Formal Timing Analysis for Fault-Tolerant Active Structural Control Systems

Wael M. Elseaidy* Rance Cleaveland† John W. Baugh Jr.‡

Abstract

This paper presents the results of a case study involving the use of a formal graphical notation, Modechart, and an automatic verification tool, the Concurrency Workbench, to the analysis of the design of a fault-tolerant active structural control system. Such control systems must satisfy strict requirements on their timing behavior; we show how to use various features supported by the Workbench to examine the timing behavior of different design alternatives, one of which has in excess of 10^{19} states. The simple techniques illustrated here may be used by systems designers to model and analyze other large and complex real-time systems as well.

1 Introduction

The development of inexpensive microprocessors over the past twenty years has prompted engineers to investigate the inclusion of embedded computer control systems in a variety of different engineering applications. Such control systems aim to improve the performance of these applications, examples of which may be found in aeronautical, mechanical, chemical, and civil engineering arenas, among others. In many cases the structures or processes being controlled are *safety-critical* in that human lives and physical well-being depend on them; ensuring that these systems behave correctly and reliably is of utmost importance. Designers of such systems are thus confronted with verifying that their systems behave correctly and with including in their designs mechanisms for fault-tolerance. In the case of real-time systems, however, these two design goals can come into conflict: the introduction of redundant components can alter the timing behavior of systems in subtle ways, thus invalidating correctness arguments obtained for their easier-to-analyze nonredundant counterparts. Accordingly, traditional design practice for real-time, fault-tolerant and safety-critical systems relies on conservative strategies and the use of expensive and time-consuming simulation and testing in order to validate system performance; nevertheless, errors still arise, with sometimes tragic results.

Our aim in this paper is to illustrate how formal reasoning may be used to analyze the timing behavior of a fault-tolerant real-time control application from the civil engineering field. In principle, formal methods offer a system builder the means to reduce testing and maintenance costs by providing a framework in which the builder can prove his system correct. In practice, formal methods have not been widely adopted, one main reason being that traditional approaches do not "scale up" to large and complex systems. In this case study, we show how one may use a process-algebra-based automatic formal verification tool, the Concurrency Workbench[5], to study the timing behavior of a fault-tolerant active structural control architecture based on the design given in [3, 4] and formalized in [7, 8]. In particular, our approach highlights the benefits of *state-space minimization* as a means of verifying complex systems such as the one considered here, which has in excess of 10^{19} states, in a modular, and hence scalable, manner.

The remainder of the paper is structured as follows. The next section reviews basic results from the civil-engineering literature regarding active structural control systems, while the section following presents the design language we use to model the real-time systems and gives a brief overview of the Concurrency Workbench. Section 4 then describes the fault-tolerant system to be analyzed, and the section following discusses our use of the Workbench to analyze its behavior. The final section closes with our conclusions and directions for future work.

2 Active Structural Control Systems

Active structures include an embedded system that acts to limit structural vibration due to external excitations such as earthquakes or high airspeed. Typically, such structures include adjustable members, or *actuators*, that may

*Research Scientist, Alphatronix Inc, Research Triangle Park, NC 27709-3978. E-mail: `wmelseai@eos.ncsu.edu`.

†Associate Professor, Department of Computer Science, North Carolina State University, Raleigh, NC 27695. E-mail: `rance@csc.ncsu.edu`. Research support provided by NSF grant CCR-9120995, ONR Young Investigator Award N00014-92-J-1582, NSF Young Investigator Award CCR-9257963, and NSF grant CCR-9402807.

‡Associate Professor, Departments of Civil Engineering and Computer Science, North Carolina State University Raleigh, NC 27695-7908. E-mail: `jwb@eos.ncsu.edu`. Research support provided by NSF grant MSS-9201687.

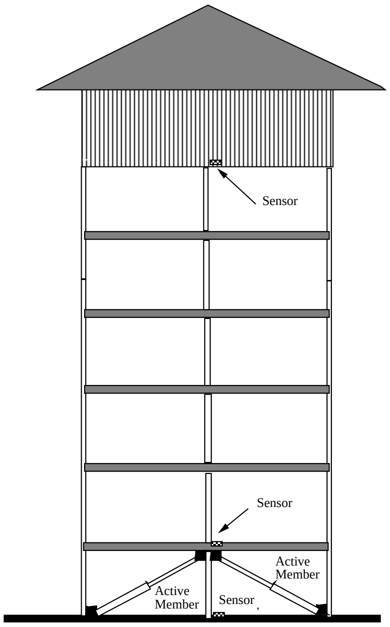

Figure 1: An Actively Controlled Structure in Tokyo.

be lengthened or contracted to counteract the external forces applied to the structure; a process controller monitors sensors and sends commands to the actuators when sensor readings indicate an undesirable state.

A major application of active structure control systems involves earthquake-resistant buildings. Earthquake damage often results less from the violence of movement than from the vibrations they induce in buildings; in particular, if they cause structures to vibrate at their resonant frequency then the structures become unstable and may collapse. To minimize vibration-induced earthquake damage, one must therefore locate the natural frequencies of a structure outside the frequency band of the seismic excitations produced by earthquake. An active structural control system attempts to do this by sensing the seismic excitations with a high sampling rate and changing the natural frequencies of the structure using the active members, with the particular method for changing these frequencies depending on the control algorithm used. Such control systems must satisfy certain timing constraints on the activity of the actuators, since if the needed adjustments of active members are not applied within the time bounds required by the control algorithm then these structures may become unstable [2].

Figure 1 contains a cross-section of a six-story office block in Tokyo that was built to test the feasibility of active structural control in earthquake resistance. The structure uses four active members connected to the first floor [14].

2.1 Pulse-control Algorithms

Different control techniques have been proposed for active structures in the literature [1, 17]; in this paper, we focus on one of the simpler approaches, *pulse control*. Pulse-control algorithms aim to limit the vibratory displacement of structures near resonance by applying an opposing pulse at a higher frequency to "break up" the resonant forces. The major design variables are the time between pulse initiations, Δt_p, and the pulse duration, Δt; these values are determined by the natural frequencies of the system, the expected forcing functions, and the desired level of displacement control. Figure 2 depicts the block diagram of a pulse-control algorithm.

In this paper we are concerned with verifying when a specific system satisfies constraints on Δt_p. Soong [16] has established the following. Let T_0 be the natural period of a structure. Then Δt_p should satisfy: $\frac{T_0}{8} \leq \Delta t_p \leq \frac{T_0}{2}$. A parametric study of simulated system behavior [15] suggests similar constraints on delays between pulse initiations.

2.2 An Experimental System

In [16] an experimental set-up is described for an actively controlled single-degree-of-freedom structure with a natural period of 290 msec $* 10^{-1}$.[1] As the experimental structure will serve as the basis of the fault-tolerant architecture we analyze later in this paper, we briefly describe it here, paying particular attention to the timing characteristics

[1] This experiment provided results used to obtain the timing bounds on Δt_p given above.

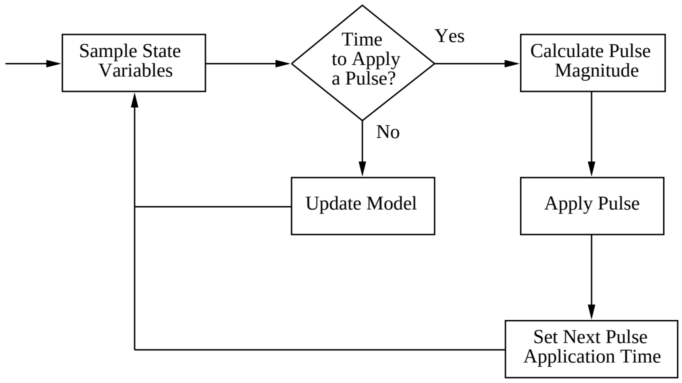

Figure 2: Block Diagram of a Pulse-Control Algorithm.

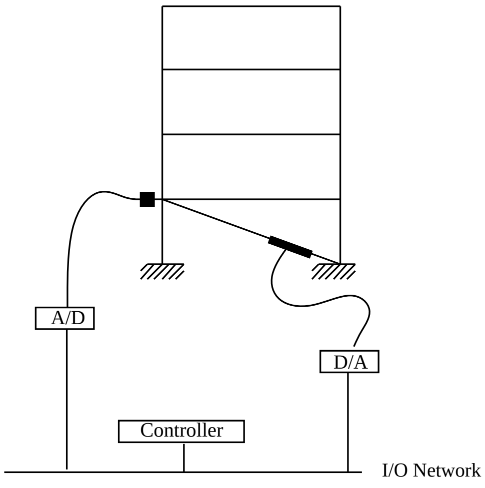

Figure 3: An Experimental Active Structural Control System.

obtained for the different components in the system. Note that the natural period implies the following bounds on Δt_p: $37 \text{ msec} * 10^{-1} \leq \Delta t_p \leq 145 \text{ msec} * 10^{-1}$.

A schematic description of the system appears in Figure 3; it contains a sensor, actuator and controller connected by an I/O network. The following table gives the (experimentally determined) time for different system activities.

Activity	Duration (in msec $* 10^{-1}$)
Sampling	[50,55]
Model updating	[20,25]
Pulse magnitude calculation	[40,45]
Pulse duration	[25,30]

The only system activity for which no information was given was the time required for data to be exchanged among the components. Based on our experiments with a Unix-based communication system, however, we have determined that 15 msec $* 10^{-1}$ appears to be reasonable, with 10 msec $* 10^{-1}$ devoted to connection establishment and 5 to message and acknowledgement transmission. The assumption that communication timings are deterministic is justified by the fact that the communication topology and amount of data exchanged by processes is fixed.

3 Modechart and the Concurrency Workbench

This section describes the language, Modechart, that we use for modeling active structural control systems and a verification tool, the Concurrency Workbench, used in the analysis of the models.

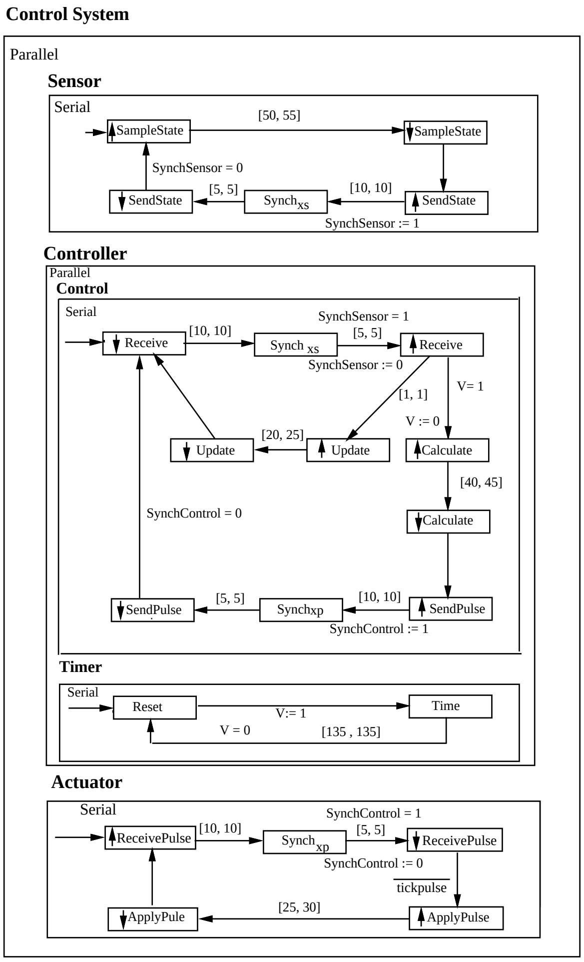

Figure 4: Modechart Model of a Control System.

3.1 Modechart

Modechart [10, 11] is a expressive yet intuitive notation that supports the graphical representation of real-time systems as hierarchical state machines. The language is a real-time extension of the Statecharts formalism [9] for modeling untimed concurrent systems. Like Statecharts, Modechart includes two basic constructs: *modes*, which may be thought of as "structured states", and transitions between modes. The notation we describe here is a simplified version of the full language, a more detailed account of which may be found in [10].

Modes may be one of three types: *primitive*, *serial* and *parallel*. Primitive modes have no internal structure and correspond precisely to states in a traditional finite-state machine. Serial and parallel modes may contain a number of child modes and are intended to capture the notion of sequential and parallel execution, respectively. Serial modes must also have a child mode designated as initial. When a serial mode is entered, its initial child mode is also entered at the same time, while when a parallel mode is entered, each of its child modes is entered simultaneously.

Modechart systems change modes by engaging in transitions, which in our account are equipped with three components: a *condition*, a *time bound* of the form $[l, u]$, and an *effect*. Intuitively, a transition becomes enabled when its condition becomes true; it may then fire any time between l and u time units after it is enabled, with the effect then altering the state of the system accordingly. If two or more transitions are enabled at the same time the choice is nondeterministic. For our purposes a condition will either consist of the testing of an integer-valued state variable or the checking of the occurrence of a (user-defined) event elsewhere in the system. An effect will either be an assignment to a boolean variable or the emission of an event, which we indicate by drawing a line over the event name: thus $\overline{e}$ corresponds to the "production" of event e. If a condition is omitted from a transition it is assumed to be "true", while if a timing constraint is left out from a transition, it is assumed to be $[0, 0]$. The formal semantics of Modechart is given via a translation into the logic RTL [10].

Figure 4 comes from [8] and gives a Modechart rendering of the active structural control system described above.[2] The over-all system consists of a parallel mode with three children: one corresponding to the sensor, one for the actuator, and one for the controller. The controller in turn consists of two parallel subcomponents, one implementing the control algorithm and one acting as a *timer* that controls the times between successive pulses. The different components synchronize with one another using variables; for example, to activate the timer the controller sets variable V to 0; the timer "goes off" 135 time units later by setting this variable to 1. Similarly, the sensor and controller use the variable *SynchSensor* to model the fact that the communication between them has an element of synchrony induced by the exchange of data and acknowledgements. In the diagrams, conditions and timing constraints appear on one side of transitions, while effects appear on the other. The timing values come from the table given in Section 2.2; note that the (emission of) the $\overline{tickpulse}$ event allows observation of the beginning of pulse applications.

3.2 The Concurrency Workbench

The Concurrency Workbench [5] supports the simulation and verification of concurrent finite-state systems expressed in the process-algebraic design language Temporal CCS [13]. In addition to facilities for simulation and various forms of reachability analysis, the tool includes routines for determining when a system satisfies temporal logic formulas, for establishing whether or not two different systems are "semantically indistinguishable" (in a precise sense), and for minimizing the state spaces of systems with respect to a particular notion of semantic equivalence. In the remainder of this section we provide more detail on Temporal CCS and on the notions of semantic equivalence and minimization supported by the Workbench.

3.2.1 Temporal CCS

Temporal CCS supports the design of timed communicating systems in which processes synchronize with one another using "channels". It constitutes an extension of Milner's Calculus of Communication Systems (CCS) [12] with time; like CCS it provides constructs for building up processes from *actions*, which may take one of four forms.

- $d \in Nat$ represents a delay of d time units.
- α represents the "receipt" of a synchronization signal on "channel" α.
- $\overline{\alpha}$ represents the "delivery" of a synchronization signal on α.
- τ represents a step of internal computation.

We use Act to represent the set of all actions. Letting $\mathcal{X}$ be a set of *identifiers* ranged over by $X, \ldots$, and $L \subseteq \Lambda$ be a set of channels, Temporal CCS processes may be built using the following grammar.[3]

$$P ::= nil \mid X \mid a.P \mid P_1 + P_2 \mid P_1|P_2 \mid P\backslash L \mid P \; where \; X_1 = P_1 \ldots X_n = P_n \; end$$

nil represents the "terminated" process, while $a.P$ "executes" action a and then behaves like P. The $+$ operator is a form of choice, while $P|Q$ represents the parallel composition of P and Q; note that if P may receive and Q may deliver a synchronization on the same channel then a synchronization is allowed, with $P|Q$ engaging in a τ-action while both P and Q execute their transitions. The restriction construct $\backslash L$ allows the channels in L to be declared "local" to P in $P\backslash L$. In $P \; where \; X_1 = P_1 \ldots X_n = P_n$ the equations provide the definitions for the occurrences of the X_i in P.

The formal semantics of Temporal CCS is given as a relation $P \xrightarrow{a} Q$; intuitively, $P \xrightarrow{a} Q$ holds if P is capable of engaging in action a and evolving to Q. The definition of $\rightarrow$ ensures that the only "idling" transitions that may be inferred have the form $P \xrightarrow{1} Q$; that is, processes idle one time unit at a time. For the purposes of the modeling in this paper, we also found it necessary to add *maximal progress* to the semantics of Temporal CCS; this feature ensures that if a process can engage in an internal τ-transition then it must do so immediately; it cannot delay it by idling. The semantics forms the basis for generating (finite-) state machines from Temporal CCS expressions, with the transitions in the machine recording the execution steps of the system.

[2] It should be noted that the labels with primitive modes are included for explanatory purposes only. Intuitively, a $\uparrow$ represents the beginning of a phase of system activity, while a $\downarrow$ corresponds to its termination.

[3] The language also includes other constructs that are not mentioned here.

3.2.2 Observational Equivalence

A prominent feature of process algebras like Temporal CCS is the use of *semantic equivalences* in reasoning about processes. In such a framework, one formulates the specification of a system as another system that describes the "high-level" behavior desired by the designer; then verifying a proposed design amounts to showing that it is semantically equivalent to, or *indistinguishable from*, its specification. One well-studied semantic relation is *observational equivalence* [12], which is designed to relate processes on the basis of their observable (i.e. non-τ) behavior. To define it, let $P \stackrel{\epsilon}{\Rightarrow} Q$ hold if P can perform some number (possibly 0) of τ-transitions and evolve to Q. If a is an action, then we write $P \stackrel{a}{\Rightarrow} Q$ if there exist P_1, P_2 such that $P \stackrel{\epsilon}{\Rightarrow} P_1 \stackrel{a}{\rightarrow} P_2 \stackrel{\epsilon}{\Rightarrow} Q$; that is, $P \stackrel{a}{\Rightarrow} Q$ indicates that P may perform some internal computation, then a, then more internal computation in evolving to Q.

Definition 1 *Observational equivalence, $\approx$, is the largest relation such that whenever $P \approx Q$ holds, then the following must be true for all $a \in (Act \cup \{\epsilon\}) - \{\tau\}$.*

- *If $P \stackrel{a}{\Rightarrow} P'$ then there is a Q' such that $Q \stackrel{a}{\Rightarrow} Q'$ and $P' \approx Q'$.*
- *If $Q \stackrel{a}{\Rightarrow} Q'$ then there is a P' such that $P \stackrel{a}{\Rightarrow} P'$ and $P' \approx Q'$.*

Intuitively, $P \approx Q$ holds whenever P and Q are able to "mimic" each other's behavior.

$\approx$ enjoys two crucial properties that we exploit in what follows.

- $\approx$ is a *congruence* for all operators in Temporal CCS except $+$. That is, if $P \approx Q$ then P and Q may be used interchangeably inside any larger context that does not use the $+$ operator.
- Finite-state systems may be *minimized* with respect to $\approx$ [12]. That is, for any finite-state system M, there is a unique (up to isomorphism) minimum-state finite-state system $min(M)$ such that $M \approx min(M)$.

These two observations permit systems of parallel components to be minimized in a *compositional*, or component-wise manner: first replace the subsystems with their minimized counterparts, then minimize the resulting system. Because of the congruence property, the new system is guaranteed to be semantically equivalent to the original one. $\approx$, and min, may be computed in polynomial time [5]; the Workbench includes implementations of these procedures.

3.2.3 Translating Modechart to TCCS

In order to use the Workbench, with its support for compositional minimization, to analyze our Modechart designs we must show how Modechart may be translated into Temporal CCS in a behavior-preserving manner. Elseaidy [7, 8] describes such a translation for a subset of Modechart sufficient for the modeling in this paper. Most of the procedure is straightforward; parallel mode become parallel compositions in Temporal CCS, while multiple transitions emanating from a mode are rendered using the + operator. The translation of timing bounds on Modechart transitions uses nondeterministic choice (+ in conjunction with initial τ actions) to model the possibility of different delays in executing a transition, while the emission of events is modeled using "output events". The only real subtlety involves translating Modechart state variables; to model these, we adopt the classical approach of Milner [12], in which each shared variable is implemented as a CCS process that intuitively acts as a "server" for the variable. Other processes wishing to manipulate the value of the variable send "requests" to the server, which changes state accordingly. As an example of the translation, the following is the Temporal CCS rendering of the control mode in Figure 4.

$Control = StartReceive\ where$

$$
\begin{aligned}
StartReceive &= 10.Synch_{XS} \\
Synch_{XS} &= rSynchSensor1.5.\overline{wSynchSensor0}.StopReceive + 1.Synch_{XS} \\
StopReceive &= rV1.StartCalculate + 1.StartUpdate \\
StartUpdate &= rV0.20.Update1 \\
Update1 &= \tau.StopUpdate + \tau.1.Update2 \\
Update2 &= \tau.StopUpdate + \tau.1.Update3 \\
Update3 &= \tau.StopUpdate + \tau.1.Update4 \\
Update4 &= \tau.StopUpdate + \tau.1.Update5 \\
Update5 &= \tau.StopUpdate + \tau.1.StopUpdate \\
StopUpdate &= StartReceive \\
StartCalculate &= \overline{wV0}.40.Calculate1 \\
Calculate1 &= \tau.StopCalculate + \tau.1.Calculate2 \\
Calculate2 &= \tau.StopCalculate + \tau.1.Calculate3 \\
Calculate3 &= \tau.StopCalculate + \tau.1.Calculate4 \\
Calculate4 &= \tau.StopCalculate + \tau.1.Calculate5 \\
Calculate5 &= \tau.StopCalculate + \tau.1.StopCalculate \\
StopCalculate &= StartSendPulse \\
StartSendPulse &= 10.\overline{wSynchControl1}.Synch_{XP} \\
Synch_{XP} &= 5.StopSendPulse \\
StopSendPulse &= rSynchControl0.StartReceive + 1.StopSendPulse
\end{aligned}
$$

end

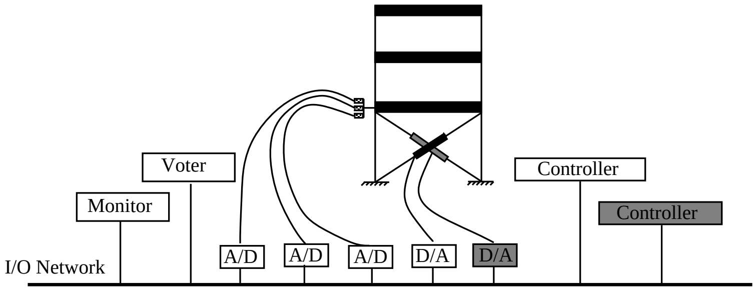

Figure 5: Schematic Drawing of Fault-Tolerant Active Structural Control System.

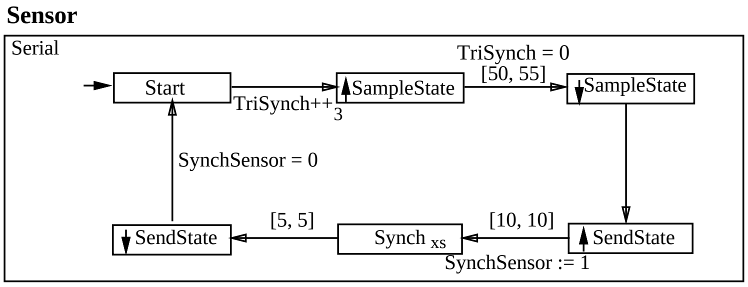

Figure 6: Modechart for a New Sensor.

4 A Fault-Tolerant Active Structural Control System

While useful for gathering experimental data about active structural control, the control system given in Figure 4 would be unsuitable for practical use because of its susceptibility to component faults. In particular, if any of the three components were to fail, the system would cease to function. Given the strains that earthquakes impose on an environment, designing a system to withstand component faults is essential to ensure optimal structure behavior.

In this section we present an elaboration of the design in Figure 4 that uses redundant components to introduce a measure of fault-tolerance into the control system. More specifically, the new design uses two forms of replication: *static*, in which the system architecture is fixed, and *dynamic*, in which system architecture may evolve during the execution of the system. These additions, and their associated recovery mechanisms, alter the timing behavior of the system; the next section examines this issue in more detail.

4.1 Static Redundancy and Sensors

Static redundancy is useful for components that may operate simultaneously without interfering with one another in the context of the larger system. To improve the reliability of a component with these characteristics, one may replicate it and then add a mechanism for turning the outputs of the replicas into a single value that is given to the rest of the system. One typical way to do this is to use voting; the replicated components transmit their outputs to a voter, which then selects the value (within certain tolerances) with a majority of votes and sends this to the rest of the system. Such a scheme is particularly good for systems in which a component might sporadically malfunction.

Figure 5 contains a schematic rendering of our fault-tolerant design, which employs a statically-redundant voting-based scheme for handling sensor failure. Instead of a single sensor, the new system has three; each communicates with a voter, who selects a value reported by at least two of the three for transmission to the controller. These are described in more detail below.

- **Sensors.** The sensors behave in essentially the same manner as the sensor described in the previous sections. The main difference is that we require the three sensors to synchronize with one another at the beginning of their measurement cycle. The necessity for this stems from the fact that the time required to collect data can vary from 50 to 55 msec $* 10^{-1}$ for each sensor and the fact that we wish the sensors to communicate at roughly the same time with the voter. A Modechart description of a new sensor may be found in Figure 6; note that the

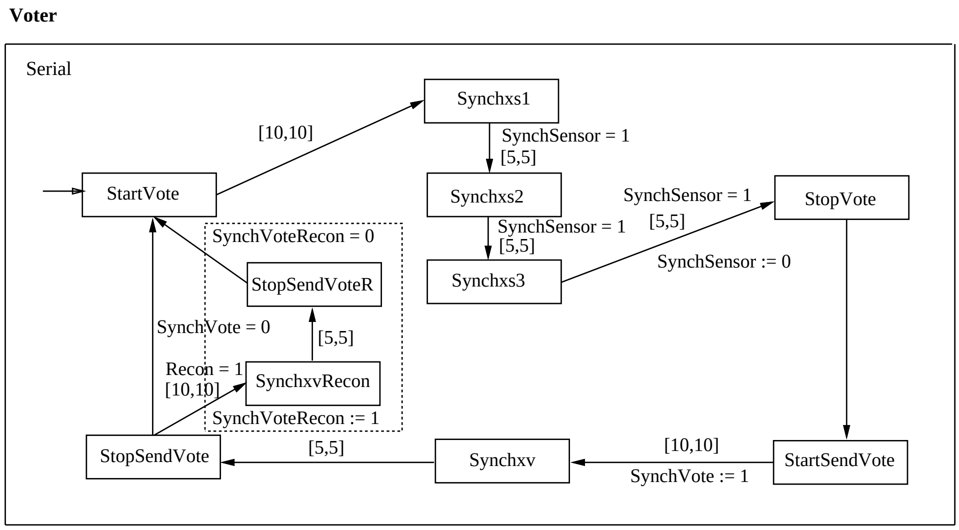

Figure 7: Voter for a Static Redundant System (Modes in Dotted Box May Be Omitted).

state variable $TriSynch$ is used enforce the three-way synchronization mechanism. The assignment statement $TriSynch + +_3$ increments $TriSynch$ modulo 3.

- **Voter.** The voter collects the sampling values from all three sensors by opening a socket (this consumes 10 msec $* 10^{-1}$) and then exchanging messages with each in turn (each exchange requires 5 msec $* 10^{-1}$). If the vote succeeds, meaning that a least two of the three values match, then the result is sent to the controller. Otherwise, the system should fail; however, in this simplified setting we assume that at least two of the three sensors are always functioning correctly. Figure 7 shows the Modechart of the voter; note that modes within the dotted box may be omitted (they are relevant only for the redundancy scheme used in the next section).

4.2 Dynamic Redundancy

In dynamically redundant systems, the occurrence of faults in a component causes the system to reconfigure itself; the faulty component is "bypassed" and a replica of it activated to assume its duties. During normal system functioning, redundant components are generally not operational. Such a scheme potentially incurs more overhead than static schemes, since one needs a monitoring scheme to detect faults, but they are appropriate when the simultaneous functioning of redundant components would lead to interference.

Our fault-tolerant design uses dynamic redundancy to improve the reliability of actuators. In the design we assume that each actuator has a dedicated controller; accordingly, replicating the actuator necessitates adding another controller as well. Figure 8 shows a high-level Modechart description of our fault-tolerant design, which includes the static redundancy scheme for sensors and the dynamic one for controllers and actuators. The following describes the system components in more detail.

- **Voter.** The functionality of the voter changes slightly from the non-dynamic case, since the "address" to which it should deliver its values will depend on which of the redundant controllers is active. Figure 7 shows a Modechart of the modified voter component (the modes within the dotted box handle the change in communication); note that the voter examines state variable *Recon* in order to determine if a reconfiguration is necessary. Also, the design assumes that the voter first attempts to communicate with the main actuator; only when this fails does it examine *Recon* to see if it should communicate instead with the back-up actuator. This set-up ensures that actuator failures can be handled even after communication has been initiated.

- **Monitor.** This component detects actuator faults and initiates system reconfiguration as necessary. We have opted for a simple set-up in which an actuator issues a *failure* event when it fails; in response to this the monitor initiates the rearrangement of the system by setting an appropriate state variable. In a more refined setting the monitor might engage in more sophisticated behavior; it might, for instance, initiate a reconfiguration if an actuator takes too long to respond to a command. Figure 9 contains a Modechart description of the monitor.

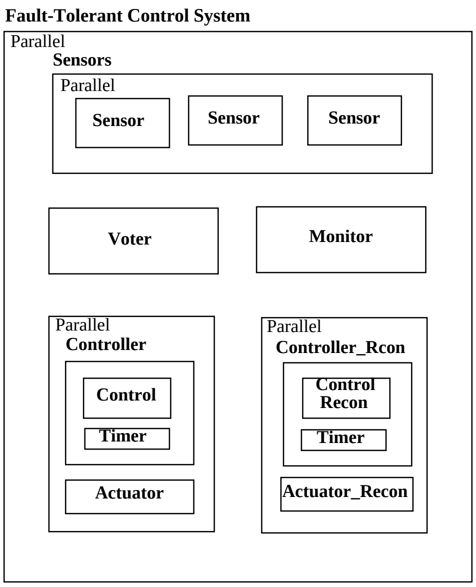

Figure 8: Modechart for Fault-Tolerant System with Static and Dynamic Redundancy.

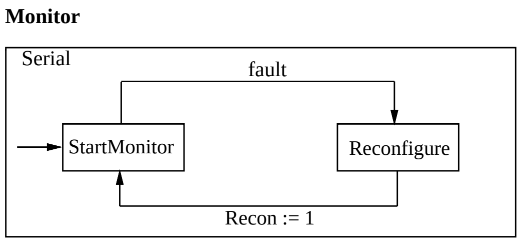

Figure 9: Modechart of the Monitor.

- **Actuators and Controllers.** The Modecharts for the actuator and back-up actuator are exactly the same as the one given in Figure 4. The controller differs only in that it communicates with the voter (via the variable *SynchVoter*) rather than with the sensors directly; the back-up voter uses the variable *SynchVoterRecon* for this purpose.

5 Analyzing the Fault-Tolerant System

In this section we use the Concurrency Workbench to analyze the fault-tolerant system design proposed in the previous section. We have two main aims: we wish to characterize how the inclusion of fault-tolerance mechanisms affects the behavior of the system in the absence of component failures, and we also want to determine how failures affect the timing behavior of the system.

5.1 Compositional State Space Construction

In order to conduct the analyses just described, the Workbench must construct the reachable global state space of the system. The default procedure used by the tool is to build these states in an "on-the-fly" manner, beginning with the initial state and adding states having transitions from states already reached. Unfortunately, the huge size of the state space made this impossible to calculate; even 512MB of memory on a Sun SparcStation-20 proved insufficient to store the state space. The problem is that the system has a huge number of states; the entire global state space (including unreachable states) contains in excess of $2.12 * 10^{19}$ states.

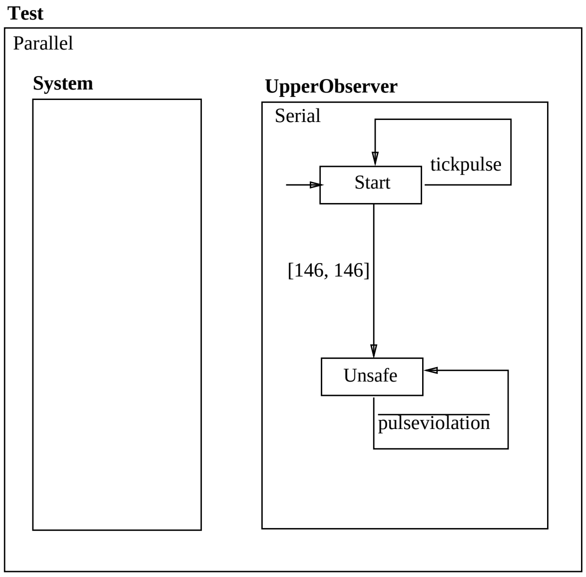

Figure 10: Modechart for Testing $\Delta t_p \leq 145$.

To circumvent this problem we use the minimization feature of the Workbench to build a system that is observationally equivalent to the global system but contains many fewer states. Our approach is incremental and bottom-up, in the following sense: we take parallel modes, compute the combined state space of their children, minimize it, and use it in place of the parallel mode in computing the next stage of the system. The following table summarizes the sizes of the systems that are built as a result.

Mode	Form	Reachable States
Sensors	Sensor | Sensor | Sensor	2432
S_3	min(Sensors)	550
SenVot	S_3 | Voter	14987
S_V	min(SenVot)	604
ContAct	Cont | Actuator	10679
C_A	min(ContAct)	3199
ContActR	Cont | Actuator_Recon	10679
C_{AR}	min(ContActR)	3199
StaticSys	S_V | C_A	21225
SSys	min(StaticSys)	1726
DynamicSys	SSys | C_{AR} | Monitor	433
DSys	min(DynamicSys)	407

On the basis of the properties of the minimization procedure and $\approx$, we know that DSys is observationally equivalent to the original (unminimized) system. Because of this equivalence, we are able to analyze DSys in what follows, as it is is indistinguishable from its larger counterpart.

5.2 Verifying the Model

After having constructed a model of the global state space, we now turn to the task of analyzing the timing properties of the design. We are interested in two questions: how does the inclusion of fault-tolerance affect the normal operation of the system, and what happens to the timing behavior in the presence of faults? We first review how we analyze the timing behavior of systems and then present our results.

5.2.1 Representing Timing Bounds

In [8] we presented an *observer-based* approach to determining whether timing constraints between system events are satisfied. Briefly, the methodology works as follows. Given an active structural control system S that emits

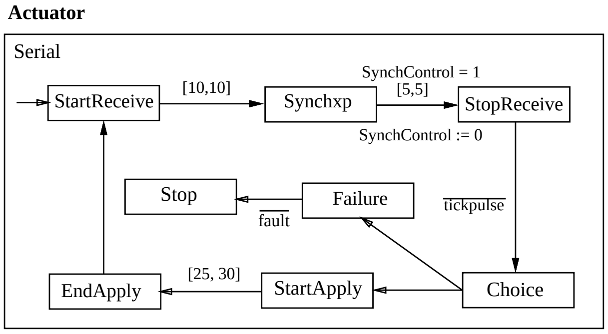

Figure 11: Modechart of a Faulty Actuator.

$\overline{tickpulse}$ events, we would like to see if the upper and lower bounds between successive events are consistent with the bounds implied by the pulse-control algorithm. To do so, we define two observers, one for the lower bound and one for the upper bound, and run them in parallel with the system. The upper bound observer repeatedly attempts to synchronize with the system on the pulse events; if the gap between such events ever exceeds the upper bound, the observer emits $\overline{pulseViolation}$ events. A system therefore meets such a requirement if the observer is never capable of a violation event. The Workbench provides several approaches to determining whether an event is ever enabled in a system; for the purposes of this case study, we use the equivalence-checking feature—in essence, we attempt to establish that the system consisting of the control system and the observer is equivalent to a system that is incapable of violations. Figure 10 contains a Modechart description of this observer-based scenario in which the upper bound is 145 time units. Lower bound analysis is handled in a similar manner, and discussion of it is omitted. Indeed, in the remainder of the paper we focus exclusively on upper bound behavior.

Note that this technique may also be used to calculate the lower and upper bounds between pulses; to do so one repeatedly adjusts the bounds and rechecks the system to see if violations are possible or not.

5.2.2 Fault-Free Behavior

We now analyze the behavior of the fault-tolerant design under the assumption that components do not fail. Our hope was that this system would meet the same timing constraints as the non-fault-tolerant model given in Section 2.2—that is, $\Delta t_p \leq 145$. However, this turned out not to be the case; leaving the timer value at 135 yielded an upper bound between pulses of 166 msec $* 10^{-1}$, and lowering the timer value proved to have no effect on this bound. To determine the source of the additional overhead, we built a design in which the additional actuator was omitted (so only the sensor system was fault-tolerant); this system turned out to be observationally equivalent to the fully fault-tolerant one. This suggests that the source of the additional overhead is in the voting mechanism; more specifically, the bottleneck turns out to be the sequentialized communication between the sensors and the voter.

On the basis of these results, it follows that, even under normal functioning, the fault-tolerant design would not be appropriate for managing the control system for the structure given in Section 2.2. To remedy this deficiency, one would need either to lengthen the natural period of the structure (by, e.g. using different materials in its construction or changing its geometry) or redesign the sensor fault-tolerance mechanism.

5.2.3 Analysis in the Presence of Faults

We now investigate the timing behavior of the system when actuator faults are possible. We consider a simple case in which the main actuator may fail; our goal is to determine the upper bound on Δt_p in this instance.

To model this situation we slightly modify our definition of the actuator; Figure 11 contains the revised Modechart. The only difference is that the actuator may nondeterministically fail at the point at which it begins to apply its pulse; in this case it emits a $\overline{fault}$ event and halts. This event is in turn detected by the monitor, which starts the necessary system reconfiguration by setting the *Recon* state variable.

An analysis of the upper bound on Δt_p reveals that it can be as high as 261 msec $* 10^{-1}$, even when the system timer is set to 0. In essence the reason for this is that when the actuator fails, its replica starts from "scratch"; its controller must resample the sensors, recalculate the pulse magnitude, and send the relevant commands to the actuator. A structure using this design would therefore need to have a period of at least 522 msec $* 10^{-1}$ in order to behave satisfactorily.

6 Conclusions and Future Work

This paper describes a case study that was concerned with the modeling and analysis of a real-time fault-tolerant active structural control. In order to perform the analysis we used the Concurrency Workbench to:

- reduce the state space of the system to a manageable size; and
- check timing bounds of the resulting system.

The former "minimization" feature proved especially valuable, as one of the systems we investigated contained in excess of 10^{19} states. The designs that we analyzed were formulated in an intuitive graphical notation, Modechart, for real-time systems; in order to use the Workbench for analysis, we first translated the designs into Temporal CCS, the language supported by the tool. We believe that our results illustrate a couple of simple yet useful techniques— minimization, verification via "observers"— that real-time system designers may use for modeling and analyzing large and complex real-time systems.

As future work we plan to improve the active structural control design by including more realistic fault-detection and failure-recovery mechanisms, with a view toward developing a design that would be implemented and analyzed as part of a structure in a laboratory setting. We also would like to investigate the use of our modeling and analyzing techniques to other type of complex real-time systems such as intelligent vehicle highway systems. We are also investigating the construction of an interface between the Workbench and MT [6], a tool that includes a Modechart editor, as a means of providing direct support for the analysis of Modechart specifications.

References

[1] M. Abdel-Rohman and H. H. E. Leipholz. Structural control by pole assignment method. *Engineering Mechanics*, 104:1157–1175, 1978.

[2] A. K. Agrawal, Y. Fujino, and B. K. Bhartia. Instability due to time delay and its compensation in active control of structures. *Earthquake Engineering and Structural Dynamics*, 22:211–224, June 1993.

[3] J. W. Baugh, Jr. and W. M. Elseaidy. Verification of real-time software for active structural control. In L. F. Cohn, editor, *Computing in Civil and Building Engineering: Proceedings of the Fifth International Conference*, pages 1672–1679. Technical Council on Computer Practices of the American Society of Civil Engineers (ASCE), 1993.

[4] J. W. Baugh, Jr. and W. M. Elseaidy. Real-time software development with formal models. *Journal of Computing in Civil Engineering*, 9(1):73–86, 1995.

[5] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: A semantics-based tool for the verification of finite-state systems. *ACM Trans. Prog. Lang. Syst.*, 15(1):36–72, January 1993.

[6] P. C. Clements, C. L. Heitmeyer, B. G. Labaw, and A. T. Rose. Mt: Toolset for specifying and analyzing real-time systems. In *IEEE 14th Real-Time Systems Symposium*. IEEE, 1993.

[7] W. M. Elseaidy. *Safety and Reliability of Real-Time Engineering Systems using Formal Methods.* PhD thesis, North Carolina State University, Raleigh, North Carolina, May 1995.

[8] W. M. Elseaidy, R. Cleaveland, and J. W. Baugh. Verifying an intelligent structural control system: A case study. In *15th IEEE Real-Time Systems Symposium*, pages 271–275, 1994.

[9] D. Harel. Statecharts: A visual formalism for complex systems. Technical report, The Weizmann Institute of Science, Israel, July 1986.

[10] F. Jahanian and A. K. Mok. Semantics of Modechart in real time logic. In *21st Hawaii International Conference on System Science*, pages 479–489, 1988.

[11] F. Jahanian and D. A. Stuart. A method for verifying properties of modechart specifications. In *IEEE 9th Real-Time System Symposium*, pages 12–21. IEEE Computer Society Press, 1988.

[12] R. Milner. *Communication and Concurrency.* Prentice-Hall, Englewood Cliffs, NJ, 1989.

[13] A. Moller and C. Tofts. A temporal calculus of communicating systems. In *Proceedings of CONCUR'90*, pages 401–415. Lecture Notes in Computer Science 458, Springer-Verlag, 1990.

[14] A. M. Reinhorn, T. T. Soong, M. A. Riley, R. C. Lin, and M. Higashino. Full-scale implementation of active control. II: Installation and performance. *Structural Engineering*, 119:1934–1961, June 1993.

[15] B. D. Rose and J. W. Baugh, Jr. Parametric study of a pulse control algorithm with time delays. Technical Report CE-302-93, Department of Civil Engineering, North Carolina State University, Raleigh, NC, August 1993.

[16] T. T. Soong. *Active Structural Control.* Longman Scientific, New York, 1990.

[17] J. N. Yang, A. Akbarpour, and P. Ghaemmaghami. New control algorithms for structural control. *Engineering Mechanics*, 113:1369–1386, 1987.