agreement. *IEEE Transactions on Computers*, 49(10):1031–1042, 2000.
- [97] Maxwell Rosenlicht. *Introduction to Analysis*. Dover Publications, Inc., 1968.
- [98] Daniel Davies and John F. Wakerly. Synchronization and matching in redundant systems. *IEEE Transactions on Computers*, 27(6):531–539, June 1978.
- [99] T. K. Srikanth and S. Toueg. Optimal clock synchronization. *Journal of the ACM*, 34(3):626–645, July 1987.
- [100] Michael R. Huth and Mark D. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2000.
- [101] Leonardo de Moura, Harald Rueß, and Maria Sorea. Bounded model checking and induction: From refutation to verification. In Andrei Voronkov, editor, *Computer-Aided Verification, CAV 2003*, volume 2725 of *Lecture Notes in Computer Science*, pages 14–26. Springer-Verlag, 2003.
- [102] Martín Abadi and Leslie Lamport. An old-fashioned recipe for real time. *ACM Transactions on Programming Languages and Systems*, 16(5):1543–1571, September 1994.
- [103] Zohar Manna and Amir Pnueli. Temporal verification diagrams. In *TACS '94: Proceedings of the International Conference on Theoretical Aspects of Computer Software*, pages 726–765. Springer-Verlag, 1994.
- [104] Paul S. Miner, Mahyar Malekpour, and Wilfredo Torres. Conceptual design of a reliable optical bus (ROBUS). In *21st AIAA/IEEE Digital Avionics Systems Conference DASC*, Irvine, CA, October, 2002.
- [105] Günther Bauer and Michael Paulitsch. An investigation of membership and clique avoidance in TTP/C. In *19th IEEE Symposium on Reliable Distributed Systems*, pages 118–124, 2000.
- [106] Geoffrey M. Brown and Lee Pike. "easy" parameterized verification of cross domain clock protocols. In *Seventh International Workshop on Designing Correct Circuits DCC: Participants' Proceedings*, 2006. Satellite Event of ETAPS. To appear. Available at http://www.cs.indiana.edu/~lepik/pub_pages/dcc.html.
- [107] C. Barrett, L. de Moura, and A. Stump. SMT-COMP: Satisfiability Modulo Theories Competition. In K. Etessami and S. Rajamani, editors, *17th International Conference on Computer Aided Verification*, pages 20–23. Springer, 2005.

- [108] W. Bevier and W. Young. The proof of correctness of a fault-tolerant circuit design. In *Second IFIP Conference on Dependable Computing For Critical Applications*, 1991. Available at <http://citeseer.ist.psu.edu/bevier91proof.html>.
- [109] Steven D. Johnson. *Synthesis of Digital Designs from Recursion Equations*. ACM Distinguished Dissertations. MIT Press, 1983.
- [110] Bhaskar Bose. *DDD-FM9001: Derivation of a Verified Microprocessor*. PhD thesis, Indiana University, December 1994.
- [111] Kamlesh Rath and Steven D. Johnson. Toward a basis for protocol specification and process decomposition. In D. Agnew, L. Claesen, and R. Camposano, editors, *Computer Hardware Description Languages and their Applications (CHDL'93)*, April 1993.
- [112] A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, and P. Le Guernic and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 2003.
- [113] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow programming language Lustre. *IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems*, page September, 1992.
- [114] Paul Caspi, Adrian Curic, Aude Maignan, Christos Sofronis, Stavros Tripakis, and Peter Niebert. From simulink to SCADE/lustre to TTA: a layered approach for distributed embedded applications. In *Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 153–162. ACM Press, 2003.
- [115] Jason Hickey. NuPRL-Light: An implementation framework for higher-order logics. In William McCune, editor, *Proceedings of the 14th International Conference on Automated Deduction*, volume 1249, pages 395–399. Springer, July 1997.
- [116] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HYTECH: A model checker for hybrid systems. *International Journal on Software Tools for Technology Transfer*, 1(1-2):110–122, 1997. Available at <http://citeseer.ist.psu.edu/henzinger97hytech.html>.
- [117] Conrado Daws, Alfredo Olivero, Stavros Tripakis, and Sergio Yovine. The tool KRONOS. In *Hybrid Systems*, pages 208–219, 1995.
- [118] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *STTT*, 1(1-2):134–152, 1997.

- [119] SRI Computer Science Laboratory. Formal methods roadmap: PVS, ICS, and SAL. Technical Report SRI-CSL-03-05, SRI International, Menlo Park, CA 94025, November 2003.
- [120] R. Jones, J. O’Leary, C. Seger, M. Aagaard, and T. Melham. Practical formal verification in microprocessor design. *IEEE Design and Test*, pages 16–25, July 2001.
- [121] Paul S. Miner. *Hardware Verification using Coinductive Assertions*. PhD thesis, Indiana University, Bloomington, 1998.
- [122] Sam Owre and N. Shankar. Theory interpretations in PVS. Technical Report SRI-CSL-01-01, SRI, International, April 2001. Available at <http://pvs.csl.sri.com/documentation.shtml>.

Index

- 0-trajectory, 34
- C_p , 65, 159, 160
- D , 66
- P , 66
- R , 72
- Λ , 71
- Σ , 66
- δ , 66
- e_l , 72
- e_u , 72
- gs , 68
- independent*, 72
- δ_{nom} , 72
- ρ , 66
- sched*, 65
- sched_p*, 71
- sendtime_p*, 68
- α , 98
- ttin_p*, 68
- k -induction, 34
- k -inductive property, 35
- k -trajectory, 34
- accepted messages, 46
- accuracy, 12
- accusations, 111
- accuse, 85, 112
- accuses, 19
- agreement, 19, 54
- antecedent, 28
- application level, 14
- architecture, 3
- assignment functions, 98
- asymmetric, 16, 49
- asynchronous composition, 34

- behavior, 3
- behavioral partitioning, 17
- benign, 15, 49
- benign faults, 110
- benign messages, 15, 46
- BIU, 13
- broadcast communication round, 55
- bus, 13
- bus architecture, 13
- bus interface unit, 130

- bus interface units, 13, 21
- bus level, 14
- Byzantine, 16, 49

- channels, 10
- choice operator, 70
- clique, 19, 22
- clique avoidance, 129
- clique initialization mode, 22
- clique join mode, 22
- clique preservation mode, 22
- clique-detection mode, 22
- clock, 100
- clock drift rate, 66
- clock function, 135
- clock skew, 66
- clock synchronization protocol, 5, 22
- clock-time, 65
- clock-time simulation, 79
- communication abstractions, 43
- communication channels, 10
- communication delay, 66
- communication offset, 66
- communication phase, 38, 45
- computation offset, 66
- computation phase, 38, 45
- consensus, 19
- consequent, 28
- conservatively extend, 27
- control applications, 13

- conviction, 86
- coordination, 3

- data, 3
- diagnostic correctness, 8
- diagnostic data, 108
- disabled mode, 22
- discrete transitions, 40, 100
- disjunctive invariants, 127
- distributed diagnosis protocol, 5, 22
- distributed system, 10
- DMFA, 23
- Dynamic Maximum Fault Assumption, 23
- dynamic schedule, 12

- echo, 63, 110
- echo messages, 110
- edge, 99
- emergent properties, 136
- enabled edge, 100, 105
- enabled timeout component, 100, 105
- error, 10
- error types, 47
- event-triggered, 63, 119
- event-triggered implementations, 39
- event-triggering, 11
- exact agreement, 54
- exact communication, 54
- exact function, 55
- exact functions, 54
- exact validity, 54

- fail-silent, 110
- failure, 10
- fault, 10
- fault abstractions, 43
- fault containment region*, 15
- fault persistence, 107
- fault types, 47, 49
- fault-masking abstractions, 43
- fault-masking vote, 51
- fault-tolerant, 14
- fault-tolerant distributed system, 11
- fault-tolerant system, 10
- FCR, 15
- finite types, 33
- formalization, 5
- formulation, 5
- frame, 110, 111
- frame gap, 114
- frame length, 111
- frame synchronization, 112
- function agreement, 55, 57
- function extensionality, 30
- functional model, 43, 54
- functions, 16

- global state, 45
- good, 15, 49
- group membership, 19

- hosts, 13
- hybrid fault model, 15, 49

- ICS, 34
- inbound neighbors, 44
- inexact agreement, 55
- inexact communication, 54
- inexact function, 55, 56, 58
- inexact functions, 54
- inexact validity, 54
- inference rule, 31
- inference rules, 28
- infinite types, 33
- initial sequent, 28
- initial states, 44
- Integrated Canonizer and Solver, 34
- integration, 14
- interactive compilation, 137
- interactive consistency protocol, 5, 22
- interactive synthesis, 137
- interconnect, 13
- inverse clock, 65
- inverse clock function, 135
- isolation*, 17

- latency, 18
- lower function error, 58

- MAFT, 25
- majority good, 55, 57, 58
- manifest, 15, 49
- maximum fault assumption, 16, 108
- message abstractions, 43
- message-generation function, 44

- MFA, 16
- module, 33

- node-level specification, 137
- nodes, 10
- nonZeno Property, 101
- null, 103

- operational clique, 108
- operational nodes, 108
- outbound neighbors, 44

- partially-synchronous model, 2, 62
- partition, 17
- partitioning, 14
- performance guarantees, 18
- permanent faults, 107
- physical partitioning, 17
- physical plants, 13
- power-up, 12
- pre-computation phase, 69
- precision, 12
- predictability, 14
- preliminary diagnosis, 112
- processes, 10
- processor elements, 20
- proof strategies, 28, 32
- protocol-level specification, 136
- Prototype Verification System, 26
- PVS, 5, 6, 8, 24–28, 33, 43, 52, 55, 62, 85, 138, 139

- PVS Prelude, 28

- RCP, 25
- real time, 11, 65
- real-time simulation, 79
- realization, 2
- receiver, 10
- receiving node, 10
- reception window, 71
- reception window offset, 72
- Reception Window Open, 77
- reconfigure, 19
- redundancy management units, 21
- reintegrate, 13, 20
- reintegration protocol, 5, 23, 86, 96, 108
- reintegrator, 108
- relational model, 43, 54
- Reliable Computing Platform, 25
- Reliable Optical Bus, 20
- restart, 12
- resynchronization frame, 110
- RMU, 21
- ROBUS, 20
- round, 38
- round-based pipelining, 64, 72

- SAL, 7, 24, 25, 33, 34, 39, 64, 85–87, 89, 91, 92, 94–97, 104, 106, 107, 115, 116, 121, 122, 129, 131–133, 138, 139

- sample, 54
- schedule, 65

- schedule constraints, 66
- schedule update protocol, 22
- security kernels, 17
- self-test mode, 22
- sender, 10
- sending node, 10
- sequent, 28
- sequent calculus, 28
- shallow embedding, 107
- SIFT, 25
- single point of failure, 10
- skew constant, 112
- skolem constants, 32
- source host, 19
- SPIDER, 13, 20, 21, 25, 62, 85, 86, 107–109, 111, 128, 130–132, 134, 135, 138
- SPIDER Clock Synchronization Protocol, 43
- SPIDER Clock Synchronization Protocol, 24, 64, 71, 91, 92, 95, 110
- SPIDER Distributed Diagnosis Protocol, 43
- SPIDER Distributed Diagnosis Protocol, 24, 25, 64, 85, 87, 89, 91, 92, 94, 95, 109
- SPIDER Interactive Consistency Protocol, 24, 43
- SPIDER Interactive Consistency Protocol, 25, 94
- SPIDER Reintegration Protocol, 97
- STA, 97, 99, 101, 103–107, 115, 116
- startup protocol, 96
- state-transition function, 44
- static schedule, 12
- Symbolic Analysis Laboratory, 33
- symmetric, 16, 49
- synchronization capture, 112
- synchronizing timeout automaton, 98
- synchronous composition, 33
- synchronous model, 2
- system assumptions, 66
- TDMA, 18
- temporal induction, 34
- throughput, 18
- ticks, 65
- time progress transition, 100
- time progress transitions, 40
- time-division multi-access, 18
- time-slice, 68
- time-triggered bus architecture, 14
- time-triggered implementations, 38
- time-triggered inbound messages, 68
- time-triggered model, 2, 38, 62
- time-triggered system state, 68
- time-triggered systems, 1
- time-triggering, 11
- timed automata, 39
- timeout, 98
- timeout automata, 39
- timeout component, 98
- timeout variables, 40
- timeout vectors, 98

TOm, 98

transient fault, 19, 25, 96

triggers, 11

trusted set, 23

TTA, 7, 20, 24, 25, 107, 109, 135, 137

turnstile, 31

type declaration, 28

type-correctness condition, 27

uniprocessor system layer, 25

unlabeled transition system, 34

untimed model, 2

untimed synchronous model, 38

upper function error, 58

validity, 54

APPENDIX A

Inconsistent Axioms in Rushby’s Specification

Rushby presents four principle assumptions (or axioms) about the behavior of time-triggered systems [10]. He describes his use of these axioms in the systematic formal specification and verification of time-triggered systems in the mechanical theorem-prover PVS [23]. Two of these four axioms are inconsistent; in fact, one is inconsistent in three separate ways. Once the axioms are made consistent, one axiom is redundant; it is a corollary of the other. Finally, a contradiction can be derived from another of the four axioms and some other minor axioms in the formal specification. These inconsistencies appear in both the printed paper and the PVS specifications, but when the printed axioms are ambiguous due to being more informally stated, we assume the PVS specifications to be definitive.

These errors were discovered while attempting to interpret these axioms by formally providing a model using theory interpretations in PVS [122]. When the “canonical model” did not satisfy the axioms,¹ These axioms not only fail to model the domain but are in fact inconsistent. Once the errors were discovered, it is straightforward to mend them.²

We begin by stating Rushby’s definition of inverse clocks and Clock Drift Rate Axiom.

¹The author thanks Paul Miner of the NASA Langley Formal Methods Group for suggesting Axioms A.3 and A.5 are necessary to axiomatize a canonical clock. He also pointed out that these changes imply that Theorem A.10 holds.

²This appendix is a preliminary version of a short comment [28]. The mended formal specifications, along with a formal theory interpretation, can be found on-line [13].

DEFINITION A.1 (Inverse Clock). An inverse clock for node p is a total function $C_p : \mathbb{R} \rightarrow \mathbb{N}$.

The domain of an inverse clock is called *realtime* and the range is called *clocktime*. The drift of nonfaulty clocks is bounded by a realtime constant $0 < \rho < 1$:

AXIOM A.1 (Clock Drift Rate). $(1 - \rho)(t_1 - t_2) \leq C_p(t_1) - C_p(t_2) \leq (1 + \rho)(t_1 - t_2)$.

THEOREM A.2. *Axiom A.1 is inconsistent.*

PROOF. Let $t_2 > t_1$. Then $(1 - \rho)(t_1 - t_2) > (1 + \rho)(t_1 - t_2)$. □

Axiom A.1 can be revised as follows:

AXIOM A.3 (Clock Drift Rate (First Revision)). *Let $t_1 \geq t_2$. Then $(1 - \rho)(t_1 - t_2) \leq C_p(t_1) - C_p(t_2) \leq (1 + \rho)(t_1 - t_2)$.*

However, even this is unsatisfiable:

THEOREM A.4. *Axiom A.3 is inconsistent.*

PROOF. Let $t_1 > t_2$ such that $(1 + \rho)(t_1 - t_2) - (1 - \rho)(t_1 - t_2) < 1$ and there exists no $n \in \mathbb{N}$ such that $(1 - \rho)(t_1 - t_2) \leq n \leq (1 + \rho)(t_1 - t_2)$. □

The inequality is weakened by taking the floor and ceiling of the drifts:

AXIOM A.5 (Clock Drift Rate (Second Revision)). *Let $t_1 \geq t_2$. Then $\lfloor (1 - \rho)(t_1 - t_2) \rfloor \leq C_p(t_1) - C_p(t_2) \leq \lceil (1 + \rho)(t_1 - t_2) \rceil$.*

Even with these revisions, no function satisfying Axiom A.5 is an inverse clock, as defined by Definition A.1.³

³It should already be intuitive that Definition A.1 is incorrect, since, e.g., a canonical inverse clock function like the floor function does not satisfy Axiom A.5.

THEOREM A.6. *No inverse clock satisfies Axiom A.5.*

PROOF. By contradiction. The set \mathbb{N} is totally ordered with a least element, so there exists some $t \in \mathbb{R}$ such that $C_p(t) \leq C_p(t')$ for all $t' \in \mathbb{R}$. Let $t'' \in \mathbb{R}$, where $t'' < t$, such that $\lfloor (1 - \rho)(t - t'') \rfloor > 0$. By Axiom A.5, $\lfloor (1 - \rho)(t - t'') \rfloor + C_p(t'') \leq C_p(t)$. However, because $\lfloor (1 - \rho)(t - t'') \rfloor$ is assumed to be strictly greater than zero, $C_p(t'') < C_p(t)$, contradicting our assumption that $C_p(t)$ is least. \square

We therefore extend the range of an inverse clock from \mathbb{N} to \mathbb{Z} .

DEFINITION A.2 (Revised Inverse Clock). An inverse clock for node p is a total function $C_p : \mathbb{R} \rightarrow \mathbb{Z}$.

Note that the inconsistencies in Axioms A.1 and A.3 hold regardless of whether an inverse clock is defined by Definition A.1 or Definition A.2.

A second inconsistent axiom is the Monotonicity Axiom. Nonfaulty clocks are monotonic:

AXIOM A.7 (Monotonicity). $t_1 < t_2$ implies $C_p(t_1) < C_p(t_2)$.

THEOREM A.8. *Axiom A.7 is inconsistent (with respect to either Definition A.1 or Definition A.2).*

PROOF. Because $<$ is a total order over \mathbb{R} , Axiom A.7 implies that C_p is an injective function, but there exists no injection from the reals into the natural numbers (or integers). \square

A satisfiable revision of monotonicity weakens the consequent:

AXIOM A.9 (Revised Monotonicity). $t_1 < t_2$ implies $C_p(t_1) \leq C_p(t_2)$.

Axiom A.9 now becomes a corollary of Axiom A.5:

COROLLARY A.10. *Let Axiom A.5 hold. Prove Axiom A.9.*

PROOF. By Axiom A.5, $C_p(t_2) \geq C_p(t_1) + \lfloor (1 - \rho)(t_2 - t_1) \rfloor$. \square

The third inconsistency can be derived from the axiomatization of when messages are sent and received by nonfaulty nodes. Let $sent_p(q, m, t)$ be a relation that holds if node p sends message m to node q at realtime t . Similarly, let $recv_q(p, m, t)$ be a relation that holds if node q receives message m from node p at realtime t . The following axiom relates the delay between when a nonfaulty node sends a message and when a nonfaulty node receives it. Let the maximum delay be a realtime constant such that $\delta \geq 0$.

AXIOM A.11 (Maximum Delay). *$sent_p(q, m, t)$ if and only if there exists some realtime delay $0 \leq d \leq \delta$ such that $recv_q(p, m, t + d)$.*

THEOREM A.12. *If $\delta > 0$, then Axiom A.11, together with other minor axioms and constraints in the formal specification, is inconsistent.*

PROOF. (Sketch.) The essential problem is that the existential quantifier is within the scope of the biconditional operator in Axiom A.11. As stated, Axiom A.11 implies that for all realtimes t , if there exists a $0 \leq d \leq \delta$ such that $recv_q(p, m, t + d)$, then $sent_p(q, m, t)$. It can be shown that there exists some t such that $recv_q(p, m, t + d)$. Because d ranges over the interval $[0, \delta]$, there exists a realtime t' and realtime delay $0 \leq d' \leq \delta$ such that $d' \neq d$ and $t' + d' = t + d$, implying that $sent_p(q, m, t)$ and $sent_p(q, m, t')$, where the distance between t and t' is less than δ . However, by other constraints, no two separate realtimes within δ of each other satisfy $sent$. \square

A possible consistent revision is as follows:

AXIOM A.13 (Revised Maximum Delay). *There exists some $0 \leq d \leq \delta$ such that $\text{sent}_p(q, m, t)$ implies $\text{recv}_q(p, m, t + d)$, and there exists some $0 \leq d' \leq \delta$ such that $\text{recv}_q(p, m, t)$ implies $\text{sent}_p(q, m, t - d')$.*

Finally, the following axiom is unnecessary:

AXIOM A.14 (Monotone Schedule). $r_1 < r_2$ implies $\text{sched}(r_1) < \text{sched}(r_2)$.

The axiom can be replaced with the following corollary.

COROLLARY A.15 (Monotone Schedule). $\text{sched}(r) < \text{sched}(r + 1)$.

PROOF. Immediate from Axiom 4.5.

□