Verifying an Intelligent Structural Control System: A Case Study*

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

Abstract

This paper describes the formal verification of the timing properties of the design of an intelligent structural control system using the Concurrency Workbench, *an automatic verification tool for finite-state processes. The high-level design of the system is first given in Modechart, a graphical specification language for real-time systems, and then translated into a temporal process algebra supported by the Workbench. The facilities provided by this tool are then used to analyze the system and ultimately show it correct.*

1 Introduction

In this paper, we report on a case study involving the application of a design notation, Modechart [3, 4], and an automatic verification tool, the Concurrency Workbench [1], to the design of a distributed real-time system for active structural control. Our goal in this exercise is to investigate the utility of automatic verification tools in the context of an actual application. The remainder of the paper is structured as follows. In the next section we define more precisely the issues involved in active structural control. The section following then reviews Modechart and Temporal CCS, the real-time language supported by the Concurrency Workbench, and describes how one may translate Modechart to Temporal CCS. Section 3 presents a formal description of our real-time active structural control system using Modechart and Temporal CCS, while the next section shows how the constraints it must satisfy may be represented and verified using the Workbench. We conclude by remarking on our experiences in verifying the system and on directions for future work.

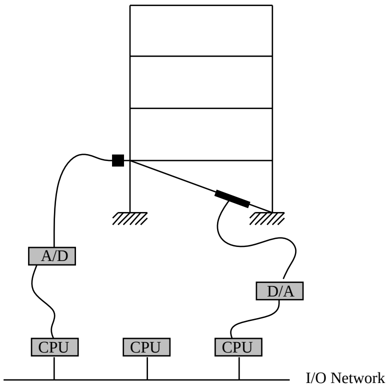

Figure 1: An Intelligent Structural System

2 An Intelligent Structural Control System

This section describes a simple distributed real-time system for active structural control. Figure 1 presents an overview of the system, which contains three major components: a sensor, which monitors the state of the system (e.g. accelerations and displacements); an actuator, or active member, which applies forces to the structure; and a control process, which feeds the data provided by the sensor to a control algorithm that calculates the appropriate forces the actuator device should provide to counter external excitation of the structure.

A variety of different control strategies for active structures have been proposed in the literature. In this case study we use a *pulse control algorithm* [8] for single-degree-of-freedom systems subjected to base accelerations. [1] To determine the appropriate pulse magnitude to apply at a given interval, pulse control algorithms measure the state variables of the system (e.g., displacements, velocities, and/or accelerations) and, if an appropriate

*This paper will appear in *11th Real-Time Systems Symposium*, San Juan, Puero Rico, December 1994.

†Research Assistant, Department of Civil Engineering, N.C. State Univ., Raleigh, NC 27695-7908. E-mail: `wmelseai@eos.ncsu.edu`.

‡Associate Professor, Department of Computer Science, N.C. State Univ., Raleigh, NC 27695. E-mail: `rance@csc.ncsu.edu`. Research support provided by by NSF/DARPA grant CCR-9014775, NSF grant CCR-9120995, ONR Young Investigator Award N00014-92-J-1582, and NSF Young Investigator Award CCR-9257963.

§Associate Professor, Department of Civil Engineering, N.C. State Univ., Raleigh. NC 27695-7908. E-mail: `jwb@eos.ncsu.edu`. Research support provided by NSF grant MSS-9201687.

[1] The system given in Figure 1 can be modeled as a single-degree-of-freedom using standard techniques [8].

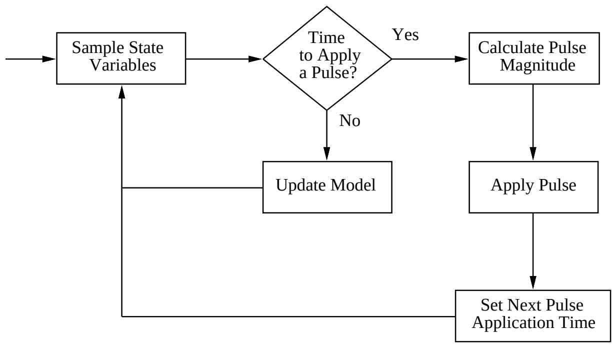

Figure 2: Block Diagram of a Pulse Control Algorithm

amount of time has elapsed since the last pulse application, instruct the actuator to emit another pulse. Figure 2 gives a flowchart describing the general structure of such an algorithm.

Results in [7] bound the variation in periods between pulses as follows. Let T_0 be the natural period of a structure. Then a pulse control algorithm will provide satisfactory performance if the time Δt_P between pulses is bounded by $T_0/8$ and $T_0/2$.

3 System Design

This section describes the two design notations for real-time systems that we use in presenting the design of the active structural control system.

- **Modechart** [4] provides an intuitive and convenient graphical notation for representing hierarchical systems with timing constraints.
- **Temporal CCS** [6] is closer in spirit to actual implementation languages, since it is equipped with an operational semantics and can model constructs that one might find in a real-time programming language.

The operational aspect of Temporal CCS's semantics also enables the definition of operational equivalences between processes which we use as a basis for verification of processes. We return to this point later.

Both languages have had automated verification tools developed for them. The MT tool set [2] provides a graphical editor for manipulating Modechart designs, and it also includes capabilities for checking whether various timing constraints are obeyed by the system. Support for Temporal CCS is included in the Concurrency Workbench [1], a tool for simulating and verifying concurrent systems.

3.1 Modechart

Modechart [4, 3] supports the graphical representation of real-time systems as hierarchical state machines. The language 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 [3].

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. For our purposes a condition will either consist of the testing of a boolean state variable or the checking of the occurrence of an event elsewhere in the system. An effect will either be an assignment to a boolean variable or the emission of a (user-defined) event. 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 $[0, 0]$; that is, the transition is performed as soon as the condition becomes true. The formal semantics of Modechart is given via a translation into the logic RTL [3].

Figure 3 gives a high-level rendering of the intelligent structural control system. Conditions and timing constraints appear either above or to the right of transitions, while effects are given below or to the left. The timing values given in Figure 3 are based on an experimental setup described by Soong [8]; numerical values are in units of tenths of a millisecond (msec$*10^{-1}$). The system consists of three major components: a sensor, an actuator, and a control process. The sensor engages in two main activities; it collects data, and it sends it to the controller. In the case of this system, the former activity takes between 50 and 55 msec$*10^{-1}$, according to the experimental observations of [8], while the latter takes 15 msec$*10^{-1}$ on the part of the sensor. (There is no variation in the data transfer because the amount and format of the data to be sent are always the same.) Our model further divides this 15 msec$*10^{-1}$ into 10 for set-up (principally sending of a notification and adding of appropriate headers) and 5 for actual delivery.

Like the sensor, the actuator has two principal activities; it awaits the delivery of commands from the controller, and it then applies a pulse to the structure by, e.g., lengthening or contracting an expandable member.

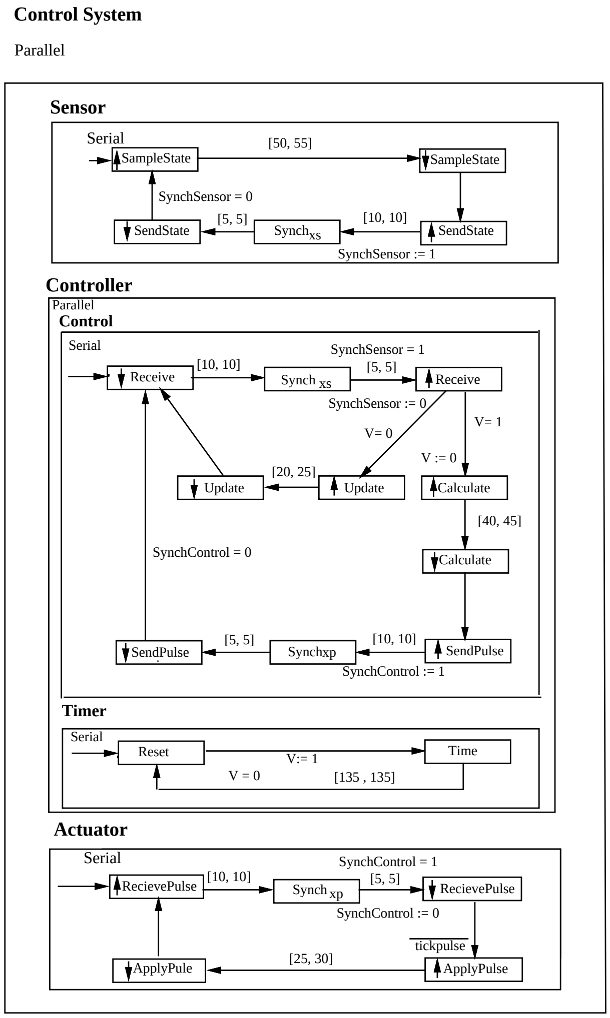

Figure 3: Modechart Model of a Control System

The communication between the controller and the actuator also takes 15 msec$*10^{-1}$, while the duration of a pulse application requires between 25 and 30 msec$*10^{-1}$. Note that the *tickpulse* event is introduced to allow observation of the beginning of pulse applications.

The activities of the controller consist first of collecting data from the sensors; if "not enough" time has elapsed since the previous pulse application it updates its internal values and resamples the sensors, while if "enough" time has passed it then calculates the appropriate pulse magnitude and sends it to the actuator. To model this behavior in Modechart we actually use two parallel subcomponents. One models the activities just described, while the second, a *timer*, ensures that appropriate intervals between successive pulse applications are maintained.

3.2 Temporal CCS

We now give an overview of the variant of Temporal CCS [6] used in this paper. The language extends Milner's Calculus of Communication Systems (CCS) [5] with timing constructs.

The definition of Temporal CCS is parameterized with respect to a time domain D and a set Λ of communication channels. For the purposes of this paper we take D to be the set of natural numbers. Letting $d \in D$ and $\alpha \in \Lambda$, the actions from which processes are built fall into one of four groups.

- d 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.
- $\tau \notin \Lambda$ represents a step of internal computation.

We use Act to denote the set of all actions and let $a, b, \ldots$ range over this set.

Now let $\mathcal{X}$ be a set of process names ranged over by $X, \ldots$, and $L \subseteq \Lambda$ be a set of channels. Temporal CCS processes may now be described using the following grammar.

$$\mathrm{P} ::= \mathit{nil} \mid X \mid a.P \mid P_1 + P_2 \mid P_1|P_2 \mid P\backslash L \mid P \text{ } \mathit{where}\ X_1 = P_1 \ldots X_n = P_n\ \mathit{end}$$

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 "delays". *nil* represents the stopped process; it is incapable of any action or idling. X may be thought of as an invocation of process X, where X is defined elsewhere. If $a \notin D$ then $a.P$ is a process that may engage in action a and then behave like P; it is incapable of idling. If $a \in D$ 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 P_i. On the other hand, if one of the P_i can idle and the other cannot, then $P_1 + P_2$ may idle, eliminating the "impatient" process in doing so. $P_1|P_2$ represents the parallel composition of P_1 and P_2. 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.

The variant of Temporal CCS considered here also enjoys the *maximal progress* property: any process capable of a τ is incapable of idling. It should be noted that Temporal CCS as described in [6] does not have this property; that language also includes other constructs that we do not describe here.

We close this section by reviewing the definition of an operational equivalence for Temporal CCS. *Observational equivalence* is designed to relate processes on the basis of their observable (i.e. non-τ) behavior. Define $P \stackrel{\epsilon}{\Rightarrow} Q$ to hold if P can perform some number (possibly 0) of τ-transitions and evolve to Q. If a is an action, then $P \stackrel{a}{\Rightarrow} Q$ holds 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. It is straightforward to establish that $\approx$ is an equivalence relation.

3.3 From Modechart to CCS

We now briefly indicate how (a subset of) of Modechart may be translated into Temporal CCS. Our methodology assumes the following restrictions on Modechart.

- Only primitive modes have outgoing transitions.
- An event can appear in the 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 as the action e.

In general, the translation converts parallel modes into CCS terms involving parallel composition and sequential modes into CCS terms that use the + operator to connect translations of individual outgoing transitions from the mode. The rendering 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. Effects are modeled using "output events".

Perhaps the most prominent feature of Modechart that is absent from CCS are boolean state variables. To model these, we adopt the classical approach of Milner [5]. Each shared variable is implemented as a CCS process that intuitively acts 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. Then the Temporal CCS model of V would be the following (assuming the variable is initially 0).

V = V0 where
 V0 = $\overline{rV0}$.V0 + $wV0$.V0 + $wV1$.V1 + 1.V0
 V1 = $\overline{rV1}$.V1 + $wV0$.V0 + $wV1$.V1 + 1.V0
 end

A process wishing to test the value of V would execute either $rV0$ or $rV1$ actions, while a process wishing to write to V would execute actions of the form $\overline{wV0}$ or $\overline{wV1}$.

As an example of the "output" of the translation procedure, the following is the Temporal CCS rendering of the sensor mode in the Modechart given in Figure 3.

Sensor = StartSampleState where
 StartSampleState = 50.Sample1
 Sample1 = τ.StopSampleState + τ.1.Sample2
 Sample2 = τ.StopSampleState + τ.1.Sample3
 Sample3 = τ.StopSampleState + τ.1.Sample4
 Sample4 = τ.StopSampleState + τ.1.Sample5
 Sample5 = τ.StopSampleState +
  τ.1.StopSampleState
 StopSampleState = StartSendState
 StartSendState = 10.$\overline{wSynchSensor1}$.Synch$_{XS}$
 Synch$_{XS}$ = 5.StopSendState
 StopSendState =
  $rSynchSensor0$.StartSampleState
  + 1.StopSendState
 end

The over-all structure of the entire Modechart would then be the following.

System = ((SynchSensor | SynchActuator) | Sensor | Actuator | (V | Control | Timer))\L

where *SynchSensor*, *SynchActuator* and *V* are the Temporal CCS processes for implementing the Modechart variables of the same names and L contains every channel mentioned in the design except *tickpulse*. The total number of states of *System* and its components as generated by the Concurrency Workbench are given below.

Agent	States
Sensor	78
Controller	115
Clock	137
Actuator	53
System	3174
State Variables (per variable)	2

4 Verification

This section reviews the approach we use in establishing that the system design presented in the previous sections is correct. It begins by stating what the constraints on the timing behavior are; it then discusses our methodology for determining whether the timing constraints are obeyed.

4.1 Timing Requirements

To formulate the timing requirements the system must satisfy we use the constraints given in Section 2. For the purposes of this paper, we concentrate on establishing that the period between successive pulse applications falls within the range indicated by [7], which established that if the natural period of a structure is T_0, then Δt_P must be constrained by: $T_0/8 \leq \Delta t_P \leq T_0/2$. In the experimental set-up of [8], the natural period of the structure under study was 290 msec$*10^{-1}$; thus, the time between successive pulse applications must fall within the range [37, 145].

4.2 Verification by Equivalence

Our approach to establishing that the pulse period is bounded by 37 and 145 msec$*10^{-1}$ involves the use of *observer processes* and equivalence checking. The basic idea is the following. We define processes that run in parallel with the system and observe its behavior by attempting to synchronize with the $\overline{tickpulse}$ events the system generates. If such an event is observed outside the allowed constraints, the observer generates a $\overline{pulseviolation}$ event. To establish that the system obeys the constraint, then, we establish that the observer and the system running in parallel are equivalent to a system in which no $\overline{pulseviolation}$ is possible.

In our particular case, we define two observers, one for the lower time bound and one for the upper. The lower bound observer should report a pulse violation if one $\overline{tickpulse}$ occurs within 36 msec$*10^{-1}$ after another. It is defined as follows.

$$
\begin{array}{ll}
\text{LowerObs} = \text{Start where} & \\
 & \text{Start} = \text{Response} + \text{CountUp} \\
 & \text{Response} = \text{tickpulse.Unsafe} + 1.\text{Response} \\
 & \text{CountUp} = 36.\tau.1.\text{Safe} \\
 & \text{Safe} = \overline{tickpulse}.\text{Start} + 1.\text{Safe} \\
 & \text{Unsafe} = \overline{pulseviolation}.\text{Unsafe} \\
 & \text{end}
\end{array}
$$

If the subsystem *Response* synchronizes with a $\overline{tickpulse}$ action then it enables $\overline{pulseviolation}$. The process *CountUp* "counts" 36 msec$*10^{-1}$; it then is capable of performing a τ (this "turns off" the idling capability of *Response*), after which it idles one more tick and evolves to *Safe*. *Safe* idles until it detects another $\overline{tickpulse}$, and the process is repeated. The net effect is that if less than 37 msec$*10^{-1}$ occur between any two $\overline{tickpulse}$ events, then a $\overline{pulseviolation}$ is possible.

The observer for the upper bound behaves analogously; in this case, a $\overline{tickpulse}$ should occur before 146 msec$*10^{-1}$ have elapsed in order for a $\overline{pulseviolation}$ to be avoided. The formal definition is omitted.

In order to establish that no timing violations occur, we can now show that *UpperConstraint* $\approx$ *Spec* and *LowerConstraint* $\approx$ *Spec*, where these processes are defined as follows.

$$
\begin{array}{l}
\text{UpperConstraint} = (\text{System} \mid \text{UpperObs}) \backslash \{tickpulse\} \\
\text{LowerConstraint} = (\text{System} \mid \text{LowerObs}) \backslash \{tickpulse\} \\
\text{Spec} = 1.\text{Spec}
\end{array}
$$

In each case, the Concurrency Workbench indicates that the equivalences are valid, meaning that our system will always verify the pulse period timing constraint.

5 Conclusions

Based on the experience gained during this exercise, we offer the following observations regarding the utility of Modechart, Temporal CCS and the Concurrency Workbench.

1. Modechart specifications have an immediate intuitive appeal, and the lower/upper bounds one can place on transitions can lead to simpler system descriptions.
2. The operational semantics of Temporal CCS eases the task of understanding what a system rendered in this language does.
3. Equivalence-based verification using "observers" can provide an intuitive and succinct approach to checking whether timing violations are present.
4. The tool would have been easier to use if it provided diagnostic information explaining why verifications failed.

As future work, we would like to investigate the development of a process algebra whose constructs more faithfully emulate those provided by Modechart. This would facilitate the direct translation of Modechart into the process algebra, which could then be given tool support by modifying the front end of the Concurrency Workbench.

References

[1] 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.

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

[3] 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.

[4] 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.

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

[6] 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.

[7] 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.

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