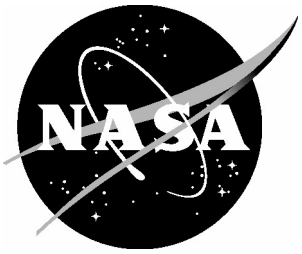
NASA/TM-2005-213751



Real-Time System Verification by *k*-Induction

Lee Pike
Langley Research Center, Hampton, Virginia

April 2005

The NASA STI Program Office . . . in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA Scientific and Technical Information (STI) Program Office plays a key part in helping NASA maintain this important role.

The NASA STI Program Office is operated by Langley Research Center, the lead center for NASA's scientific and technical information. The NASA STI Program Office provides access to the NASA STI Database, the largest collection of aeronautical and space science STI in the world. The Program Office is also NASA's institutional mechanism for disseminating the results of its research and development activities. These results are published by NASA in the NASA STI Report Series, which includes the following report types:

- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counterpart of peer-reviewed formal professional papers, but having less stringent limitations on manuscript length and extent of graphic presentations.

- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.

- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.

- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.

- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.

- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services that complement the STI Program Office's diverse offerings include creating custom thesauri, building customized databases, organizing and publishing research results ... even providing videos.

For more information about the NASA STI Program Office, see the following:

- Access the NASA STI Program Home Page at ***http://www.sti.nasa.gov***

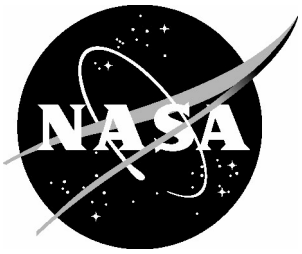
- E-mail your question via the Internet to help@sti.nasa.gov

- Fax your question to the NASA STI Help Desk at (301) 621-0134

- Phone the NASA STI Help Desk at (301) 621-0390

- Write to:
  NASA STI Help Desk
  NASA Center for AeroSpace Information
  7121 Standard Drive
  Hanover, MD 21076-1320

NASA/TM-2005-213751



Real-Time System Verification by *k*-Induction

Lee Pike
Langley Research Center, Hampton, Virginia

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

April 2005

Acknowledgments

This work was supported by NASA's Vehicle Systems Program. The author thanks Wilfredo Torres-Pomales for describing the reintegration protocol in detail and repeatedly to the author. Other members of the SPIDER design team (Paul Miner, Alfons Geser, Jeffrey Maddalon, and Mahyar Malekpour) provided helpful comments. Bruno Dutertre and Leonardo de Moura provided helpful details concerning SAL and *k*-induction.

Real-Time System Verification by k-Induction

Lee Pike

NASA Langley Research Center

Abstract

We report the first formal verification of a reintegration protocol for a safety-critical, fault-tolerant, real-time distributed embedded system. A reintegration protocol increases system survivability by allowing a node that has suffered a fault to regain state consistent with the operational nodes. The protocol is verified in the Symbolic Analysis Laboratory (SAL), where bounded model checking and decision procedures are used to verify infinite-state systems by k-induction. The protocol and its environment are modeled as *synchronizing timeout automata*. Because k-induction is exponential with respect to k, we optimize the formal model to reduce the size of k. Also, the reintegrator's event-triggered behavior is conservatively modeled as time-triggered behavior to further reduce the size of k and to make it invariant to the number of nodes modeled. A corollary is that a clique avoidance property is satisfied.

Contents

1 Introduction

Digital control (i.e., "x-by-wire") systems are being designed for use in safety-critical environments such as automobiles, commercial aircraft, and piloted space vehicles. In a single vehicle, many systems require reliable real-time intercommunication. Highly-reliable fault-tolerant virtual busses are being designed for this purpose [Rus01].[1] Some notable examples of such busses include TTTech's Time-Triggered Architecture (TTA) [Kop94], Honeywell's SAFEbus [HD92], FlexRay (being developed by an automotive consortium) [LH02], and NASA Langley Research Center's SPIDER [MMT02, MGPM04, NAS04].

These busses are implemented as distributed systems to increase their fault-tolerance. A node in the distributed system may suffer a *transient fault* causing it to lose its volatile state but suffer no permanent damage. Although such a node may be fault-free, its state no longer is coordinated with that of the *operational clique*, the set of fault-free nodes with coordinating states allowing them to provide the requested services of the system.[2] 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 extremely high reliability requirements for these busses coupled with the potential for a high number of transient faults in the environments in which they may 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.

To the author's knowledge, we present the first formal verification of a reintegration protocol. In [Rus02], Rushby describes the formal verification of TTA, one of the most mature and fully formally-verified busses in development, and therein states that the formal analysis of reintegration remains important future work. The work presented here should be extensible to other fault-tolerant systems that employ reintegration protocols, especially given that our verification is architecture-independent

[1] Rushby notes that the term 'bus' "understates their complexity, sophistication, and criticality," [Rus01].

[2] 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 [PMT04].

(see Sec. 7).

This work extends results in using bounded model checking and decision procedures to verify infinite-state systems using k-induction (also known as temporal induction), a generalization of induction over transition systems. In particular, we build on Dutertre and Sorea's work in which they develop a timeout automata model for specifying and verifying real-time systems [DS04b, DS04a]. The formalism is particularly well-suited for k-induction proofs over transition systems, and it does not require specialized algorithms for model checking (as opposed to, e.g., timed automata [Alu99]).

Our focus is to make the k-induction technique feasible for large systems; this amounts to reducing the size of k required for k-induction verification. We follow two approaches to do so. First, we extend the timed automata model so that real-time systems containing both synchronous and asynchronous components can be described more easily. We call these *Synchronizing Timeout Automata* (STA). Introducing synchrony often reduces the size of k required. Second, we optimize the semantics so that the constructed transition system includes no time transitions; all transitions are ones in which discrete state is updated. This can greatly reduce the depth at which k-induction must be applied to prove a given safety property. We also describe a means by which to model conservatively event-triggered physical behavior as in a time-triggered behavior. Such a model may contain significantly fewer state transitions than the physical system contains. Both kinds of optimizations are necessary to complete the verification of the reintegration protocol.

Organization In Sect. 2, we describe the SAL toolset and the k-induction proof technique in SAL. In 3, we describe Dutertre and Sorea's timeout automata. We then define synchronizing timeout automata (STA) and we present a STA model of the train-gate-controller, a canonical example of a real-time system. We describe the SPIDER Reintegration Protocol in Sect. 4, and in Sect. 5, we describe how the protocol is modeled as a timeout automaton in SAL. Additionally, we describe how we modeled event-triggered behavior as time-triggered behavior to ease the verification. In Sect. 6, we describe the verification of the protocol, and concluding remarks are given in Sect. 7.

2 Symbolic Analysis Laboratory (SAL) and k-Induction

This protocol was specified and verified in the Symbolic Analysis Laboratory (SAL) [BGL+00, SRI04], developed by SRI, International. SAL is a verification environment that includes explicit-state, symbolic, and bounded model checkers, an interactive simulator, as well as other tools. A single language serves as the input to the verification tools. The language includes a type system, quantification over finite domains, uninterpreted constants and functions, and synchronous and asynchronous composition

operators. SAL may be downloaded at [SRI04], free of charge, for non-commercial use.

The verification tools used here were SAL's bounded model checker in conjunction with the Integrated Canonizer and Solver (ICS), a decision procedure for a quantifier-free, first-order theory of equality, the terms of which include uniterpreted functions, linear arithmetic, products, arrays, fixed-sized vectors, etc. [dMOR+04]. Although ICS is the default decision procedure in SAL, other decision procedures such as UCLID, CVC, and SVC may be used [dMOR+04].

Together, these tools can be used to prove state invariants hold in infinite 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) [SSS00, ES03]. 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. A 0-*trajectory* (over the transition system) is a state s. For $k \in \mathbb{N}^{0<}$, a k-*trajectory* is a sequence of states, $s_0, s_1, \ldots, s_k$, such that for $0 \leq i < k$, $s_i \rightarrow s_{i+1}$. Then the k-induction principle is as follows.

Definition 1 (k-Induction Principle). Let $k \in \mathbb{N}^{0<}$, and let $P : S \rightarrow$ *bool* be some predicate defined over states of S.

- *Base Case*: For all $0 \leq j < k$, show that for each j-trajectory $s_0, s_1, \ldots, s_j$ such that $s_0 \in S^0$, $P(s_j)$ holds.
- *Induction Step*: Show that if $s_0, s_1, \ldots, s_{k-1}$ is a $(k-1)$-trajectory, and for all $0 \leq j < k$, $P(s_j)$ holds, then for all $s_k \in S$ such that $s_{k-1} \rightarrow s_k$, $P(s_k)$ 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 exceedingly 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 [HS96, Rus00].

Furthermore, SAL allows state invariants to 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. That is, if Q is an invariant over states, then the principle is as stated in Def. 2.

Definition 2 (k-Induction Principle with Inductive Invariants).

- *Base Case*: For all $0 \leq j < k$, show that for each j-trajectory $s_0, s_1, \ldots, s_j$ such that $s_0 \in S^0$ holds and for each $0 \leq i < k$, $Q(s_i)$ holds, $P(s_j)$ holds.
- *Induction Step*: Show that if $s_0, s_1, \ldots, s_{k-1}$ is a $(k-1)$-trajectory, and for all $0 \leq j < k$, $Q(s_j)$ and $P(s_j)$ hold, then for all $s_k \in S$ such that $s_{k-1} \rightarrow s_k$ and $Q(s_k)$, $P(s_k)$ holds.

Multiple invariants may be simultaneously used by taking their conjunction to be the invariant.

Other systems such as NuSMV [CCO+04] implement k-induction (its implementors call it "een-sorensson") via bounded model checking. However, the author knows of no other tools that integrate k-induction with decision procedures to verify infinite-state systems.

3 Timed Systems in SAL

Dutertre and Sorea explore the verification of infinite-state timed transition systems via k-induction in SAL [DS04b]. They first consider specifying these systems as timed automata [Alu99], one of the most prominent formalisms for the specification and verification of real-time systems. automata are a well-known Although they demonstrate that it is possible to specify timed automata in SAL via a shallow embedding (i.e., a timed automata is manually transcribed into a semantically-equivalent SAL specification), it proves to be unwieldy [DS04b]. 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 particularly well. In particular, 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.

This motivated their development of another theoretical model in which to represent timed transition systems that is more amenable to general-purpose verification environments in which composed state machines can be specified, particularly for verification by k-induction. A *timeout automaton* is another means by which to specify timed transition systems.[3] Timeout automata were motived by the model of system execution used in discrete-event simulation [BI84].

In [DS04b, DS04a], Dutertre and Sorea provide the semantics of a timeout automaton in terms of a transition system. 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* in the transition 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 $x \in T$. Like in in the definition of timed automata, there are two sorts of transitions. The two kinds of transitions are *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 $x \in T$, $\rho'(x) \geq \rho(t)$;

[3] We use 'automata' to refer to syntax, distinct from the semantics for automata.

- there exists $y \in T$ such that $\rho(y) = t$ and $\rho'(y) > \rho(t)$.

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).

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. Very loosely speaking, timeout automata and timed automata are dual with respect to their perspective of time.

Although timeout automata were initially motivated by the desire specify and verify real-time systems in SAL for k-induction verification, they are of significant interest in their own right. Results obtained by Dutertre and Sorea in specifying and verifying the startup algorithm for the Time-Triggered Architecture (TTA) using timeout automata in SAL suggest timeout automata specifications in SAL may be superior to those attainable in Uppaal, a timed automata model checker [LPY97], although a direct comparison is not made [DS04b].[4] By some measures, a timeout automaton has a simpler semantics than does a timed automaton and may allow for a convenient simple embedding of real-time system models in other general-purpose model checkers. In any event, timeout automata are another formalism by which to specify real-time systems that has proved useful in the verification of non-trivial protocols via k-induction. Theoretical comparisons between timeout automata and other real-time formalisms is important future work but not our present goal.

3.1 Synchronizing Timeout Automata (STA)

The following definitions build upon the timeout automata semantics developed in [DS04b, DS04a] and described above. We define a syntax, semantics and composition as follows. We call this the *Synchronizing Timeout Automata* (STA) model. The STA model is motivated by the desire to provide a succinct specification and efficient semantics for systems that synchronize both with respect to events (e.g., message passing) and with respect to time (e.g., time-triggered [Rus99, Kop97] behavior). For example, the train-gate-controller (TGC), is a canonical example of such a real-time system [Alu99]. In [DS04b], it is modeled as the asynchronous composition of timeout automata in which synchronous communication is modeled by the sequential application of edges with the same label. We arguably provide a timeout automata model with a semantics that more closely resembles its intended semantics.

As noted, timeout automata were motivated by Dutertre and Sorea's desire to specify timed systems amenable to k-induction, particularly in SAL. We found their development of timeout automata to be a breakthrough in this respect. Nevertheless, k-induction proofs have a complex-

[4] A meaningful comparison of the formalisms might be difficult given that different tools and formalisms are also used, and indeed, k-induction is not implemented in many other systems.

ity 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. Due to the size of the model, k-induction proofs for $k > 4$ were often infeasible for even a small number of modeled nodes. By allowing both synchronous and asynchronous composition, we can markedly reduce the depth required for proofs by k-induction since in a synchronous composition, multiple edges may be applied simultaneously.

We use the train-gate-controller (TGC) to illustrate this. In [DS04b], a simple safety property is proved using k-induction, for $k = 14$ when the TGC is modeled with the (asynchronous) timeout automata semantics described in Sec. 2. In Sec. 3.2, we prove the same property with $k = 9$ in a synchronous timeout automata model described below. When the optimization in Sec 3.3 is also applied, the property is provable for $k = 5$. This allows significantly more complex systems to be verified via k-induction without having to strengthen the invariant, and it was necessary to complete the verification of the reintegration protocol.

Syntax The definition of a synchronizing timeout automaton (STA) is as follows:

Definition 3 (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 $fV_m = \{f \restriction m \mid f \in fV\}$ 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 $TO = \left\{\alpha \mid \alpha : M \to \mathbb{R}^{0\leq}\right\}$ (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 $TO_m = \left\{\alpha \restriction m \mid \alpha : M \to \mathbb{R}^{0\leq}\right\}$ 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*.

  Then 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 1 *(Timeouts and Timeout Components).* A timeout component represents a portion of the state that updates synchronously. The notion of a timeout components is orthogonal to that of a single state machine in a composition. For example, if one automaton sends another a time-triggered message and the automata synchronize on that message, the state variables of the two automata are in the timeout component triggering the message (see Sec. 3.2, for an example). A synchronous distributed system [Lyn96] can be represented by letting M be a singleton set containing V.

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

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 4 (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 $g_m \restriction n = h_n \restriction m$, and $\beta_m \restriction n = \gamma_n \restriction m$.

The semantics are then as follows.

Definition 5 (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 $\mathcal{S}_{STA}$. A state of $\mathcal{S}_{STA}$[5] 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* $\langle f, \alpha\rangle \xrightarrow{t} \langle g, \beta\rangle$ if and only if $t < \min(\alpha)$, $t' = \min(\beta)$, $g = f$, and $\beta = \alpha$.

To specify *discrete transitions*, the following definitions are of assistance. 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 $h_m = f \restriction m$, $\gamma_m = \alpha \restriction m$, 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. Furthermore, 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 it is an enabled edge in $\langle f, \alpha, t\rangle$, $i_m = g \restriction m$, and $\delta_m = \beta \restriction m$.

Then 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.

Remark 2 *(Minimum Timeouts).* Note that 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 3 *(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

[5] Context distinguishes whether we speak of the states in fV or the constructed states of the transition system.

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 that its current value.

The *nonZeno Property* [AL94] ensures that an infinite number of transitions are not enabled within a finite interval of time. This property should also be satisfied if a specification is to be implementable.

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 6 (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 $\langle f \restriction V^1, \alpha \restriction M^1 \rangle \in I^1$, and $\langle f \restriction V^2, \alpha \restriction M^2 \rangle \in I^2$.

Remark 4 *(Compositional Specifications).* The specification of a STA is somewhat orthogonal to 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.

3.2 STA Model of the Train-Gate-Controller

The train-gate-controller (TGC) is a canonical example of a real-time system. It models the interaction of a train, a gate, and a gate controller at a railroad crossing. (For simplicity, 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 more than 2 and no more than 5 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 1 and no more than 2 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 a timed automaton in Fig. 1, by composing timed state machines, as presented in [Alu99]. 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 verticies denote the time by which the state must be left. Clock constraints at the edges constrain when the edge in 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.

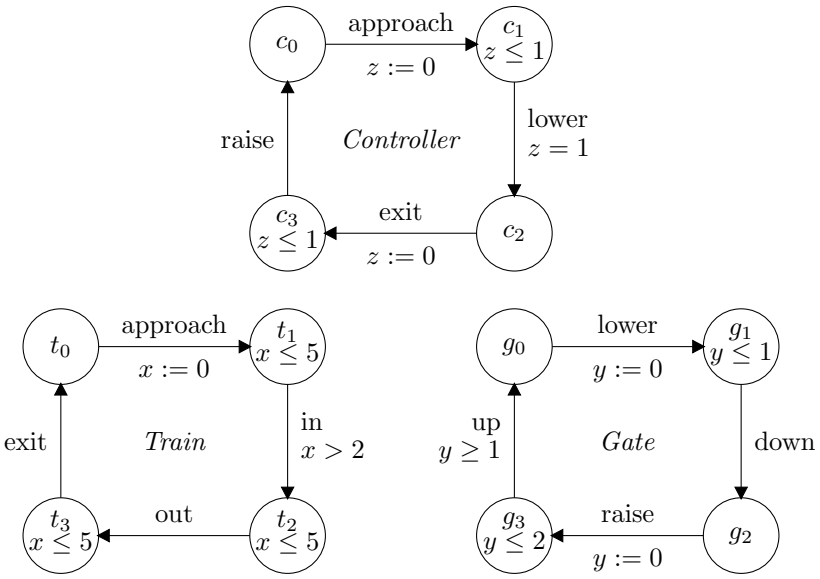


Figure 1: The Train-Gate-Controller

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_2, respectively, simultaneously. The representation in Fig. 1 is based on a timed automata model of the TGC. A full description with a formal syntax and semantics of the TGC modeled as a timed automaton can be found in [Alu99].

STA Model of the TGC Following Def. 3, the TGC is modeled as a STA $\langle V, M, I, E \rangle$, informally described as follows.

- 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 *(Timeout Vs. Timed Automata).* Note that unlike in the timed automata formalization, clocks are not reset. Timeouts continue to increase indefinitely, but they are required to satisfy the constrains 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 $t + 5$.

TGC Semantics in SAL The specification of the TGC in SAL modifies the model developed by Dutertre and Sorea in [DS04b], and is presented in Appendix A. 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. A specialized language is not required to describe these semantics. Embedding the semantics provides a great deal of flexibility; in particular, it allows the user to optimize the semantics; we describe one such possibility in Sec. 3.3.

The basic building block of a SAL specification is a *module* containing global, input, output, and local variables. In a module, transitions are specified by guarded commands in which local and output variables may be updated. Modules may be either synchronously or asynchronously composed, and they communicate via shared variables.

To implement the semantics, 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 also has an output timeout variable. Additionally, there is a module that specifies the global clock, which outputs the current time. The other modules have an input variable to read the current time. 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. The composition of the train, gate, and controller is asynchronously composed with the clock module. The clock module has a single transition that updates the clock when the current time is less than all the timeouts, and it is updated to the minimum of the timeouts.

Verification The following is a typical safety property one might wish to prove about the TGC: if the train is in the railroad crossing ($s_t = t_2$), then the gate is down ($s_g = g_2$). In model of the TGC employing asynchronous timeout automata semantics described in Sec. 3, the property can be proved by k-induction when $k = 14$. With the STA semantics defined, the property is proved when $k = 9$.

3.3 Clockless STA Semantics

We describe an optimization to the STA semantics provided in Def. 5 in which we describe how to remove the global clock from the semantics. 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 [DS04b]. After applying the optimization described here, this depth is reduced to $k = 5$. In Appendix B is a STA model of the TGC after applying the optimization. This optimization was essential to complete the verification of the reintegration protocol.

In a timeout automaton, the essential 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 state invariant via k-induction. If the number of timeouts is large, then state transitions overshadow clock transitions.

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 7 (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 $h_m = f \restriction m$, $\gamma_m = \alpha \restriction m$, 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 it is an enabled edge in $\langle f, \alpha\rangle$, $i_m = g \restriction m$, and $\delta_m = \beta \restriction m$.

Then 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 state invariants are true under both clockless semantics and the semantics in Def. 5.

Proposition 1 (Clockless Simulation). *Fix a STA* $\langle V, M, I, E\rangle$*. Let its semantics from Def. 5 be the transition system* $\mathcal{S}_{STA}$*, and let its clockless semantics be the transition system* $\mathcal{S}_{STA}^{\neg cl}$*. Let* P *be some predicate defined over the states of* $\mathcal{S}_{STA}$ *that does not take the clock variable* t *as an argument. Then* P *holds for all reachable states in* $\mathcal{S}_{STA}$ *if and only if it holds for all reachable states in* $\mathcal{S}_{STA}^{\neg cl}$*.*

Proof. By induction over the states of $\mathcal{S}_{STA}$ and $\mathcal{S}_{STA}^{\neg cl}$, the state $\langle f, \alpha\rangle$ is reachable in $\mathcal{S}_{STA}^{\neg cl}$, if and only if either $\langle f, \alpha, 0\rangle$ or $\langle f, \alpha, \min(\alpha)\rangle$ is reachable in $\mathcal{S}_{STA}^{cl}$. □

Example 1 *(TGC with Clockless STA Semantics).* Removing the clock is straightforward. In SAL, this essentially amounts to removing the module that specifies the global clock, as described in Sec. 3.2. The specifications of the train, gate and controller must then be modified slightly: rather than comparing timeouts against the global clock to determine whether an edge is enabled, timeouts are directly compared with one another. The full SAL specification is in Appendix B.

Remark 6 *(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 Sec. 3.2 from 14 under the original timeout automata semantics to $k = 9$ in the STA semantics to $k = 5$ in the clockless STA semantics. The benefit of removing the clock is particularly pronounced in the TGC model because there are only three timeouts. Thus, the number of transitions in a path devoted to updating the clock is relatively high. In a system with many more timeouts, this effect is less pronounced. For example, applying these modifications to the timeout automata model of the Fischer Mutual Exclusion Protocol described in [DS04b] for a large number of processes would yield more modest results.

4 The Reintegration Protocol

The protocol described here abstracts the reintegration protocol being designed for the latest SPIDER prototype [TPMM05]. 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 being that the reintegrator simply listens but does not broadcast messages as in the full distributed diagnosis protocol) [TPMM05]. The Distributed Diagnosis Protocol has been formally verified in PVS [ORSvH95], and the protocol and its verification is described in [MGPM04].

From a pragmatic standpoint, resynchronization during reintegration is the most complex portion of the protocol and stood to benefit the most from formal analysis. Once reintegration is achieved, the remainder of

the protocol can be modeled as being synchronous, 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.

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*, a fault-tolerant protocol in which nodes synchronize their local clocks that may have drifted (this protocol and its formal verification are also described in [MGPM04]). 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.

The reintegrator is designed to tolerate the full range of faulty behaviors, including Byzantine faults [LSP82], 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 [PMMG04].

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 reseting. 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 its diagnostic data to allow nodes a chance to reintegrate. In the current SPIDER prototype under development [TPMM05], 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.

4.1 System Assumptions

Before describing the behavior of the protocol, a preliminary understanding of the system assumptions is required. These properties are stated in

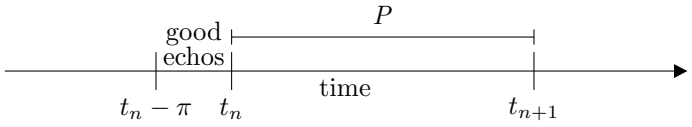


Figure 2: The Frame Property

terms of *accusations* made by the reintegrator. The reintegrator accuses a node when it believes the communication from the node is inappropriate (i.e., 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 [MGPM04] and the high-level scheduling of the protocols. It is illustrated in Fig. 2.

Definition 8 (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 nth *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 $(t_n - \pi, t_n)$,[6] 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 9 (Majority Property). Of the nodes that have not been accused by the reintegrator during the entire protocol, the majority are operational.

4.2 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 Fig. 3. We itemize the global and local state variables of the modes, and then we describe the behavior of the protocol during each mode.

4.2.1 State Description

State Variables The following state variables of the reintegrator determine the state of the reintegrator during the execution of the protocol.

[6] In this model, we include communication error in the skew.

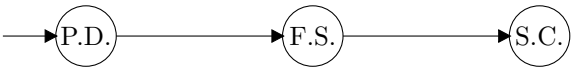


Figure 3: State Machine Model of the Protocol Mode Control

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.

State Initialization 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;
```

4.2.2 Protocol Behavior

Preliminary Diagnosis 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 + π;
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 := frame_synch;
```

Frame Synchronization 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 able 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 < π 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;
```

Synchronization 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 (Def. 9) 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 ≤ 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;
```

5 Modeling the Protocol in SAL

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 our effort to do the same for the TGC's STA model, as described in Sec. 3.2 and Ex. 1. The full model can be found both in Appendix C and on-line at [Pik05] for download.

5.1 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 it is actually in).

This model allows for 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 (this is provable by k-induction). This ensures the issuing of echo messages are always future events observable by the timeout model of the reintegrator.

```
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

5.2 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 Sec. 4, 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.

5.2.1 Operational Nodes

To model the operational nodes, we begin by defining a module that keeps track of the resynchronization frames, as presented in Fig. 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 Sec. 4.1. 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.

The operational nodes themselves are specified by an `op_node` module, parameterized by the indices of operational nodes, presented in Fig. 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 (Def. 8). 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

```
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

```
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

reality, the clock of an operational node would not drift so violently.

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 issues echos in the next frame. An invariant is proved about the model that demonstrates that this does not occur.

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.

5.2.2 Faulty Nodes

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

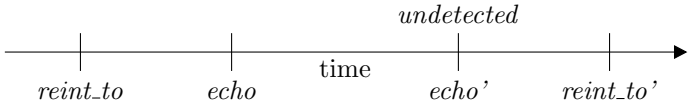


Figure 7: The Reintegrator TA Misses Echo Messages

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 Fig. 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.

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 Fig. 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 behav-

```
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

ior 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 (Def. 9), and the other echos are beyond the observation window of the reintegrator.

The precondition for the transition to update the echos is similar to that described for the frame synchronization module described in Sec. 5.2.1. 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.

5.3 The Reintegrator

Each of the three modes of the reintegration protocol is specified as a separate module. Additionally, another module handles mode control.

5.3.1 Mode Control

Each of the three modes has a binary control signal to determine whether it is active. Only one mode may be active at any time. The module specified in Fig. 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 demarcate locations in the execution of the protocol where certain invariants are supposed to be reached. An invariant guaranteed upon the completion of a mode serves as an assumption in demonstrating

```
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

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.

5.3.2 Base Modes

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 reseting the *seen* variable at the conclusion of each mode.

The following paragraphs overview the models of each mode's execution.

Preliminary Diagnosis In the preliminary diagnosis mode, there are two principle transitions, as shown in Fig. 10. The first transition models the behavior during the mode, and the second models the exiting of the

```
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

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 (Sec. 5.2.1). 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.

Frame Synchronization 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 Fig. 11, we define the relation `none_in_pi?` that determine whether or not 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 Sec. 5.2.1). When the

```
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' = deactive;
    reint_to' = reint_to + pi;
                ⋮
```

Figure 11: Frame Synchronization Module

relation does hold, the reintegrator's timeout is simply updated to be π units of time greater than its current value.

Synchronization Capture In the last mode, shown in Fig. 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 –

```
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' = deactive;
                ⋮
  ]
```

Figure 12: Synchronization Capture Module

that is, when

$$\texttt{sc\_seen\_total(sc\_op\_seen, sc\_bad\_seen)} > \texttt{not\_accd(op\_accs, bad\_accs)}/2\,.$$

This also marks the termination of the reintegration protocol.

5.3.3 Composing The Modules

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

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

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:

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

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

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

6 Verifying the Protocol

There are two main theorems to prove. First, we wish to show that the reintegrator accuses no operational nodes during the execution of the reintegration protocol. Second, we wish to show that the reintegrator has successfully resynchronized with the operational nodes upon completion of the reintegration protocol.

Theorem 1 (No Operational Accusations). *For all operational nodes i, $accs[i]$ does not hold during the reintegration protocol.*

Theorem 2 (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. As mentioned in Sect. 5.3.1, our general 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 in [DS04b, DS04a] and inspired by abstraction techniques described in [Rus00, MP94]. However, in [DS04b, DS04a], the abstraction was manually constructed after the fact to aid in the verification. It's construction seemed to require a great deal of understanding about the protocol before verifying it. Furthermore, it is a predicate abstraction: a state machine is constructed, the states of which are predicates over the protocol model. These predicates roughly correspond to the postconditions we verify. While the predicate abstraction technique is more powerful, its construction is more complicated; the mode abstraction is simply adopted from the protocol specification.

In this respect, a proof by k-induction can be seen to fall somewhere between an *inductive invariant* approach and a *clock function* approach used in verifying total correctness of transformational programs via mechanical theorem-proving [RM04]. The former approach amounts to induction over a transition system, while the latter requires one to show that from any state satisfying the precondition, the program halts and

the postcondition is satisfied after some fixed number of transitions from that state.

The proof of Thm 1 requires showing that no accusations are issued in any of the modes; accusations are not issued in the synchronization capture mode, so we need be concerned with only 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 2 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 1). 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.

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) [MMT02], and this architecture with six nodes is designed to tolerate up to two simultaneous Byzantine 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 Def. 9, the Majority Property). The proofs took on the order of seconds (and occasionally minutes) to complete on a machine 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.

Invariant k-Induction Proofs 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 $k \leq 4$.

Lemmas From Failed Proof Attempts 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 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 upon the user to interpret how the constraints lead to a counterexample.

Clique Avoidance *Clique avoidance* is the property that there exists exactly one operational clique in the system [Rus01, BP00]. 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 loose 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 [TPMM05], unlike, e.g., TTP/C [BP00, Pfe03]. However, the architecture is designed with the intent that if during the course of its operation the MFA (Sec. 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 Thm. 1, 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 [TPMM05]. Although the essential protocols that execute during startup and normal operation have been formally verified individually [MGPM04], there does not yet exist a cohesive argument to demonstrate formally that clique avoidance is preserved.

7 Conclusion

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 that has recently been proposed and successfully used now in two industrial-scale verification projects (this and the work presented in [DS04a]). Furthermore, we have described a means by which event-triggered behavior can be modeled as time-triggered behavior. This application demonstrates that both appropriate formalisms and appropriate abstractions of the physical world [PMMG04] are necessary if non-trivial problems are to be addressed by formal methods. The essential means by which we achieved our results was 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. Reintegration 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. All that is of concern are what messages the reintegrator receives from the other nodes in the system. 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 did not reveal any flaws in the protocol. Nevertheless, it was 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 well-studied fault-tolerant distributed protocols [Lyn96]. As well, the formal verification not only demonstrated the correctness of reintegration but it also strongly suggests that clique avoidance is preserved.

Nevertheless, the formal verification did reveal that a more general assumption can be used to prove the correctness of this mode: 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 Def. 9, 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.

As called for in [DS04a], future work includes theoretical studies comparing STA and other real-time formalisms. Other techniques for opti-

mizing STA for k-induction would be useful; in particular, techniques for k-induction over parameterized systems would be of much practical value. Also of value would be direct comparisons 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 [HHWT97], Kronos [DOTY95], Uppaal [LPY97], etc.).

We note too that we have concerned ourselves with only timeout automata here. In [DS04a, DS04b], Dutertre and Sorea develop a more complex real-time formalism they call *calendar automata* in which timeouts are also associated with the delay between when a message is sent and received by communicating processes. We did not model communication delays in our model of the reintegration protocol. The relationship between timeout automata, synchronizing timeout automata, and calendar automata should be examined in more detail.

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 was to specify properties we knew should 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. An alternative strategy that might be fruitful would be to construct some finite abstraction of a transition system with the property that if the finite abstraction contains a deadlock, then so does the infinite-state system. SAL contains a deadlock checker for finite state systems that could be then applied. This approach was not pursued in this work, however.

References

[AL94] 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.

[Alu99] 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`.

[BGL+00] Saddek Bensalem, Vijay Ganesh, Yassine Lakhnech, César Mu noz, 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/`.

[BI84] Jerry Banks and John S. Carson II. *Discrete-Event Simulation*. Prentice-Hall, 1984.

[BP00] Gnther Bauer and Michael Paulitsch. An investigation of membership and clique avoidance in ttp/c. *19th IEEE Symposium on Reliable Distributed Systems, 16th - 18th October 2000, Nrnberg, Germany*, October 2000.

[CCO+04] Roberto Cavada, Alessandro Cimatti, Emanuele Olivetti, Gavin Keighren, Marco Pistore, and Marco Roveri. *NuSMV 2.2 User Manual*. IRST, Via Sommarive 18, 38055 Povo (Trento) Italy, 2004. Available at http://nusmv.irst.itc.it/NuSMV/userman/index-v2.html.

[dMOR+04] Leonardo de Moura, Sam Owre, Harald Ruess, John Rushby, and N. Shankar. The ICS decision procedures for embedded deduction, July 2004.

[DOTY95] Conrado Daws, Alfredo Olivero, Stavros Tripakis, and Sergio Yovine. The tool KRONOS. In *Hybrid Systems*, pages 208–219, 1995.

[DS04a] 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.

[DS04b] 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.

[ES03] Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Computer Science*, 89(4), 2003.

[HD92] Kenneth Hoyme and Kevin Driscoll. SAFEbus. In *11th AIAA/IEEE Digital Avionics Systems Conference (DASC)*, pages 68–73, October 1992.

[HHWT97] 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.

[HS96] Klaus Havelund and Natarajan Shankar. Experiements in theorem proving and model checking for protocol verification. In *Proceedings of Formal Methods Europe FME'96*, Lecture Notes in Computer Science. Springer, 1996.

[Kop94] Hermann Kopetz. TTP—a protocol for fault-tolerant real-time systems. *IEEE Computer*, 27(1):14–23, January 1994.

[Kop97] Hermann Kopetz. *Real-Time Systems*. Kluwer Academic Publishers, 1997.

[LH02] G. Leen and D. Heffernan. Expanding automotive electronic systems. *IEEE Computer*, 0018-9162/02:88–93, Jan 2002.

[LPY97] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *STTT*, 1(1-2):134–152, 1997.

[LSP82] 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.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[MGPM04] 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://techreports.larc.nasa.gov/ltrs/PDF/2004/mtg/NASA-2004-jcfmats-pm.pdf.

[MMT02] 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.

[MP94] 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.

[NAS04] NASA Formal Methods Group. SPIDER homepage. Website, 2004. Available at http://shemesh.larc.nasa.gov/fm/spider/.

[ORSvH95] 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.

[Pfe03] 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.

[Pik05] Lee Pike. SPIDER reintegration protocol SAL files. Website, 2005. Available at http://shemesh.larc.nasa.gov/fm/spider/reint_sal/.

[PMMG04] 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://techreports.larc.nasa.gov/ltrs/PDF/2004/mtg/NASA-2004-17tphol-lsp.pdf.

[PMT04] 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://techreports.larc.nasa.gov/ltrs/PDF/2004/tm/NASA-2004-tm213278.pdf.

[RM04] Sandip Ray and J. Strother Moore. Proof styles in operational semantics. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 67–81, 2004.

[Rus99] John Rushby. Systematic formal verification for fault-tolerant time-triggered algorithms. *IEEE Transactions on Software Engineering*, 25(5):651–660, September 1999. Available at http://www.csl.sri.com/papers/tse99/.

[Rus00] 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.

[Rus01] 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.

[Rus02] 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.

[SRI04] SRI International. Symbolic analysis laboratory SAL, 2004. Available at http://sal.csl.sri.com/.

[SSS00] 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/.

[TPMM05] Wilfredo Torres-Pomales, Mahyar R. Malekpour, and Paul Miner. ROBUS-2: A fault-tolerant broadcast communication system. Technical report, NASA Langley Research Center, 2005.

A STA Model of the TGC in SAL

```
% ------------------------------------------------------------
% Lee Pike
% NASA Langley Formal Methods Group
% lee.s.pike@nasa.gov
%
% SAL 2.3
%
% Adapted from the TGC model in SAL by B. Dutertre and
% M.Sorea, in "Timed Systems in SAL," Technical Report
% SRI-SDL-04-03, July 2004.
% ------------------------------------------------------------
sta_tgc: CONTEXT =
BEGIN

SIGNAL              : TYPE = {approach, exit, lower,
                              raise, null};
TIME                : TYPE = REAL;
N                   : NATURAL = 3;
INDEX               : TYPE = [1..N];
TIMEOUT_ARRAY       : TYPE = ARRAY INDEX OF TIME;

T_STATE: TYPE = {t0, t1, t2, t3};
G_STATE: TYPE = {g0, g1, g2, g3};
C_STATE: TYPE = {c0, c1, c2, c3};

to_min(t1: TIME, t2: TIME, t3: TIME): TIME =
  min(t1, min(t2, t3));

%------------------------------------------------------------
% Clock module: makes time elapse up to the next timeout
%------------------------------------------------------------
clock: MODULE =
 BEGIN
  INPUT
    t_timeout       : TIME,
    c_timeout       : TIME,
    g_timeout       : TIME
  OUTPUT time: TIME
  INITIALIZATION time = 0
    TRANSITION
      [ time_elapses:
       time < to_min(t_timeout, c_timeout, g_timeout)
       -->
         time' = to_min(t_timeout, c_timeout, g_timeout)
      ]
 END;

%------------------------------------------------------------
```

```
% Train module
%-----------------------------------------------------------
train: MODULE =
BEGIN
 INPUT
    time            : TIME,
    c_state         : C_STATE
 OUTPUT
    t_timeout       : TIME,
    msg1            : SIGNAL
  LOCAL
    reset           : TIME,
    t_state         : T_STATE
 INITIALIZATION
    t_state = t0;
    msg1 = null;
 TRANSITION
    [ t0_t1:
         t_state = t0
     AND t_timeout = time
     AND c_state = c0
     -->
        t_state' = t1;
        msg1' = approach;
        reset' = time + 5;
        t_timeout' IN { x: TIME |     time + 2 < x
                                  AND x <= time + 5}
   [] t1_t2:
         t_state = t1
     AND t_timeout = time
     -->
        msg1' = null;
        t_state' = t2;
        t_timeout' IN { x: TIME |     time < x
                                  AND x <= reset}
   [] t2_t3:
         t_state = t2
     AND t_timeout = time
     -->
        msg1' = null;
        t_state' = t3;
        t_timeout' IN { x: TIME |     time < x
                                  AND x <= reset}
   [] t3_t0:
         t_state = t3
     AND t_timeout = time
     AND c_state = c2
     -->
        t_state' = t0;
        msg1' = exit;
```

```
        t_timeout' IN { x: TIME | time < x}
   []
     ELSE -->
    ]
 END;

%-------------------------------------------------------
% GATE module
%-------------------------------------------------------
gate: MODULE =
BEGIN
 INPUT
    time            : TIME,
    c_timeout       : TIME,
    msg2            : SIGNAL
 OUTPUT
    g_timeout       : TIME,
    g_state         : G_STATE
 INITIALIZATION
    g_state = g0;
 TRANSITION
    [ g0_g1:
         g_state = g0
     AND msg2' = lower
     AND c_timeout = time
     -->
        g_state' = g1;
        g_timeout' IN { x: TIME |     time < x
                                  AND x <= time + 1}
   [] g1_g2:
         g_state = g1
     AND g_timeout = time
     -->
        g_state' = g2;
        g_timeout' IN { x: TIME | time < x }
   [] g2_g3:
         g_state = g2
     AND msg2' = raise
     AND c_timeout = time
     -->
        g_state' = g3;
        g_timeout' IN { x: TIME |     time + 1 <= x
                                  AND x <= time + 2}
   [] g3_g0:
         g_state = g3
     AND g_timeout = time
     -->
        g_state' = g0;
        g_timeout' IN { x: TIME | time < x}
   []
```

```
    ELSE -->
   ]
END;

%------------------------------------------------------------
% Controller module
%------------------------------------------------------------
controller : MODULE =
BEGIN
 INPUT
   time            : TIME,
   t_timeout       : TIME,
   msg1            : SIGNAL,
   g_state         : G_STATE
 OUTPUT
   c_timeout       : TIME,
   msg2            : SIGNAL,
   c_state         : C_STATE
 INITIALIZATION
   c_state = c0;
   msg2 = null;
 TRANSITION
   [ c0_c1:
        c_state = c0
    AND t_timeout = time
    AND msg1' = approach
    -->
       c_state' = c1;
       c_timeout' = time + 1
  [] c1_c2:
        c_state = c1
    AND c_timeout = time
    AND g_state = g0
-->
       c_state' = c2;
       msg2' = lower;
       c_timeout' IN { x: TIME | time < x }
  [] c2_c3:
        c_state = c2
    AND msg1' = exit
    AND t_timeout = time
    -->
       c_state' = c3;
       c_timeout' IN { x: TIME |     time < x
                                 AND x <= time + 1}
  [] c3_c0:
        c_state = c3
    AND c_timeout = time
    AND g_state = g2
    -->
```

```
        c_state' = c0;
        msg2' = raise;
        c_timeout' IN { x: TIME | time < x}
   []
     ELSE -->
      ]
   END;

tgc: MODULE = train || gate || controller;

system: MODULE = clock [] tgc;

%------------------------------------------------------------
% properties
%------------------------------------------------------------
% proved d9
safe: LEMMA system |- G(t_state = t2 => g_state = g2);

%------------------------------------------------------------
% liveness checks
%------------------------------------------------------------
tstate2: LEMMA system |- G(t_state /= t2);
gstate2: LEMMA system |- G(g_state /= g2);
cstate2: LEMMA system |- G(c_state /= c2);

tstate3: LEMMA system |- G(t_state /= t3);
gstate3: LEMMA system |- G(g_state /= g3);
cstate3: LEMMA system |- G(c_state /= c3);

END
```

B STA Model of the TGC with Clockless Semantics in SAL

```
% ------------------------------------------------------------
% Lee Pike
% NASA Langley Formal Methods Group
% lee.s.pike@nasa.gov
%
% SAL 2.3
%
% Adapted from the TGC model in SAL by B. Dutertre and
% M.Sorea, in "Timed Systems in SAL," Technical Report
% SRI-SDL-04-03, July 2004.
% ------------------------------------------------------------
sta_tgc_clockless: CONTEXT =
BEGIN

SIGNAL              : TYPE = {approach, exit, lower,
                              raise, null};
TIME                : TYPE = REAL;
N                   : NATURAL = 3;
INDEX               : TYPE = [1..N];
TIMEOUT_ARRAY       : TYPE = ARRAY INDEX OF TIME;

T_STATE: TYPE = {t0, t1, t2, t3};
G_STATE: TYPE = {g0, g1, g2, g3};
C_STATE: TYPE = {c0, c1, c2, c3};

to_min(t1: TIME, t2: TIME, t3: TIME): TIME =
  min(t1, min(t2, t3));

%-----------------------------------------------------------
% Train module
%-----------------------------------------------------------
train: MODULE =
BEGIN
 INPUT
    c_timeout       : TIME,
    g_timeout       : TIME,
    c_state         : C_STATE
 OUTPUT
    t_timeout       : TIME,
    msg1            : SIGNAL
 LOCAL
    reset           : TIME,
    t_state         : T_STATE
 INITIALIZATION
    t_state = t0;
    msg1 = null;
```

```
 TRANSITION
    [ t0_t1:
         t_state = t0
     AND t_timeout = to_min(t_timeout, c_timeout, g_timeout)
     AND c_state = c0
     -->
        t_state' = t1;
        msg1' = approach;
        reset' = t_timeout + 5;
        t_timeout' IN { x: TIME |     t_timeout + 2 < x
                                  AND x <= t_timeout + 5}
   [] t1_t2:
         t_state = t1
     AND t_timeout = to_min(t_timeout, c_timeout, g_timeout)
     -->
        msg1' = null;
        t_state' = t2;
        t_timeout' IN { x: TIME |     t_timeout < x
                                  AND x <= reset}
   [] t2_t3:
         t_state = t2
     AND t_timeout = to_min(t_timeout, c_timeout, g_timeout)
     -->
        msg1' = null;
        t_state' = t3;
        t_timeout' IN { x: TIME |     t_timeout < x
                                  AND x <= reset}
   [] t3_t0:
         t_state = t3
     AND t_timeout = to_min(t_timeout, c_timeout, g_timeout)
     AND c_state = c2
     -->
        t_state' = t0;
        msg1' = exit;
        t_timeout' IN { x: TIME | t_timeout < x}
   []
     ELSE -->
    ]
 END;

%------------------------------------------------------------
% GATE module
%------------------------------------------------------------
gate: MODULE =
BEGIN
 INPUT
    c_timeout       : TIME,
    t_timeout       : TIME,
    msg2            : SIGNAL
 OUTPUT
```

```
   g_timeout       : TIME,
   g_state         : G_STATE
 INITIALIZATION
   g_state = g0;
 TRANSITION
   [ g0_g1:
        g_state = g0
    AND msg2' = lower
    AND c_timeout = to_min(t_timeout, c_timeout, g_timeout)
    -->
       g_state' = g1;
       g_timeout' IN { x: TIME |     c_timeout < x
                                  AND x <= c_timeout + 1}
  [] g1_g2:
        g_state = g1
    AND g_timeout = to_min(t_timeout, c_timeout, g_timeout)
    -->
       g_state' = g2;
       g_timeout' IN { x: TIME | g_timeout < x }
  [] g2_g3:
        g_state = g2
    AND msg2' = raise
    AND c_timeout = to_min(t_timeout, c_timeout, g_timeout)
    -->
       g_state' = g3;
       g_timeout' IN { x: TIME |     c_timeout + 1 <= x
                                  AND x <= c_timeout + 2}
  [] g3_g0:
        g_state = g3
    AND g_timeout = to_min(t_timeout, c_timeout, g_timeout)
    -->
       g_state' = g0;
       g_timeout' IN { x: TIME | g_timeout < x}
  []
    ELSE -->
   ]
 END;

%------------------------------------------------------------
% Controller module
%------------------------------------------------------------
controller : MODULE =
BEGIN
 INPUT
   t_timeout       : TIME,
   g_timeout       : TIME,
   msg1            : SIGNAL,
   g_state         : G_STATE
 OUTPUT
   c_timeout       : TIME,
```

```
   msg2            : SIGNAL,
   c_state         : C_STATE
INITIALIZATION
   c_state = c0;
   msg2 = raise;
TRANSITION
   [ c0_c1:
        c_state = c0
    AND t_timeout = to_min(t_timeout, c_timeout, g_timeout)
    AND msg1' = approach
    -->
       c_state' = c1;
       c_timeout' = t_timeout + 1
  [] c1_c2:
       c_state = c1
    AND c_timeout = to_min(t_timeout, c_timeout, g_timeout)
    AND g_state = g0
-->
       c_state' = c2;
       msg2' = lower;
       c_timeout' IN { x: TIME | c_timeout < x }
  [] c2_c3:
       c_state = c2
    AND msg1' = exit
    AND t_timeout = to_min(t_timeout, c_timeout, g_timeout)
    -->
       c_state' = c3;
       c_timeout' IN { x: TIME |     t_timeout < x
                                 AND x <= t_timeout + 1}
  [] c3_c0:
       c_state = c3
    AND c_timeout = to_min(t_timeout, c_timeout, g_timeout)
    AND g_state = g2
    -->
       c_state' = c0;
       msg2' = raise;
       c_timeout' IN { x: TIME | c_timeout < x}
  []
    ELSE -->
     ]
  END;

system: MODULE = train || gate || controller;

%-----------------------------------------------------------
% properties
%-----------------------------------------------------------
% proved d5
safe: LEMMA system |- G(t_state = t2 => g_state = g2);
```

```
%-----------------------------------------------------------
% liveness checks
%-----------------------------------------------------------
tstate2: LEMMA system |- G(t_state /= t2);
gstate2: LEMMA system |- G(g_state /= g2);
cstate2: LEMMA system |- G(c_state /= c2);

tstate3: LEMMA system |- G(t_state /= t3);
gstate3: LEMMA system |- G(g_state /= g3);
cstate3: LEMMA system |- G(c_state /= c3);

END
```

C STA Model of the Reintegration Protocol in SAL

```
% ------------------------------------------------------------
% Lee Pike
% NASA Langley Formal Methods Group
% lee.s.pike@nasa.gov
%
% Compatible with SAL 2.3 & SAL 2.4
% ------------------------------------------------------------
reint: CONTEXT =
BEGIN

% --------------------------TYPES AND CONSTANTS -------------
% The nonnegative reals.
TIME            : TYPE = {x: REAL | 0 <= x};

MODES           : TYPE = {pd_mode, fs_mode, sc_mode};
CNTRL           : TYPE = {active, deactive};

% -------------------------------------
% IF THE NUMBER OF NODES ARE CHANGED, UPDATE THESE TYPES AND
% CONSTANTS APPROPRIATELY.
% The user must ensure that there are enough operational
% nodes to ensure the majority property always holds
% (this requires the majority of nodes to be operational).

% Number of abstract operational nodes.
op_total        : NATURAL = 2;
% Number of abstract bad nodes.
bad_total       : NATURAL = 1;
% Total number of abstract nodes.
total           : NATURAL = op_total + bad_total;
% The set of all abstract nodes.
ALL_IDS         : TYPE = {x: [1..total] | x=1 OR x=2 OR x=3};
ALL_CNT         : TYPE = {x: [0..total] |    x=0 OR x=1
                                          OR x=2 OR x=3};
% The set of abstract operational nodes.
OP_IDS          : TYPE = {x: [1..op_total] | x=1 OR x=2};
% Where the indexing of bad nodes starts.
b_st            : NATURAL = op_total + 1;
% The set of abstract bad nodes.
BAD_IDS         : TYPE = {x: [b_st..total] | x=3};
% -------------------------------------

% The size of the seen variable in preliminary diagnosis.
pd_see_top      : NATURAL = 3;
% The set of possible seen vals in preliminary diagnosis.
PD_SEEN         : TYPE = {x: [0..pd_see_top]
```

```
                            |    x=0 OR x=1 OR x=2
                              OR x=pd_see_top};
% The set of times at which abtract operational nodes send
% echos.
OP_ECHOS        : TYPE = ARRAY OP_IDS OF TIME;
% Each bad node has an array of times at which it echos.
% The array is as big as the number of times that echos
% can be seen in a mode.
BAD_ECHO_ARRAY  : TYPE = ARRAY [1..pd_see_top] OF TIME;
% The set of echo arrays for the abstract bad nodes.
BAD_ECHOS       : TYPE = ARRAY BAD_IDS OF BAD_ECHO_ARRAY;
% The reintegrator's set of accusations of operational nodes.
OP_ACCS         : TYPE = ARRAY OP_IDS OF BOOLEAN;
% The reintegrator's set of accusations of bad nodes.
BAD_ACCS        : TYPE = ARRAY BAD_IDS OF BOOLEAN;
% The reintegrator's record of how many times an operational
% node has been seen in preliminary diagnosis.
PD_OP_SEEN      : TYPE = ARRAY OP_IDS OF PD_SEEN;
% The reintegrator's record of how many times an bad node
% has been seen in preliminary diagnosis.
PD_BAD_SEEN     : TYPE = ARRAY BAD_IDS OF PD_SEEN;
% maximum skew between operational nodes
pi              : {t: TIME |  0 < t};
% Length of synchronization frame.  Constrained by skew.
P               : {t: TIME | t > pi*(bad_total + 2)};
% ------------------------------------------------------------



% ------------------FUNCTIONS--------------------------------
% Is the node id of an operational node?
operational?(i: ALL_IDS): BOOLEAN = i <= op_total;

% --------------------------------------
% Functions for monitoring echos from faulty nodes in the
% preliminary diagnosis mode.
pd_bad_see_rec(bad_ecs: BAD_ECHO_ARRAY, ind: PD_SEEN,
               start: TIME, ending: TIME, seen: PD_SEEN)
                 : PD_SEEN =
  IF ind=0 THEN seen
  ELSIF (bad_ecs[ind] > start AND bad_ecs[ind] <= ending)
    THEN pd_bad_see_rec(bad_ecs, ind-1, start, ending, seen+1)
    ELSE pd_bad_see_rec(bad_ecs, ind-1, start, ending, seen) ENDIF;

pd_bad_see(seen: PD_SEEN, bad_ecs: BAD_ECHO_ARRAY,
           start: TIME, ending: TIME): PD_SEEN =
  IF      pd_bad_see_rec(bad_ecs, pd_see_top, start, ending, 0)
        + seen
     >= pd_see_top
  THEN pd_see_top
```

```
  ELSE   pd_bad_see_rec(bad_ecs, pd_see_top, start, ending, 0)
       + seen
  ENDIF;

pd_bad_echos(seen: PD_SEEN, bad_ec: BAD_ECHO_ARRAY,
             start: TIME, ending: TIME): PD_SEEN =
  pd_bad_see(seen, bad_ec, start, ending);
% ------------------------------------

% Functions for monitoring echos from faulty nodes in the
% preliminary diagnosis mode.
pd_op_echos(seen: PD_SEEN, op_ec: TIME,
            start: TIME, ending: TIME): PD_SEEN =
  IF (op_ec > start AND op_ec <= ending AND seen < pd_see_top)
  THEN seen+1 ELSE seen ENDIF;

% ------------------------------------
% Functions for finding the last valid echo in the
% frame synchronization mode transitions.

% No echos from eligible nodes within pi ticks.
none_in_pi?(reint_to: TIME, op_echos: OP_ECHOS,
            bad_echos: BAD_ECHOS, fs_op_seen: OP_ACCS,
            fs_bad_seen: BAD_ACCS, op_accs: OP_ACCS,
            bad_accs: BAD_ACCS): BOOLEAN =
      (FORALL (i: OP_IDS):
            (    op_echos[i] > reint_to
             AND (NOT op_accs[i]) AND (NOT fs_op_seen[i]))
         => op_echos[i] > pi+reint_to)
  AND (FORALL (i: BAD_IDS):
            ((NOT bad_accs[i]) AND (NOT fs_bad_seen[i]))
         => bad_echos[i][1] > pi+reint_to);

% Defined as a predicate rather than a recursive function;
% see Dutertre and Sorea's tech report for an explanation.
% True at the time of the last eligible echo in pi ticks.
last_in_pi?(t: TIME, reint_to: TIME, op_echos: OP_ECHOS,
            bad_echos: BAD_ECHOS, fs_op_seen: OP_ACCS,
            fs_bad_seen: BAD_ACCS, op_accs: OP_ACCS,
            bad_accs: BAD_ACCS, frame_to: TIME): BOOLEAN =
      (FORALL (i: OP_IDS):
            (    op_echos[i] > reint_to
             AND op_echos[i] <= pi+reint_to
             AND (NOT op_accs[i]) AND (NOT fs_op_seen[i]))
         => t >= op_echos[i])
  AND (FORALL (i: BAD_IDS):
            (    bad_echos[i][1] <= pi+reint_to
             AND (NOT bad_accs[i]) AND (NOT fs_bad_seen[i]))
         => t >= bad_echos[i][1])
  AND (   (EXISTS (i: OP_IDS):
```

```
                (NOT op_accs[i]) AND (NOT fs_op_seen[i])
             AND op_echos[i] <= pi+reint_to
             AND op_echos[i] < frame_to
             AND t = op_echos[i])
       OR (EXISTS (j: BAD_IDS):
                (NOT bad_accs[j]) AND (NOT fs_bad_seen[j])
             AND bad_echos[j][1] <= pi+reint_to
             AND bad_echos[j][1] < frame_to
             AND t = bad_echos[j][1])
       OR (frame_to < reint_to+pi AND t = frame_to));
% ------------------------------------

% ------------------------------------
% Functions for counting the number of accused
% for the synchronization mode.
not_accd_rec(op_accs: OP_ACCS, bad_accs: BAD_ACCS,
             i: ALL_CNT, cnt: ALL_CNT): ALL_CNT =
  IF i=0 THEN cnt
  ELSE (IF operational?(i)
        THEN
          (IF op_accs[i]
           THEN not_accd_rec(op_accs, bad_accs, i-1, cnt)
           ELSE not_accd_rec(op_accs, bad_accs, i-1, cnt+1)
           ENDIF)
        ELSE
          (IF bad_accs[i]
           THEN not_accd_rec(op_accs, bad_accs, i-1, cnt)
           ELSE not_accd_rec(op_accs, bad_accs, i-1, cnt+1)
           ENDIF)
        ENDIF)
  ENDIF;

not_accd(op_accs: OP_ACCS, bad_accs: BAD_ACCS): ALL_CNT =
  not_accd_rec(op_accs, bad_accs, total, 0);
% ------------------------------------

% How many nodes seen in synch capture mode.
sc_seen_rec(sc_op_seen: OP_ACCS, sc_bad_seen: BAD_ACCS,
            i: ALL_CNT, cnt: ALL_CNT): ALL_CNT =
  IF i = 0 THEN cnt
  ELSE
    (IF operational?(i)
     THEN
       (IF sc_op_seen[i]
        THEN sc_seen_rec(sc_op_seen, sc_bad_seen, i-1, cnt+1)
        ELSE sc_seen_rec(sc_op_seen, sc_bad_seen, i-1, cnt)
        ENDIF)
     ELSE
       (IF sc_bad_seen[i]
        THEN sc_seen_rec(sc_op_seen, sc_bad_seen, i-1, cnt+1)
```

```
        ELSE sc_seen_rec(sc_op_seen, sc_bad_seen, i-1, cnt)
        ENDIF)
     ENDIF)
  ENDIF;

sc_seen_total(sc_op_seen: OP_ACCS,
              sc_bad_seen: BAD_ACCS): ALL_CNT =
  sc_seen_rec(sc_op_seen, sc_bad_seen, total, 0);

% Defined as a predicate rather than a recursive function;
% see Dutertre and Sorea's tech report for an explanation.
% Determining the next valid echo for the synchronization
% mode transitions.
next?(t: TIME, reint_to: TIME, op_echos: OP_ECHOS,
      bad_echos: BAD_ECHOS, fs_op_seen: OP_ACCS,
      fs_bad_seen: BAD_ACCS, op_accs: OP_ACCS,
      bad_accs: BAD_ACCS, frame_to: TIME): BOOLEAN =
      (FORALL (i: OP_IDS):
            (    op_echos[i] > reint_to
             AND (NOT op_accs[i]) AND (NOT fs_op_seen[i]))
        => t <= op_echos[i])
  AND (FORALL (i: BAD_IDS):
           (NOT bad_accs[i]) AND (NOT fs_bad_seen[i])
        => t <= bad_echos[i][1])
  AND (   (EXISTS (i: OP_IDS):
                (NOT op_accs[i]) AND (NOT fs_op_seen[i])
             AND reint_to < op_echos[i]
             AND op_echos[i] < frame_to
             AND t = op_echos[i])
       OR (EXISTS (j: BAD_IDS):
                (NOT bad_accs[j]) AND (NOT fs_bad_seen[j])
             AND bad_echos[j][1] < frame_to
             AND t = bad_echos[j][1])
       OR t = frame_to);

% Ensures the array of echos from faulty nodes satisfy
% are ascending.
ascending?(be: BAD_ECHO_ARRAY, reint_to: TIME): BOOLEAN =
  FORALL (e: [1..pd_see_top-1]):     be[e] > reint_to
                                  AND be[e] < be[e+1];
% ------------------------------------------------------------



% ------------------------------- MODEL --------------------
% MODES --------------------------------------------------
modes: MODULE =
BEGIN
  INPUT
    pd_cntrl           : CNTRL,
```

```
    fs_cntrl           : CNTRL,
    sc_cntrl           : CNTRL
  OUTPUT
    mode               : MODES
  INITIALIZATION
    mode = pd_mode
  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 -->
  ]
END;
% ------------------------------------------------------------


% PRELIMINARY DIAGNOSIS MODE ---------------------------------
preliminary_diagnosis_mode: MODULE =
BEGIN
  INPUT
    mode             : MODES,
    frame_to         : TIME,
    op_echos         : OP_ECHOS,
    bad_echos        : BAD_ECHOS
  OUTPUT
    pd_cntrl         : CNTRL
  LOCAL
    pd_finish        : TIME,
    pd_op_seen       : PD_OP_SEEN,
    pd_bad_seen      : PD_BAD_SEEN
  GLOBAL
    op_accs          : OP_ACCS,
    bad_accs         : BAD_ACCS,
    reint_to         : TIME
  INITIALIZATION
    pd_cntrl = active;
    op_accs = [[i: OP_IDS] FALSE];
    bad_accs = [[i: BAD_IDS] FALSE];
    pd_op_seen = [[i: OP_IDS] 0];
    pd_bad_seen = [[i: BAD_IDS] 0];
    reint_to IN {t: TIME | t >= 0 AND t < P};
    pd_finish = reint_to+P+pi
```

```
TRANSITION
[
        mode' = pd_mode
    AND frame_to < pd_finish
    -->
      reint_to' = frame_to;
      pd_op_seen' = [[i: OP_IDS]
                      IF (NOT op_accs[i])
                      THEN pd_op_echos(pd_op_seen[i],
                                       op_echos[i],
                                       reint_to, reint_to')
                      ELSE pd_op_seen[i] ENDIF];
      pd_bad_seen' = [[i: BAD_IDS]
                       IF (NOT bad_accs[i])
                       THEN pd_bad_echos(pd_bad_seen[i],
                                         bad_echos[i],
                                         reint_to,
                                         reint_to')
                       ELSE pd_bad_seen[i] ENDIF];
      op_accs' = [[i: OP_IDS]
                      op_accs[i]
                   OR pd_op_seen'[i] = pd_see_top];
      bad_accs' = [[i: BAD_IDS]
                       bad_accs[i]
                    OR pd_bad_seen'[i] = pd_see_top]
 [] % ------------------------------------------------------
        mode' = pd_mode
    AND frame_to >= pd_finish
    -->
      pd_cntrl' = deactive;
      reint_to' = pd_finish;
      pd_op_seen' = [[i: OP_IDS]
                      IF (NOT op_accs[i])
                      THEN pd_op_echos(pd_op_seen[i],
                                       op_echos[i],
                                       reint_to, reint_to')
                      ELSE pd_op_seen[i] ENDIF];
      pd_bad_seen' = [[i: BAD_IDS]
                       IF (NOT bad_accs[i])
                       THEN pd_bad_echos(pd_bad_seen[i],
                                         bad_echos[i],
                                         reint_to, reint_to')
                       ELSE pd_bad_seen[i] ENDIF];
      op_accs' = [[i: OP_IDS]
                      op_accs[i]
                   OR pd_op_seen'[i] = pd_see_top
                   OR pd_op_seen'[i] = 0];
      bad_accs' = [[i: BAD_IDS]
                       bad_accs[i]
                    OR pd_bad_seen'[i] = pd_see_top
```

```
                    OR pd_bad_seen'[i] = 0]
  ]
END;
% ------------------------------------------------------------


% FRAME SYNCHRONIZATION ------------------------------------
frame_synchronization_mode: MODULE =
BEGIN
  INPUT
    mode            : MODES,
    op_echos        : OP_ECHOS,
    bad_echos       : BAD_ECHOS,
    frame_to        : TIME
  OUTPUT
    fs_cntrl        : CNTRL
  LOCAL
    fs_op_seen      : OP_ACCS,
    fs_bad_seen     : BAD_ACCS
  GLOBAL
    op_accs         : OP_ACCS,
    bad_accs        : BAD_ACCS,
    reint_to        : TIME
  INITIALIZATION
    fs_cntrl = deactive;
    fs_op_seen = [[i: OP_IDS] FALSE];
    fs_bad_seen = [[i: BAD_IDS] FALSE]
  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, frame_to)};
        fs_op_seen' = [[i: OP_IDS]
                           fs_op_seen[i]
                       OR (    op_echos[i] > reint_to
                           AND op_echos[i] <= reint_to')];
        fs_bad_seen' = [[i: BAD_IDS]
                           fs_bad_seen[i]
                        OR bad_echos[i][1] <= reint_to'];
        op_accs' = [[i: OP_IDS]
                       op_accs[i]
```

```
                    OR (    op_echos[i] > reint_to
                        AND op_echos[i] <= reint_to'
                        AND fs_op_seen[i])];
        bad_accs' = [[i: BAD_IDS]
                        bad_accs[i]
                     OR (    bad_echos[i][1] <= reint_to'
                         AND fs_bad_seen[i])];
    []
          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' = deactive;
        reint_to' = reint_to+pi;
        op_accs' = [[i: OP_IDS]
                       op_accs[i]
                    OR (    op_echos[i] > reint_to
                        AND op_echos[i] <= reint_to'
                        AND fs_op_seen[i])];
        bad_accs' = [[i: BAD_IDS]
                        bad_accs[i]
                     OR (    bad_echos[i][1] <= reint_to'
                         AND fs_bad_seen[i])];
  ]
END;
% ------------------------------------------------------------


% SYNCHRONIZATION CAPTURE MODE -------------------------------
synch_capture_mode: MODULE =
BEGIN
  INPUT
    mode            : MODES,
    frame_to        : TIME,
    op_echos        : OP_ECHOS,
    bad_echos       : BAD_ECHOS
  OUTPUT
    sc_cntrl        : CNTRL
  LOCAL
    sc_op_seen      : OP_ACCS,
    sc_bad_seen     : BAD_ACCS
  GLOBAL
    reint_to        : TIME,
    op_accs         : OP_ACCS,
    bad_accs        : BAD_ACCS
  INITIALIZATION
    sc_cntrl = deactive;
    sc_op_seen = [[i: OP_IDS] FALSE];
    sc_bad_seen = [[i: BAD_IDS] FALSE]
```

```
  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)};
        sc_op_seen' = [[i: OP_IDS]
                           sc_op_seen[i]
                        OR op_echos[i] = reint_to'];
        sc_bad_seen' = [[i: BAD_IDS]
                            sc_bad_seen[i]
                         OR bad_echos[i][1] = reint_to'];
   []
          mode' = sc_mode
      AND   sc_seen_total(sc_op_seen, sc_bad_seen)
          > not_accd(op_accs, bad_accs)/2
      -->
        sc_cntrl' = deactive
  ]
END;
% ------------------------------------------------------------


% CLIQUE -------------------------------------------------
op_node[i: OP_IDS]: MODULE =
BEGIN
  INPUT
    frame_to: TIME,
    reint_to: TIME
  OUTPUT
    op_echo: TIME
  INITIALIZATION
    op_echo IN {t: TIME | frame_to > t AND t > frame_to-pi}
  TRANSITION
  [
      frame_to <= reint_to'
      -->
        op_echo' IN {t: TIME |     frame_to' > t
                               AND t > frame_to'-pi}
   []
      ELSE -->
  ]
END;
```

```
P_update: MODULE =
BEGIN
  INPUT
    reint_to: TIME
  LOCAL
    new_frame: BOOLEAN
  OUTPUT
    frame_to: TIME
  INITIALIZATION
    frame_to = IF reint_to >= pi THEN P+pi ELSE pi
               ENDIF;
    new_frame = FALSE
  TRANSITION
  [
      frame_to <= reint_to'
      -->
        frame_to' = frame_to+P;
        new_frame' = TRUE
   []
      ELSE -->
        new_frame' = FALSE
  ]
END;

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;
%-------------------------------------------------------------


% BAD NODES ----------------------------------------------
bad_node[i: BAD_IDS]: MODULE =
BEGIN
  INPUT
    reint_to: TIME
  OUTPUT
    bad_echo: BAD_ECHO_ARRAY
  INITIALIZATION
    bad_echo IN {be: BAD_ECHO_ARRAY
                   | ascending?(be, reint_to)}

  TRANSITION
  [
      bad_echo[1] <= reint_to'
      -->
        bad_echo' IN {be: BAD_ECHO_ARRAY
                        | ascending?(be, reint_to')}
```

```
   []
      ELSE -->
  ]
END;

bad_nodes: MODULE =
  WITH OUTPUT bad_echos: BAD_ECHOS
    (|| (i: BAD_IDS): RENAME bad_echo TO bad_echos[i]
                      IN bad_node[i]);
%------------------------------------------------------------


% FULL SYSTEM ------------------------------------------------
base_modes: MODULE =
     preliminary_diagnosis_mode
  []
     frame_synchronization_mode
  []
     synch_capture_mode;

reintegrator: MODULE = base_modes || modes;

system: MODULE = reintegrator || clique || bad_nodes;
%------------------------------------------------------------



% ------------------------------ CONJECTURES ---------------
% Depths for a model with 2-3 operational nodes and
% one faulty node.

% In the comments above a conjecture, "dn", where n is a
% natural number, is the k-induction depth.  The lemmas
% follow, preceded by "-l."

% NOTE: With a great number of nodes, it is often useful to
% disable disable expensive buchi automata optimizations
% by setting the --disable-expensive-ba-opt flag.

% SYSTEM INVARIANTS ------------------------------------------
% If a mode is enabled, then the other modes are deactive.
% proved d1
mode_cntrl: LEMMA
  system |- G(    (   mode = pd_mode
                   => (    fs_cntrl = deactive
                       AND sc_cntrl = deactive))
              AND (   mode = fs_mode
                   => (    pd_cntrl = deactive
                       AND sc_cntrl = deactive))
              AND (   mode = sc_mode
```

```
                  => (    pd_cntrl = deactive
                      AND fs_cntrl = deactive)));

% Echos from operational nodes satisfy the frame property.
% proved d1
frame_prop: LEMMA
  system |- G(FORALL (i: OP_IDS):
                   frame_to > op_echos[i]
                AND frame_to-op_echos[i] < pi);
%-------------------------------------------------------------


% PRELIMINARY DIAGNOSIS INVARIANTS --------------------------
% Bounds the finish time for the preliminary diagnosis mode.
% proved d1
pd_finish: LEMMA system |- G(pd_finish < 2*P+pi);

% No operational nodes are accused upon initialization.
% proved d1
pd_init_op_accs: LEMMA
  system |- G(FORALL (i: OP_IDS):
                  (    mode = pd_mode AND pd_op_seen[i] = 0
                   AND pd_cntrl = active)
               => NOT op_accs[i]);

% Operational nodes are seen no more than twice during the
% preliminary diagnosis mode.
% proved d4 -l pd_finish -l mode_cntrl
op_seen_less2: LEMMA
  system |- G(FORALL (i: OP_IDS): pd_op_seen[i] <= 2);

% Operational nodes are seen no less than once during the
% preliminary diagnosis mode.
% proved d3 -l mode_cntrl -l pd_init_op_accs
op_seen_more1: LEMMA
  system |- G(   pd_cntrl = deactive
              => FORALL (i: OP_IDS): pd_op_seen[i] >= 1);

% No operational nodes are accused in the preliminary
% diagnosis mode.
% proved d1 -l op_seen_more1 -l op_seen_less2
pd_no_op_accs: LEMMA
  system |- G(   mode = pd_mode
              => FORALL (i: OP_IDS): NOT op_accs[i]);

% The frame synchronization mode seen variables are
% unchanged in the preliminary diagnosis mode.
% proved d1
pd_not_fs_seen: LEMMA
  system |- G(   mode = pd_mode
```

```
              => (    (FORALL(i: OP_IDS): NOT fs_op_seen[i])
                  AND (FORALL(i: BAD_IDS): NOT fs_bad_seen[i])));

% The synchronization capture mode seen variables are
% unchanged in the preliminary diagnosis mode.
% proved d1
pd_not_sc_seen: LEMMA
  system |- G(   mode = pd_mode
              => (    (FORALL(i: OP_IDS): NOT sc_op_seen[i])
                  AND (FORALL(i: BAD_IDS): NOT sc_bad_seen[i])));
%-------------------------------------------------------------


% FRAME SYNCHRONIZATION INVARIANTS --------------------------
% No operational nodes are accused when the frame
% synchronization mode initializes.
% proved d1 -l pd_no_op_accs
fs_init_no_op_accs: LEMMA
  system |- G(FORALL (i: OP_IDS):
                 (mode = fs_mode AND NOT fs_op_seen[i])
               => NOT op_accs[i]);

% The frame gap is found at the conclusion of the
% frame synchronization mode.
% proved d3 -l pd_no_op_accs -l fs_init_no_op_accs
%           -l frame_prop -l pd_not_fs_seen
fs_frame_gap: LEMMA
  system |- G(   (mode = fs_mode AND fs_cntrl = deactive)
              => (FORALL (i: OP_IDS):
                    reint_to < op_echos[i]));

% Demonstrates that the reintegrator's timeout has passed
% the echo of an operational node in the current frame
% if it has seen the echo already.
% proved d3 -l mode_cntrl -l pd_not_fs_seen
%           -l fs_init_no_op_accs -l pd_no_op_accs
%           -l frame_prop -l fs_frame_gap
fs_window: LEMMA
  system |- G(   fs_cntrl = active
              => (FORALL (i: OP_IDS):
                        (reint_to > frame_to-pi
                         AND fs_op_seen[i]
                         AND NOT reint_to = op_echos[i])
                    => reint_to > op_echos[i]));

% No operational nodes are accused in the frame
% synchronization mode.
% proved d3 -l mode_cntrl -l pd_not_fs_seen
%           -l fs_init_no_op_accs -l pd_no_op_accs
%           -l frame_prop -l fs_window
```

```
fs_no_op_accs: LEMMA
  system |- G(   mode = fs_mode
              => FORALL (i: OP_IDS): NOT op_accs[i]);

% The seen state variables for the synchronization capture
% mode are invariant in the frame synchronization mode.
% proved d1 -l pd_not_sc_seen
fs_not_sc_seen: LEMMA
  system |- G(   mode = fs_mode
              => (    (FORALL(i: OP_IDS): NOT sc_op_seen[i])
                  AND (FORALL(i: BAD_IDS): NOT sc_bad_seen[i])));
%-------------------------------------------------------------


% SYNCHRONIZATION CAPTURE INVARIANTS ------------------------
% No operational nodes are accused during the reintegration
% protocol.
% proved d1 -l fs_no_op_accs -l pd_no_op_accs
no_op_accs: THEOREM
  system |- G(FORALL (i: OP_IDS): NOT op_accs[i]);


% When the synchronization protocol initializes, the
% reintegrator is in a frame gap.
% proved d1 -l mode_cntrl -l frame_prop -l no_op_accs
%           -l fs_not_sc_seen -l fs_frame_gap
sc_init_frame_gap: LEMMA
  system |- G(    mode = sc_mode
              => (FORALL (i: OP_IDS):
                        NOT sc_op_seen[i]
                    => reint_to < op_echos[i]));

% The reintegration protocol synchronizes the reintegrator.
% proved d4 -l mode_cntrl -l frame_prop -l no_op_accs
%           -l fs_not_sc_seen -l sc_init_frame_gap
synched: THEOREM
  system |- G((mode = sc_mode AND sc_cntrl = deactive)
              => (FORALL (i: OP_IDS):
                    IF reint_to >= op_echos[i]
                    THEN reint_to - op_echos[i] < pi
                    ELSE op_echos[i] - reint_to < pi
                    ENDIF));
%-------------------------------------------------------------


% MODEL INVARIANTS ------------------------------------------
% To prove the timeout automata model with time-triggered
% transitions is faithful.

% The bad echo array behaves correctly.
```

```
% proved d1
bad_echos_ascend: LEMMA
  system |- G(FORALL (i: BAD_IDS):
               FORALL (e: [1..pd_see_top-1]):
                      bad_echos[i][e] < bad_echos[i][e+1]);

% The reintegrator's timeout is always less than the frame
% timeout and the echos of the faulty nodes.
% proved d2 -l mode_cntrl -l sc_init_frame_gap
%           -l fs_frame_gap -l frame_prop -l bad_echos_ascend
reint_to_least: LEMMA
  system |- G(    reint_to < frame_to
              AND (FORALL (i: BAD_IDS):
                     FORALL (e: [1..pd_see_top]):
                       reint_to < bad_echos[i][e]));

% The reintegrator's timeout is always within the current
% frame.
% proved d3 -l reint_to_least -l fs_frame_gap -l synched
current_frame: LEMMA
  system |- G(frame_to - reint_to <= P);

% Whenever the reintegrator moves to a new frame, it
% does not immediately move to the window in which
% operational nodes issue their echos.
% proved d2
good_frame_update: LEMMA
  system |- G(new_frame => reint_to <= frame_to-pi);
%------------------------------------------------------------


% LIVENESS CHECKS -----------------------------------------
% These should be false.  A counterexample to them provides
% some assurance that the model is not deadlocked (making
% conjectures vaccuously true).

% Counter examples at
% d4 -l frame_prop -l mode_cntrl -l pd_finish
%    -l pd_not_fs_seen -l pd_not_sc_seen -l fs_frame_gap
%    -l fs_window -l fs_no_op_accs -l no_op_accs
%    -l synched -l sc_init_frame_gap -l pd_no_op_accs
%    -l current_frame -l reint_to_least
pd_ck: LEMMA system |- G(mode /= pd_mode);
fs_ck: LEMMA system |- G(mode /= fs_mode);
sc_ck: LEMMA system |- G(mode /= sc_mode);
%------------------------------------------------------------

END
```

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
01- 04 - 2005	Technical Memorandum	

4. TITLE AND SUBTITLE
Real-Time System Verification by *k*-Induction

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Pike, Lee

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER
23-063-30-RF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
NASA Langley Research Center
Hampton, VA 23681-2199

8. PERFORMING ORGANIZATION REPORT NUMBER
L-19110

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSOR/MONITOR'S ACRONYM(S)
NASA

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
NASA/TM-2005-213751

12. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified - Unlimited
Subject Category 62
Availability: NASA CASI (301) 621-0390

13. SUPPLEMENTARY NOTES
An electronic version can be found at http://ntrs.nasa.gov

14. ABSTRACT

We report the first formal verification of a reintegration protocol for a safety-critical, fault-tolerant, real-time distributed embedded system. A reintegration protocol increases system survivability by allowing a node that has suffered a fault to regain state consistent with the operational nodes. The protocol is verified in the Symbolic Analysis Laboratory (SAL), where bounded model checking and decision procedures are used to verify infinite-state systems by *k*-induction. The protocol and its environment are modeled as synchronizing timeout automata. Because *k*-induction is exponential with respect to *k*, we optimize the formal model to reduce the size of *k*. Also, the reintegrator's event-triggered behavior is conservatively modeled as time-triggered behavior to further reduce the size of *k* and to make it invariant to the number of nodes modeled. A corollary is that a clique avoidance property is satisfied.

15. SUBJECT TERMS
Fault-tolerance; Formal verification; Model checking; Real-time systems; Reintegration protocol

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			STI Help Desk (email: help@sti.nasa.gov)
					19b. TELEPHONE NUMBER *(Include area code)*
U	U	U	UU	66	(301) 621-0390