Modeling and Verifying 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, in 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 equivalence-based 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 central insight arising from the study involves the importance of compositionality for reasoning about large and complex systems; in particular, the success of the case study depends integrally on our notation's and tool's support of *component-wise* minimization.

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 applications. Such control systems aim to improve the performance of these applications, 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; in such environments, 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: introducing redundant components into a system can greatly complicate the task of establishing correctness, while reasoning about the system without its fault-tolerant mechanisms often obscures the subtle variations in timing behavior that they can introduce. 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 gain confidence in system performance. Nevertheless, errors still pass unnoticed, with sometimes tragic results. In principle, formal methods offer a system builder the means to prove his system correct and thereby avoid the cost and uncertainty inherent in testing-based validation techniques. In practice, however, the impact of formal methods has remain limited, one main reason being that traditional approaches do not "scale up" to large and complex systems.

Our aim in this paper is to illustrate how a formal design notation for real-time systems may be used in conjunction with a process-algebra-inspired notion of *semantic equivalence* and an automatic verification tool to reason about the timing behavior of finite-state, real-time control systems. More specifically, we show how one 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.

- use a simple equivalence-based technique for establishing that a system meets constraints on its timing behavior; and
- *minimize* systems with respect to the equivalence in a compositional, component-wise fashion in order to reduce the size of the state space that must be considered.

We develop these themes in the context of a case study involving the analysis of a fault-tolerant extension of an experimental *active structural control* architecture from the civil engineering field; the system is described and formalized in [6, 8, 4, 7]. As this system has in excess of 10^{19} states, some means of state-space reduction is absolutely necessary in order for the analysis to be performed automatically. For these purposes, compositional minimization proves to be very effective. Throughout the case study we use a fully automated verification tool, the Concurrency Workbench [5], to calculate the semantic equivalence and to minimize systems with respect to it.

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, defines the semantic equivalence that forms the basis of our formal methodology, and gives a brief overview of the Concurrency Workbench. Section 4 then describes the fault-tolerant system to be analyzed, and Section 5 presents the results of our analysis of its behavior. The final section closes with a discussion of the practical import of our work and of directions for future research.

Related Work. Other researchers have conducted case studies involving the automated formal analysis of the timing behavior of real-time systems. Most of the examples involve the use of dense time models, which require more sophisticated modeling techniques than those mentioned in this paper; the interested reader is referred to [11], which contains the analysis of several different systems. Case studies involving the simpler discrete model of time used here have been less well-studied, although [9] does present the analysis of a simple communications protocol with real-time and probabilistic behavior.

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 winds. Typically, such structures include length-adjustable members, or *actuators*, that may be expanded or contracted to counteract the external forces applied to the structure. A process controller monitors sensors that measure the state of the structure 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 [21]. 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. Thus, to minimize vibration-induced earthquake damage, the natural frequencies of a structure should be located outside the frequency band of the seismic excitations produced by earthquakes. 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 length-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 an "end-on" view of a six-story, actively controlled building in Tokyo that was built to test the feasibility of active structural control systems in earthquake resistance. The structure features six sensors and four active members connected to the first floor [19] as indicated in the figure.

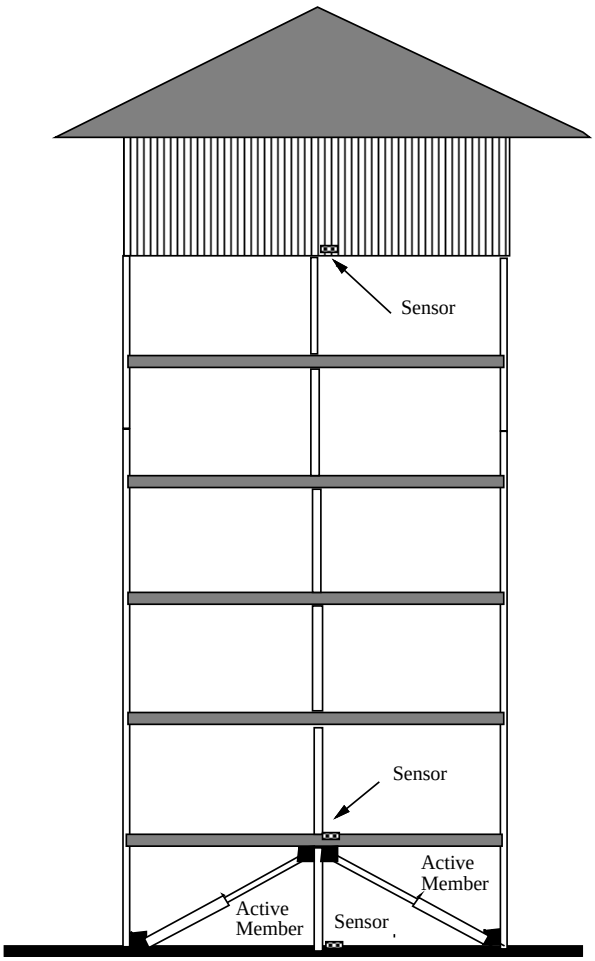

Figure 1: An Actively Controlled Structure in Tokyo.

2.1 Pulse-control Algorithms

Several different control techniques have been proposed for active structures in the literature [1, 22]. 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 the durations between successive pulse applications. Traditional pulse-control theory requires that the interpulse delays be of exactly the same value, namely, Δt_p; when this is the case, one may use the results in [18, 21] for selecting this value so that vibration is dissipated correctly. In real structures, however, ensuring that these delays are invariant is virtually impossible, owing to "play" in factors such as the time needed for calculating pulse magnitudes and the ramp-up times for actuators. On the other hand, a parametric simulation-based

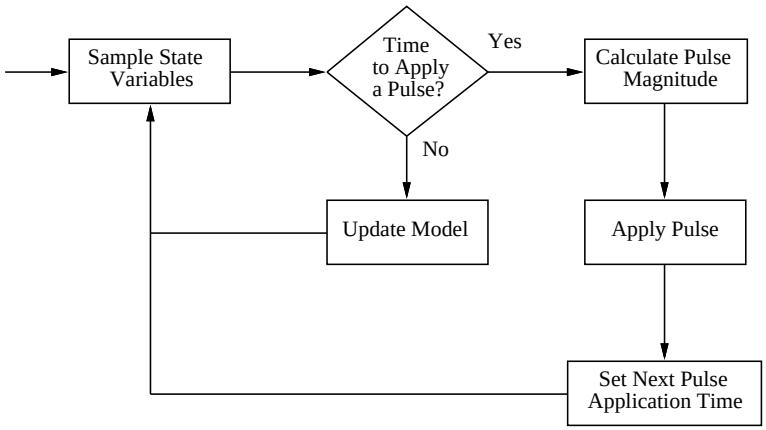

Figure 2: Block Diagram of a Pulse-Control Algorithm.

study [20] indicates that some variability in duration times is allowable, depending on the chosen Δt_p and structural characteristics such as natural frequency and geometry. These results form the basis for the constraints that we impose on our system designs.

2.2 An Experimental System

In [21] an experimental set-up is described for an actively controlled single-degree-of-freedom structure. The same structure was used in the parametric study of [20]; when Δt_p was taken to be 135 msec $* 10^{-1}$, the structure was found to behave correctly if the interpulse delays were at least 37 msec $* 10^{-1}$ but no more than 145 msec $* 10^{-1}$. In the remainder of this paper we use these figures as the timing constraint that our designs should satisfy.

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 summarizes the (experimentally determined) timing information 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 socket-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.

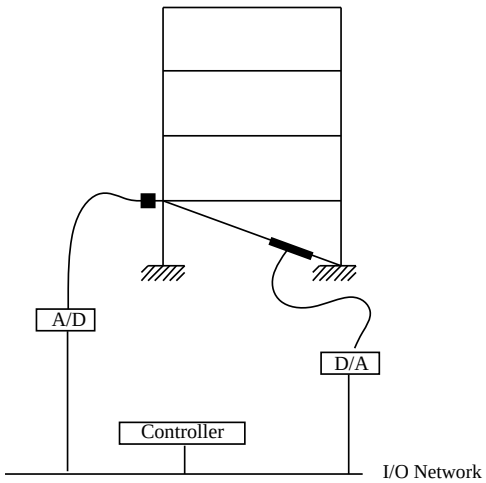

Figure 3: An Experimental Active Structural Control System.

3 Formal Design Support

This section presents the formal notations and the verification tool used in this paper. More specifically, it describes:

- a graphical design language, Modechart, that we use to represent our designs;
- a process algebra based on Temporal CCS [16] and a behavioral equivalence relation, observational equivalence, for relating systems expressed in it; and
- the Concurrency Workbench, a tool that includes support for calculating whether Temporal CCS systems are equivalent and for minimizing them.

In what follows we first develop our designs using Modechart; we do this because the graphical notation has an intuitive semantics and is therefore easy to use. However, notions of semantic equivalence have not been explicitly developed for Modechart, while they have for Temporal CCS. Accordingly, we then translate our designs in a structure-preserving manner into this language and then use the Workbench to conduct our reasoning.

3.1 Modechart

Modechart [13, 14] extends the graphical design language Statecharts [10] with constructs for modeling real-time behavior. The notation we describe here is a simplified variant of the full language, a more detailed account of which may be found in [13]. In Modechart, systems are represented as finite-state machines; in line with its connection to Statecharts, however, the states in these machines may have other finite-state systems embedded in them. More precisely, the language includes two basic constructs: *modes*, which may be thought of as "structured states" that may contain subsystems, and transitions. We describe each of these in turn.

Modes correspond to states in finite-state machines; the point of departure for these two notions is that modes may contain embedded subsystems. Intuitively, when a system enters a particular mode it initiates execution of the mode's subsystems, if it has any; when the system leaves a mode, the corresponding

subsystems are "interrupted". More formally, Modechart contains three kinds of modes: *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 "submodes" and are intended to capture the notion of sequential and parallel execution, respectively. Serial modes must also have a submode designated as initial. When a serial mode is entered, its initial submode is also entered at the same time; when a parallel mode is entered, each of its submodes is entered simultaneously.

Modechart systems change modes by engaging in transitions between modes. In our variant of Modechart transitions consist of three components: a *condition*, a *timing constraint* of the form $[l, u]$, and an *effect*. Intuitively, a transition becomes enabled when control resides in its "source mode" and its condition becomes true; it may then fire any time between l and u time units after it is enabled. When a transition fires, control passes to its "target mode" and the state of the system is updated as indicated by the effect, which typically alters variables declared by the system designer. If two or more transitions are capable of firing at the same time the choice as to which occurs is made nondeterministically. For the purposes of this case study a condition will consist either of the testing of a finitely-valued state variable or of the detection of the occurrence of a (user-defined) event elsewhere in the system. An effect will either be an assignment to a variable or the generation 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 it is assumed to be $[0, 0]$. A missing effect is assumed to be a "no-op". The formal semantics of Modechart is given via a translation into the logic RTL [13]; the translation essentially encodes the transition firing rule described informally above.

Figure 4 gives a Modechart rendering of the active structural control system described in the previous section. The overall system consists of a single parallel mode containing three submodes: 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 sensor and actuator modes are serial. The former contains five submodes, with $\downarrow SampleState$ designated as initial, while the latter also contains five submodes, with $\uparrow ReceivePulse$ initial. [1] When the system begins execution, control resides in the initial submodes—$\uparrow SampleState$, $\uparrow Receive$, *Reset*, and $\uparrow ReceivePulse$— of each of the four serial subcomponents. From this starting point, the transition firing rule described above governs the configurations the system may enter; for example, after 10 time units transitions in modes *Control* and *Actuator* would fire, and the submodes would change to $Synch_{xs}$ and $Synch_{xp}$, respectively, as the two corresponding system components prepare to receive data.

The system design includes mention of three boolean variables: *SynchSensor*, V and *SynchActuator*. These are used to synchronize the behavior of the different components appropriately. For example, when the timer is in mode *Time*, it remains there until variable V is set to 0; 135 time units later, it changes mode to *Reset*, whence it instantaneously "goes off" by setting V to 1 and then returning to *Time*. To "set" the timer, then, the controller need only set V to 0. Once set, however, the timer cannot be reset until it expires. In a similar fashion, 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 generation of the $\overline{tickpulse}$ event allows one to observe the beginning of pulse applications.

3.2 Temporal CCS

Modechart provides a convenient and intuitive formal notation for describing hierarchically organized real-time systems. However, verifying designs given in Modechart can be still be a formidable task, owing to

[1] The names inside the primitive modes are included for explanatory purposes only. We adopt the convention that $\uparrow$ and $\downarrow$ indicate the beginning and ending of different system activities. For example, when the Sensor submode enters mode $\uparrow SampleState$ the mode name indicates that the component is beginning its sampling phase; this ends between 50 and 55 time units later, when the Sensor enters the mode $\downarrow SampleState$.

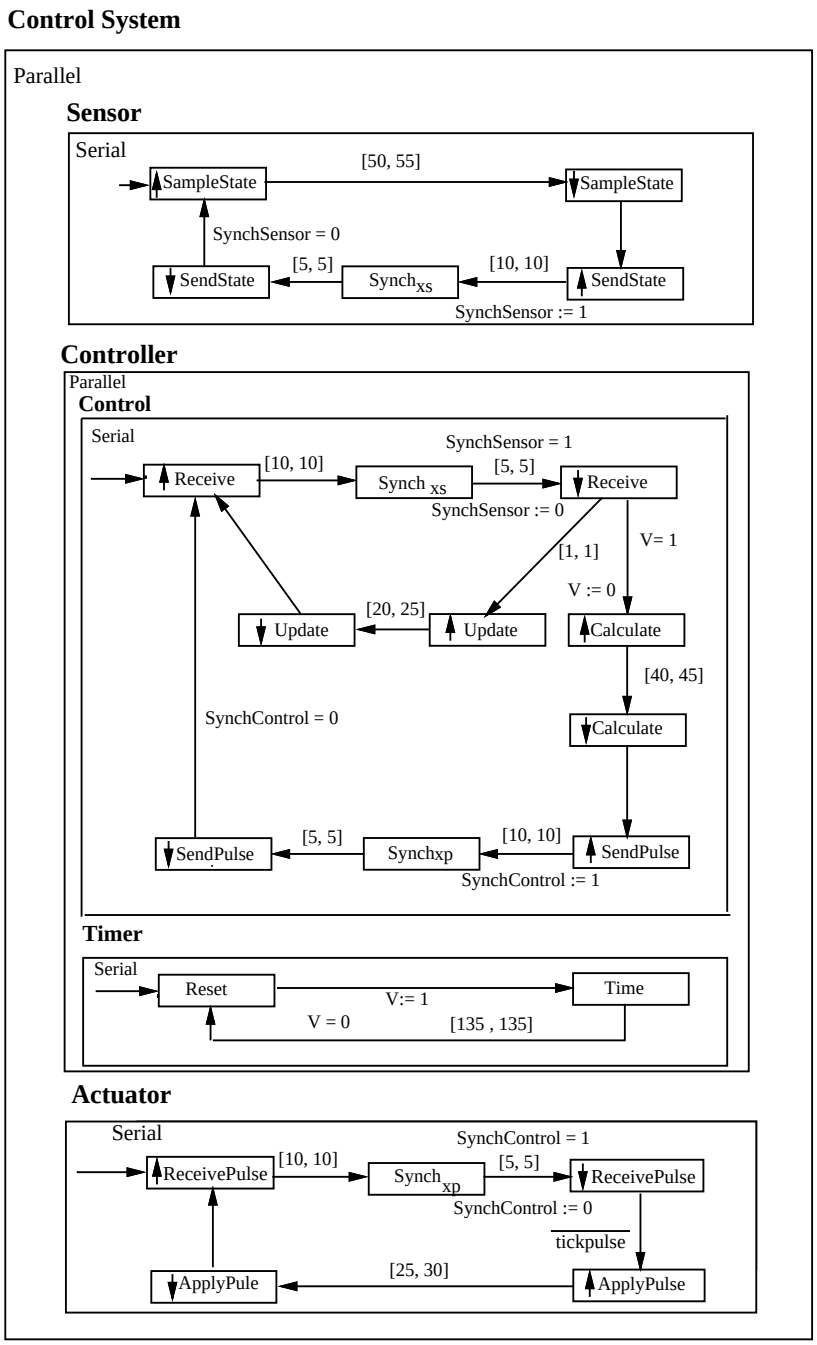

Figure 4: Modechart Model of a Control System.

the state-explosion problem: the number of global states in a system can be exponential in the number of parallel components. In the absence of a scheme for reducing the size of state spaces, state-exploration-based verification rapidly becomes impractical in such a setting.

Process algebras [3, 12, 15] provide one approach to handling state-explosion that is based on the use of compositionality (i.e. component-wise analysis) and semantic equivalence. Such formalisms typically contain a language consisting of a small number of constructs for building systems and a formally defined semantic equivalence that indicates when two processes "behave the same" and may be used interchangeably. In what follows we describe one such process algebra and show how Modechart systems may be translated into it. This translation enables one to apply process-algebraic "technology" to the analysis of Modechart systems.

The process algebra we use is based on Temporal CCS [16], which extends Milner's Calculus of Communication Systems (CCS) [15] with features for describing timing behavior. The language provides operators 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, processes in our algebra may be built using the following grammar.[2]

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

Intuitively, these constructs may be understood in terms of the communication actions and units of delay (or idling) they may engage in. Non-delay actions are assumed to be durationless; so time passes only when processes are capable of "idling". *nil* represents the stopped process; it is incapable of any action or idling. X may be thought of as an invocation of the process currently "bound" to X. If $a \notin Nat$ then $a.P$ is a process that immediately engages in action a and then behaves like P; it is incapable of idling. If $a \in Nat$ then $a.P$ idles for a time units and then behaves like P. The $+$ construct represents choice; $P_1 + P_2$ may engage in any non-delay actions that either P_1 or P_2 is capable of and then behave like the process whose action was performed. If both P_1 and P_2 can idle then $P_1 + P_2$ may also idle without discarding either component. On the other hand, if one of the P_i can idle and the other cannot, then $P_1 + P_2$ discards the "impatient" process when it idles. $P_1|P_2$ represents the parallel composition of P_1 and P_2. For the composite system to idle, both components must be capable of idling. Non-delay actions are executed in an interleaved fashion; moreover, if either P_1 or P_2 is capable of an output on a channel that the other is capable of an input on, then a synchronization occurs, with both processes performing their actions and a τ resulting. If $L \subseteq \Lambda$ then $P\backslash L$ defines a process in which the channels in L may be thought of as "local". Finally, in processes of the form $P \; where \; X_1 = P_1 \ldots X_n = P_n \; end$ the equations $X_i = P_i$ provide the definitions of occurrences of X_i in P.

As is the case with Temporal CCS [16], the semantics of our algebra 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. This property is also shared by Temporal CCS; indeed, the only substantive difference between our language and Temporal CCS is that to perform the modeling in this paper, we found it necessary to add *maximal progress* to the semantics. 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. Given the similarity between Temporal CCS and our language, in what follows we refer to our language as Temporal CCS also.

The semantics forms the basis for generating (finite-) state machines from process algebra expressions, with the transitions in the machine recording the execution steps of the system.

[2] Temporal CCS contains these constructs as well as others that we omit here, since we do not need them in our modeling.

3.2.1 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. Verifying a proposed design then amounts to showing that it is semantically equivalent to its specification. One well-studied semantic relation is *observational equivalence*, 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. The equivalence may now be defined as follows.

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. For the purposes of this paper, the crucial properties enjoyed by $\approx$ are the following.

- It is an equivalence relation.
- It is a *congruence* for all the operators in (our version of) Temporal CCS except $+$. That is, if $P \approx Q$ then P and Q may be used interchangeably inside any Temporal CCS context that does not use the $+$ operator.[3]
- It is *decidable* for finite-state systems.
- Finite-state systems may be *minimized* with respect to it. That is, for any finite-state system M, there is a unique (up to state renaming) minimum-state finite-state system $min(M)$ such that $M \approx min(M)$.

For finite-state systems, $\approx$ may be computed in time that is polynomial in the number of states and transitions of the system [17].

3.2.2 Translating Modechart to Temporal CCS

We now describe how (a subset of) of Modechart may be translated into our version of Temporal CCS. The methodology presented here assumes the following restrictions on Modechart.

- Only primitive modes have outgoing transitions.
- An event can appear in the outgoing transition conditions of at most one serial mode.

The second condition allows the direct modeling of Modechart events as Temporal CCS actions, with the "generation" of an event e being modeled as Temporal CCS action $\overline{e}$ and the condition e being modeled as the action e. Removing this restriction is not too difficult, although it complicates the translation somewhat. Removing the first, however, would necessitate the addition of operators to Temporal CCS in order for the structure of a translation to mimic the structure of the Modechart from which it was generated. For the purposes of this paper, this fragment of Modechart provides sufficient expressive power.

[3] A slightly finer equivalence relation that is a congruence for all Temporal CCS operators may be defined using standard techniques. For the purposes of this paper, however, it is unnecessary.

To translate Modechart into Temporal CCS, we first must describe how to model state variables. Following traditional practice in process algebra, we model such variables using processes, one for each variable. The process for a variable may be thought of as a server; other processes wishing to manipulate the value of the variable send "requests" to the server, which changes state accordingly. More specifically, suppose V is a variable that may assume values 0 and 1 and whose initial value is 0. Then the Temporal CCS model of V would be the following.

$$
\begin{array}{l}
V \ where \\
\quad \begin{array}{lcl}
V & = & V0 \\
V0 & = & \overline{rV0}.V0 + wV0.V0 + wV1.V1 + 1.V0 \\
V1 & = & \overline{rV1}.V1 + wV0.V0 + wV1.V1 + 1.V1
\end{array} \\
end
\end{array}
$$

Intuitively, V is in state $V0$ when the value of the variable V is 0 (or false), and in $V1$ when its value is 1. A process that wishes to change the value of V to 1 would execute action $\overline{wV1}$ (for "write 1 to V"); a process that wants to behave like P if $V = 0$ could be given as $rV0.P$ (where $rV0$ stands for "read 0 from V"). Note that in either of its states, V is capable of idling.

We now describe how to translate serial and parallel modes. Renderings of serial mode S will have the general form

$$
\begin{array}{l}
S \ where \\
\quad \begin{array}{lcl}
S & = & S1 \\
S1 & = & T1_1 + \cdots + T1_{k_1} \\
 & \vdots & \\
Sn & = & Tn_1 + \cdots + Tn_{k_n} \\
T1_{k_1} & = & \ldots \\
 & \vdots & \\
Tn_{k_n} & = & \ldots
\end{array} \\
end
\end{array}
$$

where $S1, \ldots, Sn$ are the submodes of S, $S1$ is the initial mode, and the Ti_j are the transitions emanating from mode Si. Note that each transition has its own equation.

We know turn to the translation of individual transitions. Let T be a transition from mode Si to Sj. T has three components:

- A condition C, which may be either a test of a variable V or the presence of an event e.
- A time constraint $[l, u]$.
- An effect A to perform upon executing the transition. A may be either the generation of an event e' or the setting of a variable V'.

Each of these components is translated in turn. We use Temporal CCS input actions to represent conditions. Define the action a_C to be rVi if C involves testing whether $V = i$ and e if C involves testing for the presence of event e. By convention, we say that a condition is true if the environment of the mode offers the action $\overline{a_C}$; when this is the case, a_C and $\overline{a_C}$ may synchronize and produce a τ. We may now give the following partial of the translation of T, where $P1, P2$ and $P3$ have translations of the different components of T.

$$
\begin{array}{l}
T \ where \\
\quad \begin{array}{lcl}
T & = & P1 \\
P1 & = & a_C.P2 + 1.P1 \\
P2 & = & \ldots \\
P3 & = & \ldots
\end{array} \\
end
\end{array}
$$

Note that $P1$ idles by performing 1-transitions until the environment offers the action $\overline{a_C}$; then, because of the maximal progress assumption, no more idling can occur until a_C and $\overline{a_C}$ synchronize. If condition C is absent then we generate the equation $P1 = P2$.

To translate the timing constraint of T, recall that we must idle for at least l but no more that u time units, with the choice being made nondeterministically. We model this by introducing $u - l$ new equations into the translation with left-hand sides $P2_i$ for $1 \leq i \leq u - l$. The intuition is that $P2$ will delay for l time units and evolve to $P2_1$. Then each $P2_i$ may nondeterministically elect to enter $P3$ or delay one time unit and evolve to $P2_{i+1}$ (or $P3$ in the case of $P2_{u-l}$); the nondeterminism is modeled using τ-transitions. We thus have the following.

$$\begin{array}{lcl} P2 & = & l.P2_1 \\ P2_1 & = & \tau.P3 + \tau.1.P2_2 \\ & \vdots & \\ P2_{u-l} & = & \tau.P3 + \tau.1.P3 \end{array}$$

Note that if l is 0 then we take $P2 = P2_1$ and that if u is also 0 then we take $P2 = P3$.

To complete the translation of T, we now describe how to model the execution of the effect A that is performed as T is taken. If A is an assignment of i to V' then we use the following.

$$P3 \;=\; \overline{wVi}.Sj$$

Otherwise, if A is the generation of event e, then we use:

$$P3 \;=\; \overline{e}.Sj + 1.Sj$$

(Recall that Sj is the destination mode of T.) The $1.Sj$ summand is necessitated by the fact that if no process "consumes" event e in the current time instant, then it should disappear in the next time instant.

Now suppose S is a parallel mode with submodes $S1, \ldots, Sn$. The Temporal CCS translation for S would be $P1|\cdots|Pn$, where Pi is the translation of Si.

Finally, the translation of an entire Modechart M meeting the restrictions mentioned previously may be given as $(Var|P_M)\backslash L$, where Var consists of the parallel composition of all the variable processes, P_M is the translation of the mode structure of M, and L contains the set of events that appear in conditions of transitions or are used to read and write from variables. Thus the only actions this Temporal CCS process will engage in are idling actions, τ-actions, and events that are generated but which are not used in conditions. (These latter events constitute the behavior that the user of the system would observe.)

We close by remarking on the faithfulness of our translation procedure. Ideally, one would prove it correct by establishing a correspondence between a Modechart and its associated Temporal CCS translation. Doing so, however, is complicated by the radically different way in which the semantics of each language is defined, and thus is beyond the scope of this paper. However, we note that the translation of each transition closely follows the informal operational description of transitions given in [14]. As an example of the "output" of the translation procedure, the following is the Temporal CCS rendering of the control mode in the Modechart in Figure 4; we have cleaned it up slightly by removing unnecessary equations.

$$\begin{array}{l} Control = StartReceive \ where \\ \qquad StartReceive = 10.Synch_{XS} \\ \qquad Synch_{XS} = rSynchSensor1.5.\overline{wSynchSensor0}.StopReceive + 1.Synch_{XS} \\ \qquad StopReceive = rV1.StartCalculate + 1.StartUpdate \\ \qquad StartUpdate = rV0.20.Update1 \\ \qquad Update1 = \tau.StopUpdate + \tau.1.Update2 \\ \qquad Update2 = \tau.StopUpdate + \tau.1.Update3 \\ \qquad Update3 = \tau.StopUpdate + \tau.1.Update4 \\ \qquad Update4 = \tau.StopUpdate + \tau.1.Update5 \end{array}$$

$$Update5 = \tau.StopUpdate + \tau.1.StopUpdate$$
$$StopUpdate = StartReceive$$
$$StartCalculate = \overline{wV0}.40.Calculate1$$
$$Calculate1 = \tau.StopCalculate + \tau.1.Calculate2$$
$$Calculate2 = \tau.StopCalculate + \tau.1.Calculate3$$
$$Calculate3 = \tau.StopCalculate + \tau.1.Calculate4$$
$$Calculate4 = \tau.StopCalculate + \tau.1.Calculate5$$
$$Calculate5 = \tau.StopCalculate + \tau.1.StopCalculate$$
$$StopCalculate = StartSendPulse$$
$$StartSendPulse = 10.\overline{wSynchControl1}.Synch_{XP}$$
$$Synch_{XP} = 5.StopSendPulse$$
$$StopSendPulse = rSynchControl0.StartReceive + 1.StopSendPulse$$
$$end$$

3.3 The Concurrency Workbench

The Concurrency Workbench [5] supports the simulation and verification of concurrent finite-state systems expressed in Temporal CCS. In addition to facilities for simulation and various forms of state-space exploration, the tool includes routines for calculating whether or not two systems are observationally equivalent and for minimizing a system with respect to observational equivalence. It should be noted that we needed to alter the system in order to support our version of Temporal CCS, since its semantics differs somewhat from standard Temporal CCS. The change required the modification of fewer five lines of code.

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, designing a system to withstand component faults is essential for 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 transient failures 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.

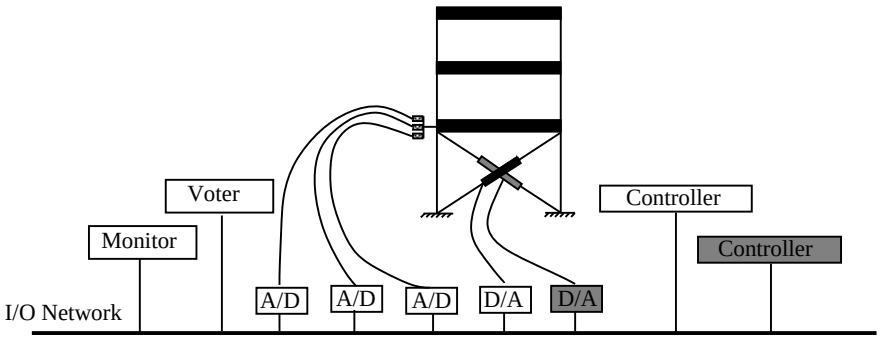

Figure 5: Schematic Drawing of Fault-Tolerant Active Structural Control System.

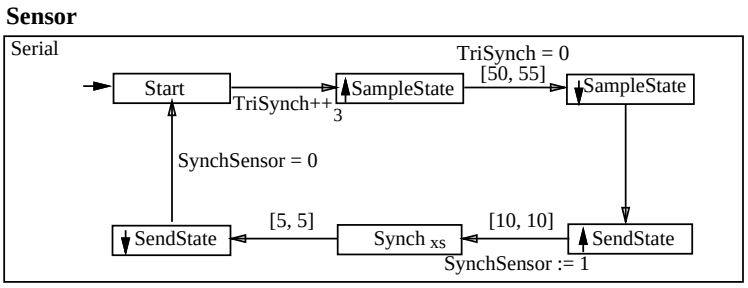

Figure 6: Modechart for a New Sensor.

- **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 state variable $TriSynch$ is used to enforce the three-way synchronization mechanism. The 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, we would want the system to 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).

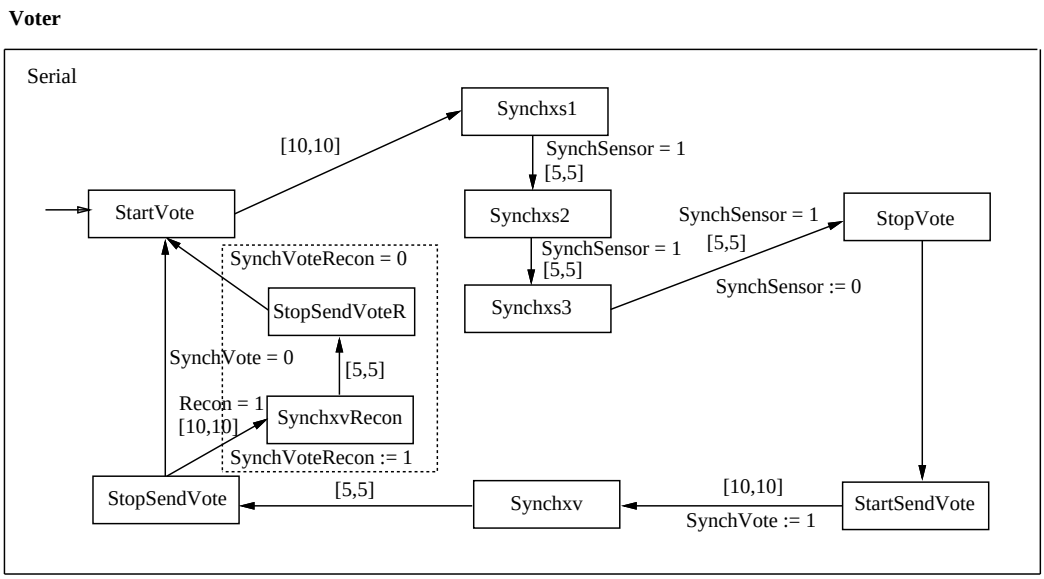

Figure 7: Voter for a Static Redundant System (Modes in Dotted Box May Be Omitted).

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, then, redundant components are 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, spark a reconfiguration if an actuator takes too long to respond to a command. Figure 9 contains a

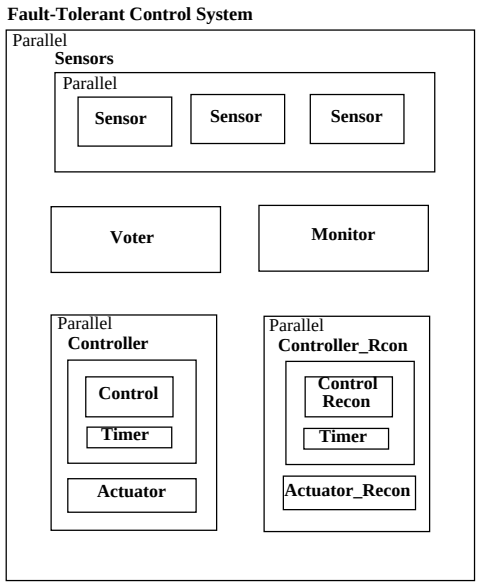

Figure 8: Modechart for Fault-Tolerant System with Static and Dynamic Redundancy.

Modechart description 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 instead.

We close this section with a remark concerning the communication scheme used in the new system design. One the basis of Figure 5 one might conclude that a single bus is used to handle all communication among system components and then wonder why this is not replicated. In fact, however, our modeling is abstract with respect to the specific communication architecture used; the only assumptions we make about the I/O infrastructure involves the timings involved in the transfer of data. A more detailed design might in fact commit to a specific communication framework, in which case issues involving networking fault-tolerance would need addressing.

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 the system in the absence of component failures, and we also would like 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"

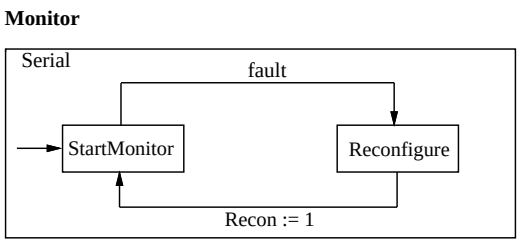

Figure 9: Modechart of the Monitor.

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. In fact, it turns out that the system contains in excess of $2.12 * 10^{19}$ global states (some of which are unreachable, it should be noted).

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 compositional and bottom-up, in the following sense. We take parallel components, compute their combined state space, minimize the result, and use it in place of the parallel components in computing the next stage of the system. The following table summarizes the sizes of the systems that were 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 Analyzing the Model

After constructing 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 show how we analyze the timing behavior of systems and then present our results.

5.2.1 Representing Timing Bounds

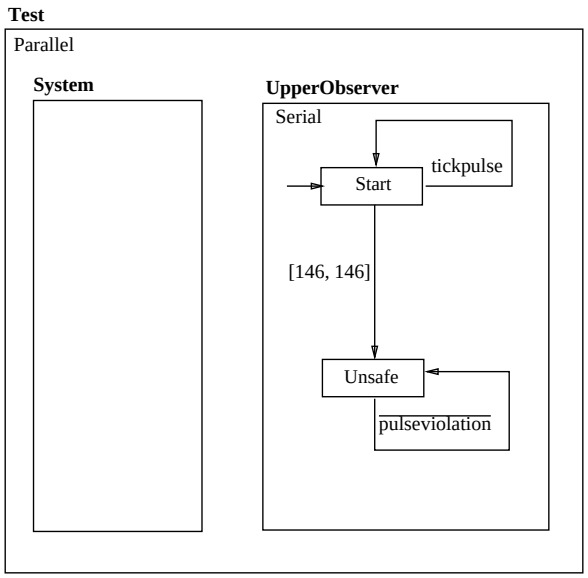

Figure 10: Modechart for Testing $\Delta t_p \leq 145$.

Our approach to determining whether whether timing constraints between system events are satisfied is also based on the use of $\approx$; it works as follows. Given an active structural control system S that generates $\overline{tickpulse}$ events, we would like to see if the upper and lower bounds between successive events are consistent with the bounds dictated by the pulse-control algorithm. To do so, we define two *observer processes*, 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. To determine if this is the case, 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: $\Delta t_p \leq 145$. However, this turned out not to be the case; leaving the timer value at 135 yielded an upper bound between pulses of 166 msec $* 10^{-1}$ (this upper bound was determined by varying the timing value in the observer until no violation event was emitted), 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 subsystem was fault-tolerant); this system turned out to be observationally equivalent to the fully fault-tolerant one under the assumption of fault-freedom. This suggests that the source of the additional overhead is in the voting mechanism; more specifically, the

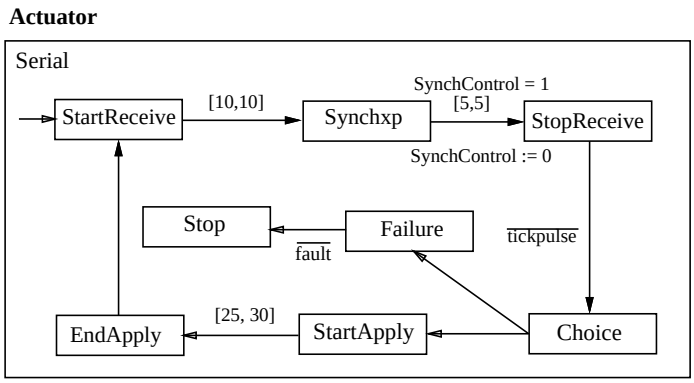

Figure 11: Modechart of a Faulty Actuator.

bottleneck turns out to be the sequentialized communication between the sensors and the voter.

On the basis of these results, it follows that, even in the absence of failures the fault-tolerant design would not be appropriate for managing the control system for the structure given in Section 2.2. In the next subsection we discuss different approaches to overcoming this problem.

5.2.3 Analysis in the Presence of Faults

We now investigate the timing behavior of the system when actuator faults are possible. Our goal is to determine the upper bound on Δt_p in this situation.

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.

5.3 Design Alternatives

In this section we have seen that the proposed fault-tolerant design cannot be used in place of the simpler non-fault-tolerant one; even in the absence of faults the system fails to meet the necessary upper time bound. We now turn to a discussion of different design alternatives that can overcome this problem.

One approach would be to alter the natural period of the structure being controlled. In many cases this alternative is practical, since by varying the materials used in construction and the geometry of the structure one may significantly alter this quantity.

Another approach would be to change the architecture of the control system itself. For example, we identified the sequentialized communication between the sensors and the voter as a reason for the worsened timing behavior of the fault-tolerant system under normal operation. One possible remedy for this would be to allow the controller to communicate simultaneously with all three. To check whether this would fix the sensor/voter bottleneck, one could alter our design and use the Concurrency Workbench to analyze the

timing behavior of the new system. Unfortunately, space constraints prevent us from doing this. In the case of the redundant actuator, one could imagine having the replica "shadow" the behavior of the main device as a means of alleviating the change in timing behavior resulting from our fault-tolerance scheme. That is, each controller would receive data from the sensors and maintain a current model of the system; only the "active" actuator would actually undergo length adjustments, however.

In practice, which design alternative one would pursue would depend on a number of criteria, including the cost and capabilities of available equipment and materials, the flexibility in the design of the structure being controlled, and the particular expertise of the engineers employed on the project. In general, engineers are familiar with such trade-offs; our analysis framework may be seen as giving them more data on which to base their design decisions.

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 features supported by the Concurrency Workbench to formulate and check timing bounds and to reduce the state space of the system to a manageable size in a compositional manner. The latter utility proved especially valuable, as one of the designs 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.

Although the paper has focused on active structural control, the structure of the pulse-control algorithm ("sample/compute/act") matches that of real-time control applications found in other engineering disciplines. Our system was also realistically sized— by way of comparison, recall that the structure in Figure 1 contained four actuators and six sensors— and used experimentally-derived data for its timing information. Consequently, we believe that our results illustrate a couple of simple yet useful techniques— minimization, verification via "observers"— that system designers may use for modeling and analyzing real-time process controllers. In addition, the fact that the formal analysis was automated suggests that these techniques may be useful in an industrial setting, where system builders usually do not have the time to construct laborious proofs of correctness manually. Finally, the time required for the analysis was on the order of minutes on a workstation; consequently, for systems of similar scale our methodology may be used by system designers to experiment with different design alternatives without having to build prototypes or do extensive simulation.

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.

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. Baeten and W. Weijland. *Process Algebra*, volume 18 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, England, 1990.

[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 Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.

[6] 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.

[7] W. M. Elseaidy, J. W. Baugh Jr., and R. Cleaveland. Verification of an active control system using temporal process algebra. *Engineering with Computers*, 12:46–61, 1996.

[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] H. Hansson. Modeling timeouts and unreliable media with a timed probabilistic calculus. In K. Parker and G. Rose, editors, *Formal Description Techniques, IV*, pages 67–82, Sydney, Australia, 1991. IFIP TC6/WG6.1, North-Holland.

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

[11] T. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: The next generation. In 16^{th} *Annual IEEE Real-Time Systems Symposium*, pages 56–65, Pisa, Italy, Dec. 1995. IEEE Computer Society Press.

[12] C. Hoare. *Communicating Sequential Processes*. Prentice-Hall, London, 1985.

[13] 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.

[14] 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.

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

[16] 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.

[17] R. Paige and R. Tarjan. Three partition refinement algorithms. *SIAM Journal of Computing*, 16(6):973–989, December 1987.

[18] Z. Prucz, T. T. Soong, and A. Reinhorn. An analysis of pulse control for simple mechanical systems. *Dynamic Systems, Measurement, and Control*, 107:123–131, Jun 1985.

[19] 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.

[20] 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.

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

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