Comparing Architectural Design Styles

MARY SHAW, *Carnegie Mellon University*

◆*Different architectural styles lead not simply to different designs, but to designs with significantly different properties. This look at 11 designs of a cruise-control system shows that solutions varied, even within a design style, because of how the architectural choice leads the designer to view the system's environment.*

Good engineering solves problems not only by applying scientific techniques but also by making design choices that reconcile conflicting requirements. In the architectural design of software systems — which deals with system organization and system-level properties — early decisions about design strategies help define the problem and give insight into how to express the design.

One of the more difficult decisions in this area is the selection of an appropriate architectural style. As with most architectural design decisions, the choice of architectural style and its associated notations can have far-reaching consequences. The choice affects not only the system's description and its decomposition into components, but also its functionality and performance. Although strong advocates of some architectural styles tout each as the best choice for all problems, designers should select a style that matches the needs of each problem.

Unfortunately, although certain styles are commonly used, there is no formal documentation or "handbook" of architectural styles and their consequences. Designers often fail to explain their architectural decisions, nor can they record their architectural decisions permanently with the code. Thus, although many design idioms are available, they are not clearly described or distinguished, and the consequences of choosing a style are not well understood. This is a serious drawback because different architec-

0740-7459/94/$04.00 © 1994 IEEE

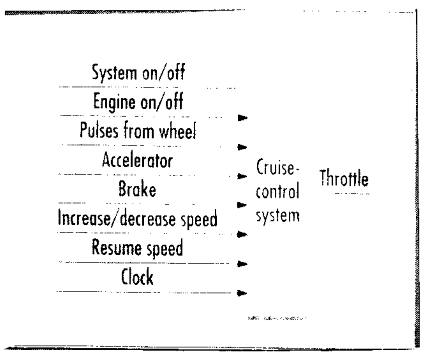


Figure 1. Block diagram for the Booch version of the cruise-control problem.

tural styles can lead not just to different designs, but to designs with significantly different properties, and designers need to understand the implications of choosing a particular style before committing to it.

In this article, I examine 11 designs for an automobile cruise-control system. Most of the designs appeal to multiple styles, but they generally fall into four main groups: object-oriented architectures, including information hiding; state-based architectures; feedback-control architectures; and architectures that emphasize the system's real-time properties.

It is my hope that this evaluation will not only make it easier to understand the relative merits of different architectural design idioms, but also serve as a springboard for analyzing the remaining obstacles to practical architectural design at the system level.

CRUISE-CONTROL PROBLEM

Researchers in many disciplines work out the details of their methods through *type problems*, commonly used examples for comparing models and methods.[1] The design of an automobile cruise-control system is a good type problem for software architecture and related issues because it accommodates different partitions of functionality and different relations among the resulting components.

Five of the designs presented here used a popular version of this problem formulated by Grady Booch.[2] The problem is adapted from a method developed by Paul Ward, which was used as a class exercise at the Rocky Mountain Institute for Software Engineering.

In the Booch version, the cruise-control system has the following properties, as stated by Booch:

It exists to maintain the speed of a car, even over varying terrain, when turned on by the driver. When the brake is applied, the system must relinquish speed control until told to resume. The system must increase and decrease speed as directed by the driver.

Figure 1 is the block diagram of the hardware for such a system. The inputs are defined as

♦ *System on/off.* If on, denotes that the cruise-control system should maintain the car speed.

♦ *Engine on/off.* If on, denotes that the car engine is turned on; the cruise-control system is only active if the engine is on.

♦ *Pulses from wheel.* A pulse is sent for every revolution of the wheel.

♦ *Accelerator.* Indication of how far the accelerator has been pressed.

♦ *Brake.* On when the brake is pressed; the cruise-control system temporarily reverts to manual control if the brake is pressed.

♦ *Increase/decrease speed.* Increase or decrease the maintained speed; only applicable if the cruise-control system is on.

♦ *Resume speed.* The last maintained speed is resumed; applicable only if the cruise-control system is on.

♦ *Clock.* Timing pulse every millisecond.

As the figure shows, the system has only one output:

♦ *Throttle.* Digital value for the engine-throttle setting.[2]

Three designs used less formal problem statements, and three designs used another, more complex, version, defined by James Kirby and John Brackett.[3,4] The Kirby version has minor differences from Booch's:

♦ It assigns functionality to the system inputs slightly differently.

♦ It requires a monitoring system to compute average speed, compute fuel consumption, and notify the driver of scheduled maintenance.

All these problem statements capture the same essential problem. The minor differences among them do not appear to have a significant impact on the styles of the solutions.

However, bear in mind that none of these problem statements quite captures the requirements of a real cruise-control system. For example, some do not fully specify how to determine the desired speed, only how to increase and decrease it. A real cruise-control system commands changes to current throttle settings, not absolute settings. It is hard to determine to what extent these discrepancies are accidents of specification and to what extent they crept in to make the problem more tractable, given the expressive power and limitations of the definition technique at hand.

OBJECT-ORIENTED ARCHITECTURES

Object-oriented architectures decompose systems into discrete objects that encapsulate state and definitions of operations. These objects interact by invoking each others' operations. Grady Booch defines an object as[2]

> *...an entity that has state; is characterized by the actions that it suffers and that it requires of other objects; is an instance of some class; is denoted by a name; has restricted visibility of and by other objects; may be viewed either by its specification or by its implementation.*

The first two terms of this definition are structural; the others affect the way objects are defined.

Methods for object-oriented design differ in how they define objects, timing, sequencing, and other dynamic properties and how they determine that the design satisfies the requirements. As the designs presented here show, other models often complement the object-oriented part of the design.

Booch: Object-oriented programming. Booch used the Booch version of the cruise-control problem to demonstrate object-oriented programming. He begins by modeling the problem space in a dataflow diagram. The diagram, which is shown in Figure 2, shows how information passes from its various sources through computations and internal states to the output value.

From this model of the requirements, Booch presents a functional decomposition of the design, in which modules correspond to major functions in the overall process. Figure 3 shows the functional decomposition.

From the dataflow model in Figure 1, Booch structures an object-oriented design around objects in the task description. This design, shown in Figure 4, yields an architecture whose elements correspond to important quantities and physical entities in the system. The blobs in the figure correspond approximately to the inputs and internal states in Figure 2. The arrows are derived from the data paths in Figure 2. Other object-oriented designs show slightly different dependencies, as I describe later.

The object-oriented design emphasizes the objects in the problem domain and the dependencies they have with major elements of internal state. It makes no distinction between objects of the problem domain and internal objects, nor does it show the nature of the dependencies.

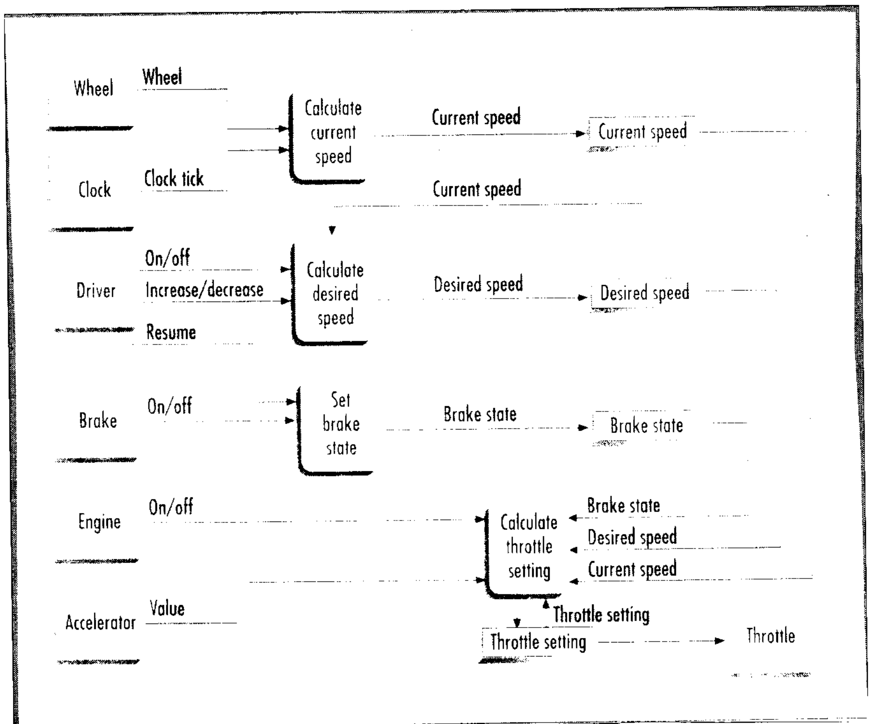


Figure 2. Booch's dataflow diagram for the Booch version of the cruise-control problem.

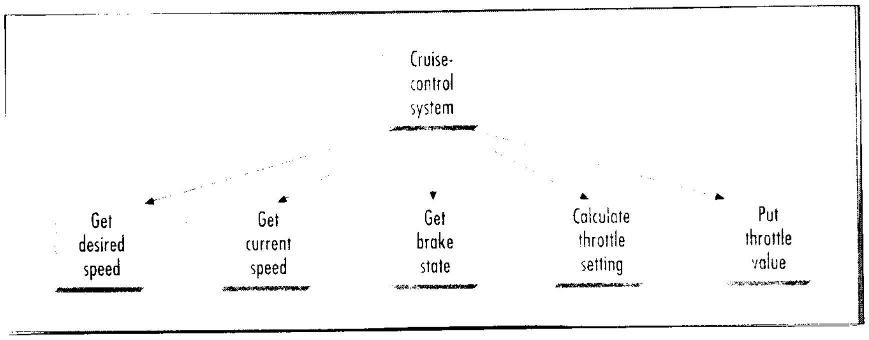


Figure 3. Booch's functional design.

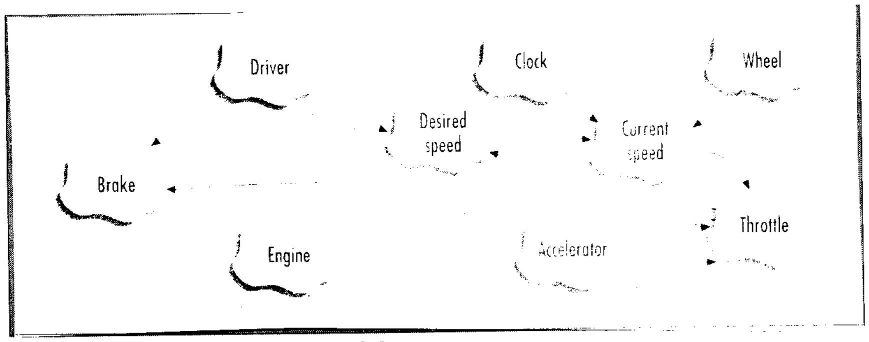


Figure 4. Booch's object-oriented diagram.

Yin and Tanik: Reusability. W.P. Yin and Murat Tanik used Booch's version of the cruise-control problem to demonstrate software reusability in Ada.[5] They chose the problem because it requires parallel processing, real-time control, exception handling, and unique I/O control. They use essentially the same dataflow diagram as Booch's to model the requirements. Following Booch's development technique, they derive an object-oriented design, as shown in Figure 5.

Their design differs from Booch's (Figure 4), however, in that they create objects for all the external elements and one single object for the entire cruise-control system. Thus, Figure 5 shows just the dependencies of the core solution on the external elements and any

Best Copy Available

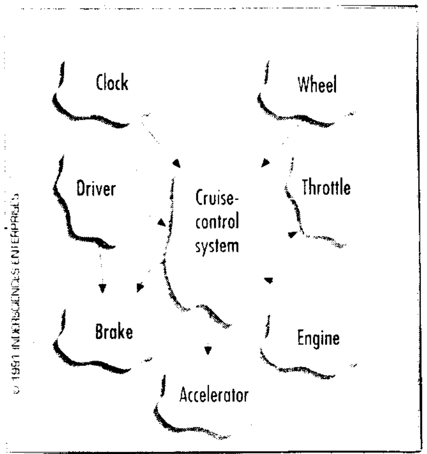


Figure 5. Yin and Tanik's object-oriented design.

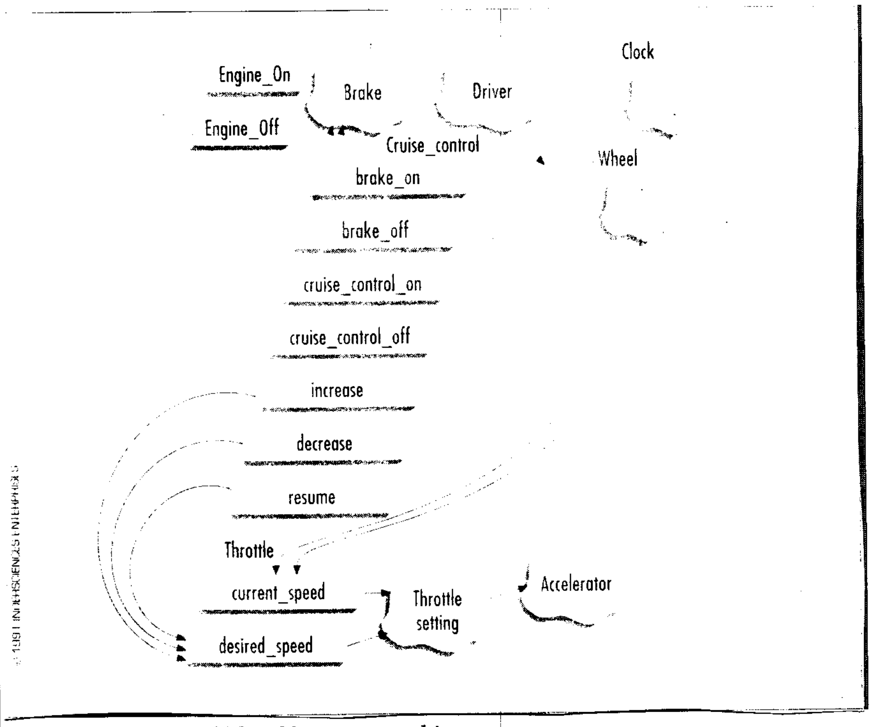


Figure 6. Yin and Tanik's system architecture.

interactions among external elements. The system architecture of Figure 6 elaborates Figure 5 by showing operations for the major objects. It also rearranges the design substantially: all other objects are (evidently) internal to the engine, and the relation of the throttle to everything else is much different.

Although they chose the problem because of its real-time characteristics, Yin and Tanik's design does not address timing questions. In fact, they cite the need for additional facilities to deal with timing constraints.[5]

Birchenough and Cameron: JSD and OOD. In this design, Alan Birchenough and John Cameron used the Booch version

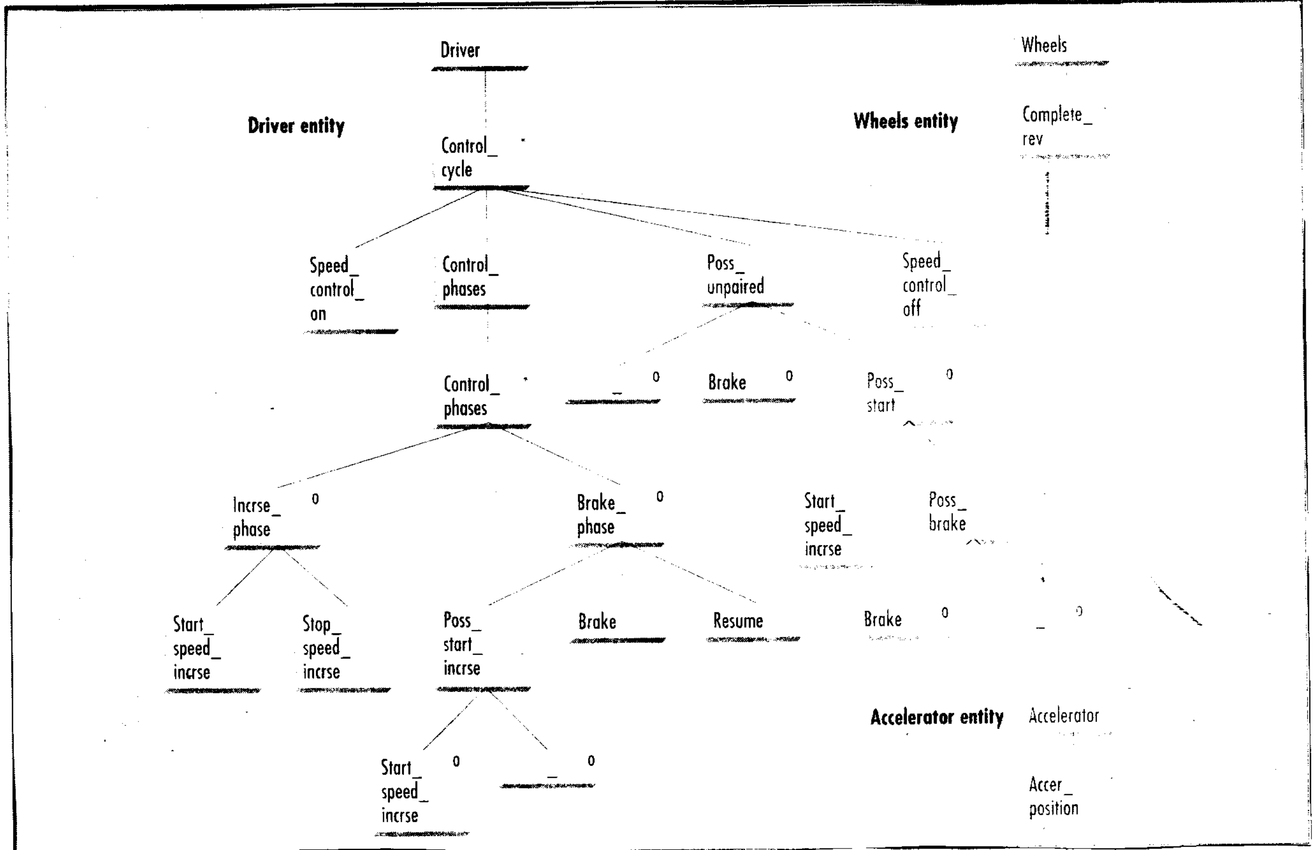


Figure 7. Cruise-control-system entity structures in the Jackson System Development method.

Best Copy Available

Cruise-Control Request

Input Data Item:	Cruise-control request
Acronym:	/Lever/
Hardware:	Cruise-control lever
Description:	The value of /lever/ is determined by the cruise-control lever. The lever has five positions. Four of the five positions are labeled CONST, OFF (two instances), and RESUME. The lever may be pushed and held in one of these positions at a time. When released, the lever will always return to the unlabeled RELEASE position. The value of /lever/ is described by the table below.

Characteristics of Values:
Values: $inactivate$, $activate$, $start incr$, $stop incr$, $resume$

Inputs from Cruise-Control Lever

Event	**/Lever/**
@T(enter *on*)	$inactivate$
@T(!const pos!)	$activate$
@T(!const pos! for > 500 msec)	$start incr$
@T(!release pos!) when !const pos! for > 500 msec	$stop incr$
@T(!off pos!)	$inactivate$
@T(!resume pos!)	$resume$

***Figure 8.** Kirby's input data item (cruise-control request).*

of the cruise-control problem (with some very minor differences) to show how designers can use Jackson System Development and object-oriented design in a complementary fashion.[6] Both approaches support the principle that the software-system structure should match the structure of the problem it solves. Both rely on identifying discrete entities (which correspond to objects) and the operations they commit on each other.

However, JSD has three stages: modeling (separate from function), network, and implementation. It encourages the analysis of actions, or events, before identifying entities. OOD, on the other hand, proceeds in the opposite order. In JSD, the modeling stage (the only stage of concern here) considers time-ordered or state-changing operations when identifying objects. As a result it is more conservative in object identification than OOD.

In contrast to Booch's application, which found nine objects (Figure 4), JSD found three: `wheels`, `accelerator`, and `driver`. Figure 7 shows these three entities along with their substructure. The substructure shows time-ordering among the operations through notations in the upper right corners of the boxes.

In the JSD methodology, functional components are chosen for problem-specific reasons, not to correspond with real-world entities (as in object-oriented methods). The authors conclude that a more object-oriented approach would make the design easier to understand in later stages.

Kirby: NRL/SCR approach. Kirby used the Kirby version of the cruise-control problem and the Naval Research Laboratory/Software Cost Reduction approach to design the system.[3] The NRL/SCR approach is based on information hiding, precision, and completeness. It provides separate definitions of inputs and outputs, the modes of operation, functionality, timing, accuracy, and undesired events. Each of these comprise a set of discrete definitions in uniform format. The approach uses tables heavily but no architectural diagrams.

Kirby's solution isolates 19 distinct I/O items, one of which is the cruise-control request, which Figure 8 shows.

The modes of operation and the state-transition table are essentially the same as those provided by Atlee and Gannon's state-based translation (described later).

Figure 9 shows the definition of the throttle-setting function and the output description, given as a decision table. The other functions — timing, accuracy, and undesired events —are enumerated and tabulated in a similar manner.

Determine Throttle-Setting Function

Modes in which function is required:	*Cruise*
Output data item:	//Throttle//
Initiation Event:	@T(enter mode)
Termination Event:	@T(exit mode)
Output description	Table shows required value of //throttle//

Throttle Setting				
Mode	**Condition**			
cruise	!too slow! OR !start incr!	!speed ok! AND NOT !start incr! AND NOT !driver accel!	!too fast! AND NOT !start incr! AND NOT !driver accel!	!driver accel! AND NOT !too slow! AND NOT !start incr!
//Throttle//	!throttle accel!	!throttle maintain!	!throttle decel!	!throttle off!

***Figure 9.** Kirby's definition and output description of the throttle-setting function.*

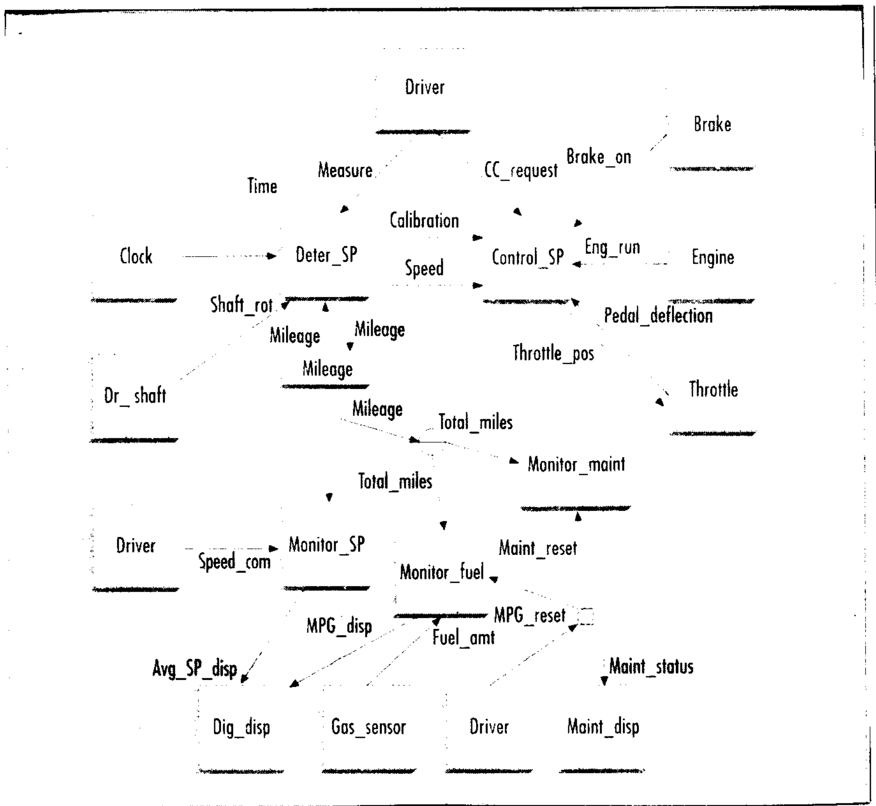


Figure 10. *Smith and Gerhart's top-level activity chart.* `Mileage` *is a data store consisting of* `Total_Miles` *and* `Time_Set`.

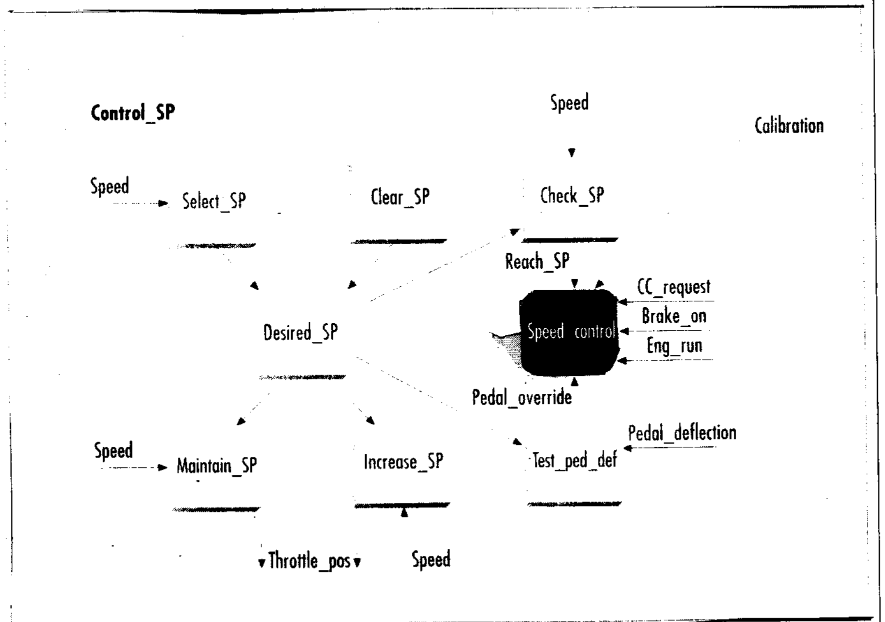


Figure 11. *Smith and Gerhart's speed-control activity chart.* `Desired_SP` *is a data store.*

STATE-BASED ARCHITECTURES

State-based architectures focus on the major modes, or *states*, of the system and on the events that cause transitions between states. As with object-oriented designs, other models may complement the state analysis, for example to identify the major modes. State machines also appear as secondary views in the Kirby design, as well as in my feedback-control design and Paul Ward and Dinesh Keskar's real-time design (described later).

Smith and Gerhart: Statemate. Statemate uses a state-transition formalism supported by two graphical notations: activity charts (a form of functional decomposition) and statecharts (a representation of finite-state machines). It is particularly well-suited to reactive systems. It can be used with many methodologies: functional methods focus on activity charts; behavioral methods focus on statecharts.

Sharon Smith and Susan Gerhart[7] used the Kirby version of the cruise-control problem to compare Statemate to Brackett's use of the Hatley method of the same problem.[4] The Hatley method uses a graphical notation for a functional view. Smith and Gerhart abstract from physical devices such as "brake pedal" to obtain the actions selected by the driver such as "inactivate system."

To compare their result to Brackett's, Smith and Gerhart chose the functional-decomposition design method, in which top-level functions emerge directly from the system requirements. In the activity chart in Figure 10, the top-level functions are the boxes at the boundary of the system. Because all the functions at this level operate concurrently, no statechart is required. The granularity of the activities is coarser than Booch's functional elements, which are simple constructors and selectors.

The second level decomposes the internal activities of the first level. Smith and Gerhart illustrate this step with the function for speed control. The requirements lead to the functions shown in Figure 11. Now, however, ordering dependencies affect execution, so a controlling statechart is required. This appears as a shaded box in Figure 11 and is elaborated in Figure 12.

As Smith and Gerhart observe, the advantages of statecharts include good notations for showing concurrency and restrictions on concurrency, mechanisms for showing state changes, notations for certain timing constraints, and Statemate's simulation and analysis capabilities. However, this formulation requires more notation than others for the same level of detail, and large (or variable) numbers of similar activities are hard to represent. The authors also note that behavioral and functional approaches will lead to different designs.

Atlee and Gannon: State-based model checking. Joanne Atlee and John Gannon were interested in using a model checker on requirements for large systems.[8] Their state-based formulation of a cruise-control system was therefore presented as a requirement, but it is close enough to a design for consideration here. They partition the possible states of the system into four major modes: off, inactive, cruise, and override (on but not in control).

As Figure 13 shows, the requirements are given as a table that shows how events (flagged with @) and conditions cause mode transitions. Their analysis considers interactions among conditions ("what if the brake is on when the system is activated?") and identifies important invariants among conditions and events. Their definition is essentially the same as Kirby's for this aspect of the problem.

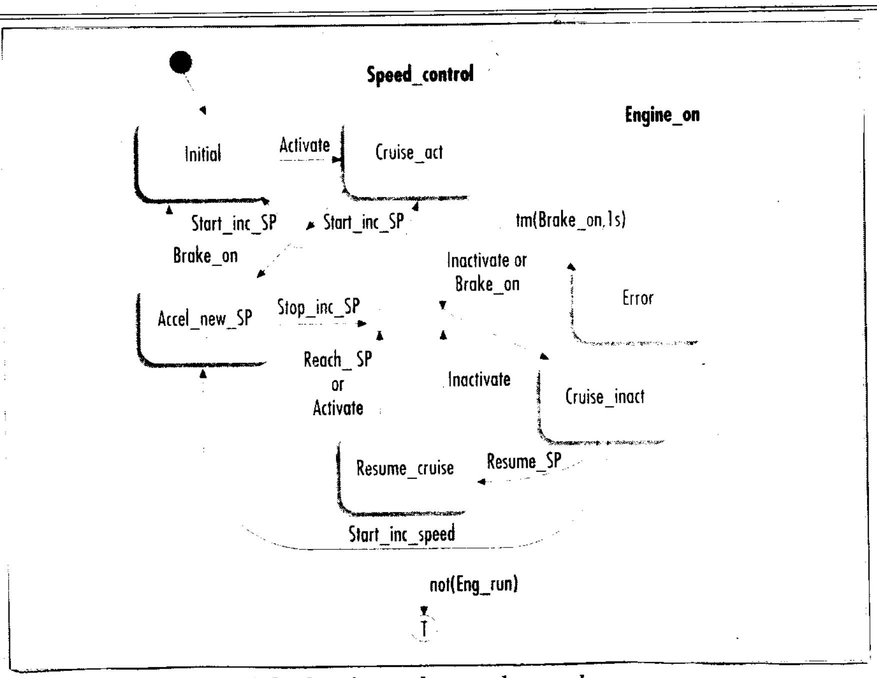


Figure 12. Smith and Gerhart's speed-control state chart.

FEEDBACK-CONTROL ARCHITECTURES

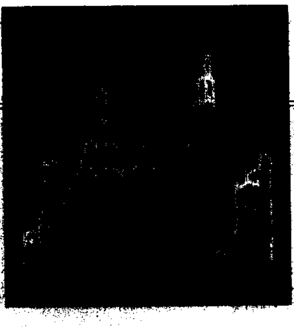
Feedback-control architectures are a special form of dataflow architecture adapted for those embedded systems in which the process must regularly adapt to compensate for external perturbations. These architectures are modeled on analog process-control systems, and they model the current values of the outputs and the value that controls the process as continuously available signals.

Higgins: Extension of DSSD for real-time systems. David Higgins extended the Data Structured System Development

Mode transitions for automobile cruise control

Current mode	Ignited	Running	Too fast	Brake	Activate	Deactivate	Resume	New mode
Off	@T	—	—	—	—	—	—	Inactive
Inactive	@F	—	—	—	—	—	—	Off
	t	t	—	f	@T	—	—	Cruise
Cruise	@F	—	—	—	—	—	—	Off
	—	@F	—	—	—	—	—	Inactive
	—	—	@T	—	—	—	—	
	—	—	—	@T	—	—	—	Override
	—	—	—	—	—	@T	—	
Override	@F	—	—	—	—	—	—	Off
	—	@F	—	—	—	—	—	Inactive
	t	t	—	f	@T	—	—	Cruise
	t	t	—	f	—	—	@T	

Figure 13. Atlee and Gannon's mode-transition table. Initial mode is `Off`.



method for real-time embedded systems by adding a standard template that lets designers analyze feedback-control models.[9] The conversion from a standard feedback-control block diagram to a data-structure entity diagram is straightforward; it includes a "summing point" for feedback to compare the feedback signal with the reference input. It also explicitly recognizes the external disturbance and the source of the target value of the process variable as well as the more obvious processing elements. The entity diagram in Figure 14 is static, so it is complemented by a functional flow diagram, in Figure 15, to establish the communication dynamics.

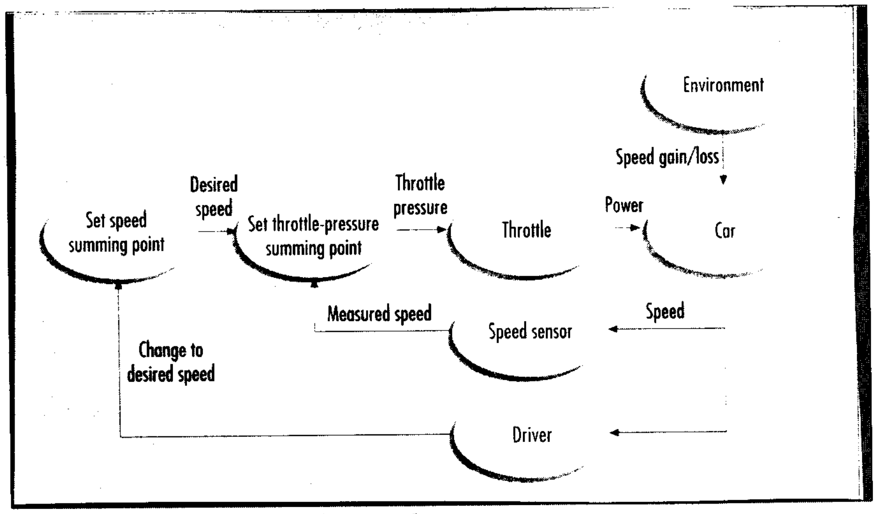


Figure 14. Higgins' feedback entity diagram.

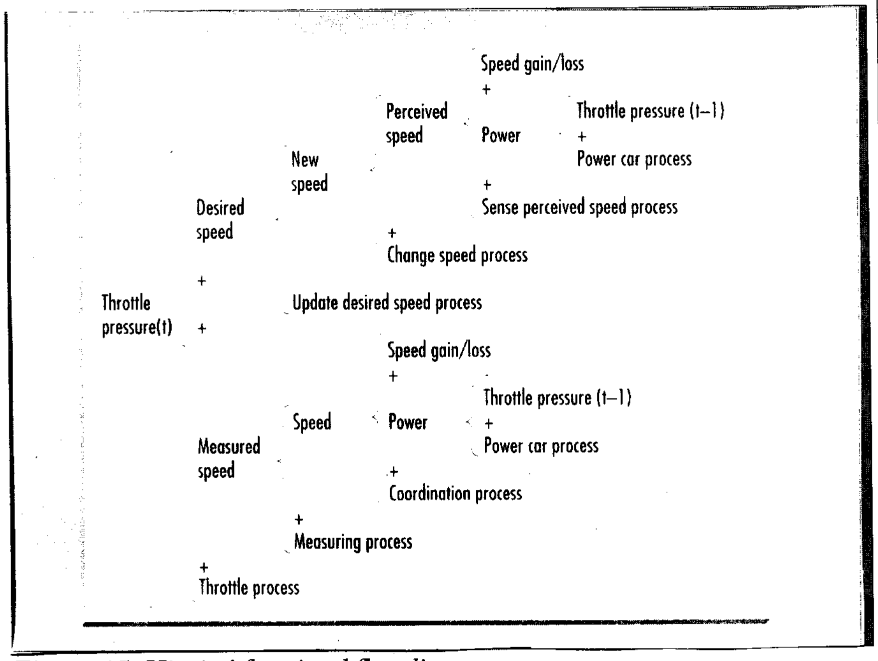


Figure 15. Higgins' functional flow diagram.

The functional flow diagram shows how the input signals produce the output signals. The inputs to each level of flow (vertical lines in a right-to-left progression) are on the right and the output is on the left; by convention, the lowest element on the input list (the element at the bottom right of the line) is the name of the transformation. When the system flow is defined, the final step is to convert the functional flow diagram to a data-structure diagram that shows the hierarchy of the system data.

Higgins used an informal statement of cruise-control system requirements to illustrate the method. Figure 15 is a simple form of the example; additional levels of control, such as if the engine is on or the system is activated, are handled as secondary control loops that embed this example as the controlled system in a surrounding control loop.

Show: Process-control paradigm. I explored a software idiom based on process-control loops[10] using the Booch version of the cruise-control problem. Unlike object-oriented or functional designs, which are characterized by the kinds of components that appear, control-loop designs are characterized by both the kinds of components and the special relations that must hold among the components.

The elements of this pattern incorporate the essential parts of a process-control loop, and the methodology requires designers to explicitly identify these parts. The parts include two computational elements (the process definition and the control algorithm), three data elements (the process variables, the set point or reference value, and the sensors), and the control-loop paradigm, which establishes how the control algorithm drives the process. I characterized the result as a specialized form of dataflow architecture.

The essential control problem establishes a control relation over the engine, as Figure 16 shows. This does

Best Copy Available

not, however, solve the entire problem because most of Booch's inputs are actually used to determine if the system is active or inactive and to set the desired speed. The problem of determining if the system is active or inactive is a state-transition problem, and the solution, shown in Figure 17, is much like that of the state-based designs. The problem of setting the desired speed lends itself to a decision table, such as that in Figure 18.

Figure 19 shows how the control architecture, the state machine for activation, and the event table for determining the set point form a complete cruise-control system.

As I concluded from this design, it is appropriate to consider a control-loop design when the task involves continuing action, behavior, or state; when the software is embedded (controls a physical system); and when uncontrolled, or open loop, computation does not suffice (usually because of external perturbations or imprecise knowledge of the external state).

REAL-TIME ARCHITECTURES

Real-time systems must meet stringent response-time requirements. Extensions to several methods add various features to satisfy the special demands of real-time processing. Interestingly, the designs given here deal with event ordering but not with absolute time. Higgins' process-control model (described earlier), for example, was motivated by real-time problems but is primarily a feedback-control architecture.

Ward and Keskar: Extensions of DMSA. Ward and Keskar compared two extensions of DeMarco Structured Analysis that support real-time system models: Ward/Mellor and Boeing/Hatley.[11] DMSA was developed for commercial business applications, and these extensions were developed because pure structured analysis could not effectively capture the kinds of

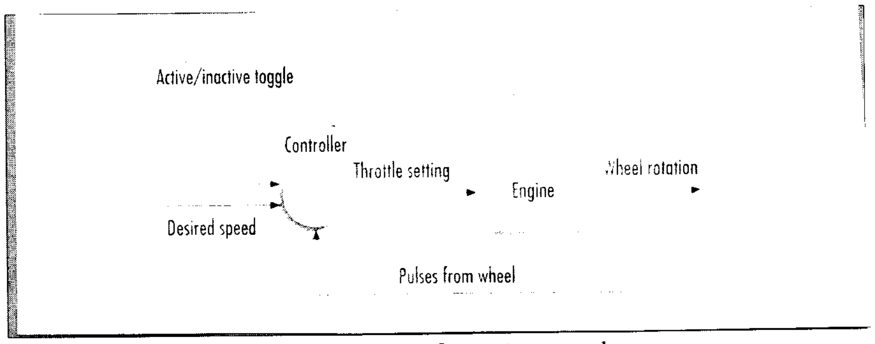


Figure 16. Shaw's control architecture for cruise control

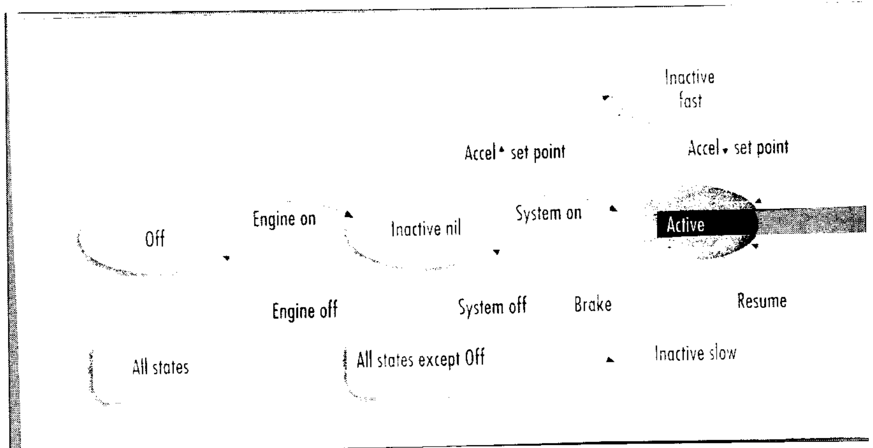


Figure 17. Shaw's state machine for activation.

Event	Effect on desired speed
Engine off, system off	Set to "undefined"
System on	Set to current speed as estimated from wheel pulses
Increase speed	Increment desired speed by constant
Decrease speed	Decrement desired speed by constant

Figure 18. Shaw's event table for determining set point.

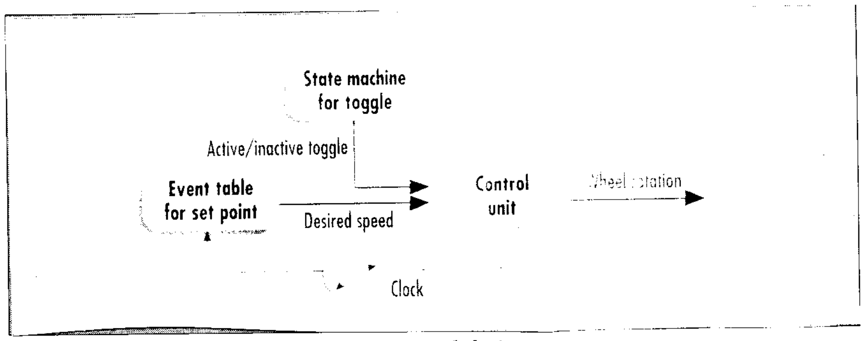
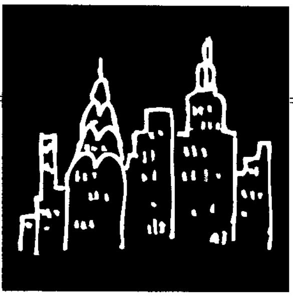


Figure 19. Shaw's complete cruise-control design.

Best Copy Available



time-dependent actions that appear in process control, avionics, medical instrumentation, communications, and similar domains. Ward and Keskar used an informal statement of cruise-control requirements in both designs.

Ward/Mellor approach. The WM approach extends the basic structured-analysis dataflow diagram, which is depicted by the solid lines in Figure 20, by adding event flows, the dotted lines in the figure, to show time-dependent behavior. The control transformation Control Speed receives events from the external interface and enables, disables, or triggers the basic functions. The logic of the control transformations is described by the state-transition diagram in Figure 21. In a given state, only certain events are recognized; when they occur, they change state (>) and enable (>>), disable (<<), or trigger ([) the indicated transformations.

This design is meaningful only when both the engine and cruise-control system are on. Ward and Keskar show a hierarchical extension with an additional control state to handle this. Some of the interactions examined by Atlee and Gannon can arise here, but designers can prevent the attendant problems by adding explicit detail to handle the interactions.

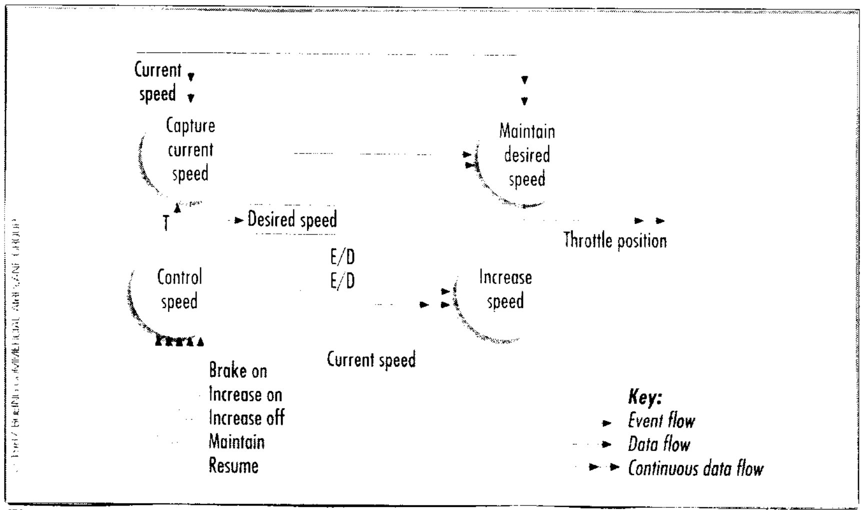


Figure 20. *Ward and Keskar's control transformation diagram for the Ward/Mellor design.*

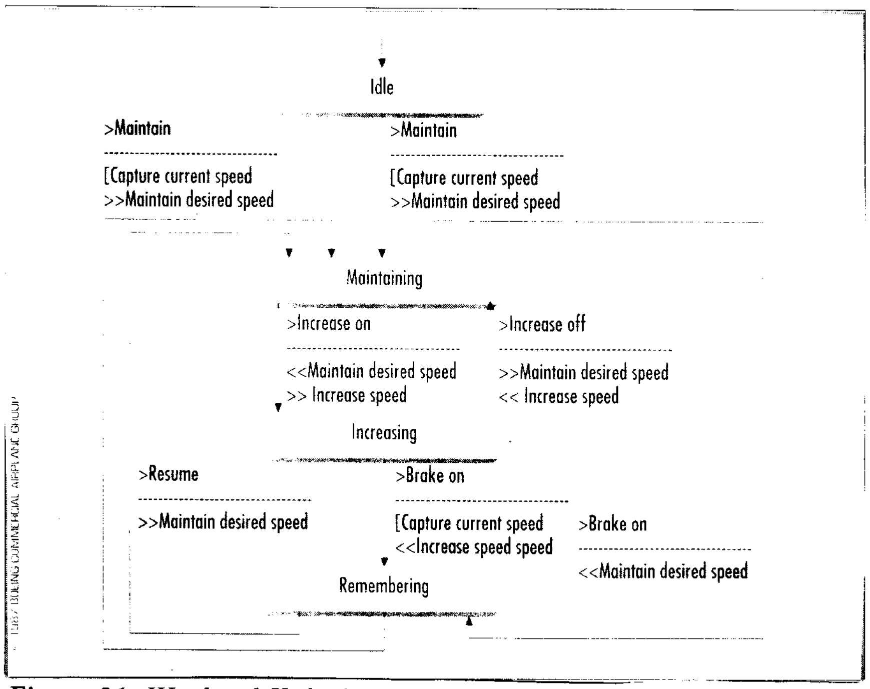


Figure 21. *Ward and Keskar's state-transition diagram for the Ward/Mellor design.*

Boeing/Hatley approach. The BH extension begins with two context diagrams that show dataflow and control flow among the physical components. The highest level of the design consists of a dataflow diagram, in Figure 22, and a variant on the dataflow diagram to show control flow, in Figure 23. These two diagrams are based on the same entities.

Ward and Keskar then add control specifications to show how to activate or deactivate processes on the dataflow diagram. The control specifications may be combinatorial (no state) or sequential (internal state). Cruise control is sequential, so its control is described in three parts: a decision table, in Figure 24, that converts combinations of input signals to output signals; a state-transition diagram (essentially similar to Figure 20); and an activation table, in Figure 25, that relates the transition actions of the state machine to the processes of the dataflow diagram.

COMPARISON

All these designs address a single

Best Copy Available

task. Indeed, most started from one of two problem statements. Nevertheless, the solutions differ substantially, even within a single architectural style. Some of the differences can be attributed to variations among individual designers, but others follow from the way each architecture leads the designer to view the world.

Designs based on different architectures emphasize different aspects of the problem. Some focus on interpreting driver controls, some focus on internal state, and still others focus on the actual control of automobile speed. The ability to zero in on specific critical issues is important, but designers must consciously match the technology to the needs of the client so as to emphasize the right issues.

Design models. Even when a design is declared to be in a particular style, the designer usually appeals to two or three design representations or models. These models are used in many combinations and provide different views of the design. The only real point of consistency is that one of the models matches the declared style. Table 1 identifies the models used in the examples presented here.

Within a design, each view uses a model that expresses certain relations and may be silent on others. As a result, the designer must take special care to determine that the views are mutually consistent. Within a single

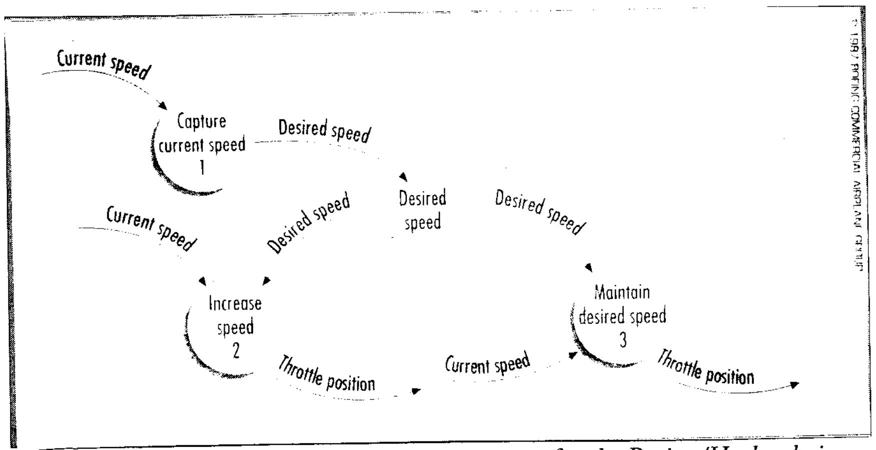


Figure 22. Ward and Keskar's dataflow diagram for the Boeing/Hatley design.

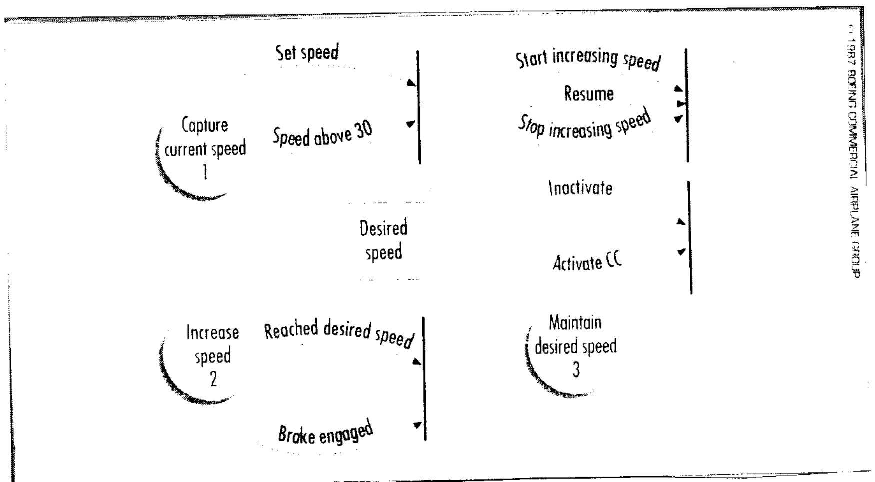


Figure 23. Ward and Keskar's control-flow diagram for the Boeing/Hatley design.

Speed_above_30	Set_speed	Hold _speed
True	True	True
True	False	False
False	True	False
False	False	False

Figure 24. Ward and Keskar's decision table for the Boeing/Hatley design.

Process / Action	Capture current speed 1	Increase speed 2	Maintain speed 3
Stop_accelerating	0	0	0
Accelerate	0	1	0
Maintain_speed	0	0	1
Accelerate_to_desired_speed	0	1	0
Get_ and _maintain_speed	1	0	1
Stop_maintaining_speed	0	0	0
Turn_CC_off	0	0	0
Cruise_control_ready	0	0	0

Figure 25. Ward and Keskar's activation table for the Boeing/Hatley design.

TABLE 1
DESIGN MODELS USED IN 11 DESIGNS OF A CRUISE-CONTROL SYSTEM

Design Strategy	Functional Decomposition	Dataflow	Object-Oriented	State Machine	Event-Oriented	Process Control	Decision Table	Data Structure
Functional	√	For rqts.						
Object-oriented (Booch)		For rqts.	√					
Object-oriented (Yin and Tanik)		For rqts.	√					
JSD and OO	JSD activity chart		√					
NRL/SCR			Info. hiding	√			√	
Statemate	Activity chart			State chart				
State machines				√				
Process control (Higgins)	√					√		√
Process control (Shaw)				√		√	√	
Ward/Mellor		Mixed with event flow		√	Mixed with dataflow			
Boeing/Hatley		√		√	Control flows		√	

design, each view may decompose the task into different entities, but not all these entities will persist until runtime.

Evaluation criteria. Designers can evaluate designs in several ways. The precise selection of criteria should depend on the requirements of each application. Certain considerations generally apply, such as locality and separation of concerns, perpescuity of design, the ability to analyze and check the design, and abstraction power. There will also be criteria that apply specifically to each problem. For cruise control, we consider safety and integration with the vehicle.

These criteria provide a basis for comparing the design approaches.

Separation of concerns and locality. Does the design separate independent parts of the system, group closely related parts, and avoid redundant representation of information? How easy will it be to make modifications?

All the models provide a way to decompose the system into separate parts that localize decisions that are significant *with respect to that model.* Generally speaking,

♦ Object-oriented designs focus on real-world entities and data/computational dependencies from input entities to output. They are little concerned with internal relations or operation sequencing.

♦ State-oriented designs focus on the system's operational modes and the conditions that cause transitions from one state to another.

♦ Process-control designs focus on the feedback relation between actual and desired speed.

♦ Real-time designs focus on events and the order in which they occur. The real-time designs presented here don't say much about actual timing.

Each design involves choosing what information to localize and which concerns to separate. These choices can cause large differences in which aspects of the problem each design addresses.

Locality is motivated not only by design simplification but also by the ability to interchange parts among designs. When a designer combines architectural styles, there is a tendency to select parts with different packaging — parts that have different interaction methods. This can interfere substantially with exchange or interoperability. In fact, failure to recognize these discrepancies may be one of the bigger problems in software reuse.

Locality is also motivated by the prospect of future modifications. Booch, for example, argues that object decompositions are superior to functional decompositions because functional decompositions have global data, and future changes may require representation changes. This is sometimes true, but not always. As David Parnas argued two decades ago, locality should hide the decisions *most likely* to be changed. Which decisions these are depends on the application.

Separation of concerns becomes more complex when multiple models are used, which happened in most of these designs. Whenever multiple views are defined, they constrain the design in different ways. Designers must show how those views are related (as I describe later under the ability to analyze criterion).

Perspicuity of design. Does the expression of the design correspond clearly to the problem being solved? Is the design's response to the most significant requirements easy to identify and check?

Perspicuity is in the mind of the beholder. Most methodologies exhort designers to make the design match the real world. But as these examples show, the real world has many faces. Each design presented here has a claim to some view of reality, though some (objects, process control) do so more consciously than others (functional). As I describe later under the safety evaluation criterion, the designer must understand which aspects of the real world are most important to the client.

If the client is most concerned with the devices the driver manipulates to indicate desired speed and changes of speed, object-oriented designs are a good match. If the client is most concerned with the possible modes of the system and with the assurance that obscure interactions will not make the system unsafe, a state-based design will more likely bring out the information of interest. Finally, if the client is most concerned with the embedded-feedback problem of actually controlling the speed, a process-control design will show the necessary relations without extraneous detail.

Many software-development methodologies begin with a domain analysis. This provides a good opportunity for an initial choice of architecture.

MOST METHODS EXHORT DESIGNERS TO MAKE THE DESIGN MATCH THE REAL WORLD. BUT THE REAL WORLD HAS MANY FACES.

Ability to analyze and check the design. Is the design easy to analyze or check for correctness, performance, and other properties of interest? If more than one model or notation is used, how easy is it to understand the interaction?

Most of the models have associated analysis techniques for the aspects of the design they are intended to bring out. I just described how the important aspects of the design depend on the client and the problem. Designers must also consider the problem of analyzing and checking the design when multiple models are used.

Multiple models in many combinations exacerbate consistency problems, especially when they use different decompositions at the same level. Initial consistency is important, of course, but the ability to make changes later is also at stake. This is a challenge when different styles decompose the problem into different elements.

Some uses of different models are easy to handle. For example, if designers use one model to refine a component that appears in another view, the interaction between models lies at the interface of the component being expanded. This is the case in my design, in which Figure 19 shows how the models in Figures 17 and 18 provide inputs needed in the feedback process in Figure 16. Another tractable interaction is when one model is derived from another, as Higgins derives first functional flow (Figure 15) and then the data structure from the feedback entity diagram (Figure 14).

The situation is more difficult when multiple kinds of structure are imposed on a single set of base entities, such as in Figure 20, where Ward and Keskar add event flows to a dataflow diagram.

Still more difficult is the situation in which a designer decomposes the problem in essentially different ways, and the semantics of the models interact. This arises, for example, when state models are added to other designs in Statemate (Figure 12) and in the Ward/Mellor analysis (Figure 21). Each of these combinations can, of course, be handled as a special case of the methodology, but as Table 1 shows, the models are used in many combinations, which makes the task of devising special analyses for each combination somewhat daunting.

Abstraction power. Does the design highlight and hide the appropriate details? Does it help the designer avoid premature implementation decisions?

The essence of abstraction is identifying, organizing, and presenting appropriate details while suppressing details that are not currently relevant. Abstraction is often supported by design discipline, notation, or analysis tools, which guide the designer in selecting details to emphasize and suppress. Disci-plines do this explicitly; notations and tools do it implicitly, by providing the ability to express some things, by requiring certain details, or by having no means of expressing other things.

The need to decide what's currently

relevant is illustrated by the differences between the object-oriented and process-oriented designs given here. These object-oriented designs are concerned with the external devices the driver will manipulate. They begin by enumerating objects of the real world and relations among these objects. They also relegate control of the system to internal entities.

SYSTEMATIC DIFFERENCES IN ARCHITECTURES LEAD DESIGNERS TO ASK DIFFERENT KINDS OF QUESTIONS.

The feedback designs, on the other hand, are concerned with the speed of the vehicle and how to control it. They begin by setting up the feedback loop and establishing the relation among reference speed, the model of the current speed, and the engine (as controlled by the throttle). These designs deal lightly with the conversion of driver actions to the signals of interest.

Abstraction should also suppress implementation decisions. Several designs unnecessarily reveal implementation decisions: Booch and Yin and Tanik included a global system clock and the fact that it has a millisecond resolution. Kirby included extensive information about range, resolution, and accuracy of values. Ward and Keskar included details of the definition of events at the beginning of the design. On the other hand, Ward and Keskar did not include timing information, even though they claimed to be producing a real-time design.

Safety. Can the system fully control the vehicle, and can it ensure that the vehicle will not enter an unsafe state as a consequence of the control?

Cruise-control systems exercise largely auto-nomous control over moving machines. De-signers of such systems should consider explicitly if the machines will be safe in operation. Two particular questions arise here: Does the system have full control over the speed of the vehicle? Can the system's model of the world (the current speed) be sufficiently wrong to be dangerous?

The requirement says that the cruise-control system should "maintain the speed of the car." However, most automobile cruise controls (and all the designs here) can control only the throttle, not the brake. As a result, if the car picks up excess speed (coasting down a hill, for example), the system cannot slow it down. The only designs that recognize this problem are Kirby's NRL/SCR design, Atlee and Gannon's state-based formulation, and my feedback-loop model.

The design process should also include some provision that leads the designer to consider an even more serious problem: the possibility that the cruise control's model of current speed is radically different from the actual speed. This can happen, for example, if the sensor is on a nondrive wheel and the drive wheels start spinning. It can also happen if the sensor fails. The Kirby problem formulation calls for a calibration capability, but this addresses only the problem of gradual drift, not that of sudden inaccuracy. The only designs that explicitly direct the designer's attention to the accuracy of the current speed model are Higgins' and my feedback-loop models. This is not surprising, since these are the only two designs that explicitly treat the problem as one of feedback control.

Integration with the vehicle. How well do inputs and outputs match the actual controls? How do the manual and automatic modes interact? What are the characteristics of real-time response? How rapidly and smoothly does the vehicle respond to control inputs? How accurately does the system maintain speed?

Ultimately, the designed system must not only satisfy the problem statement but also integrate with the whole automobile. The Kirby problem statement recognizes this by requiring calibration capability to deal with different tire sizes. However, this problem would not arise in a system that uses relative rather than absolute speeds; such a system deals with "faster" and "slower" rather than calculated speeds. In such a system, cruise control can be independent of speedometer display.

The solutions differ in the extent to which they address the relation between the user-manipulated controls and the system's internal state data. The object-oriented architectures focus on this aspect. The state-based architectures define this as outside the scope of their problem. The feedback-control architectures focus on the control question first but include the interpretation of user inputs as a separate design stage. The real-time architectures show the relation as being in a system context.

Most of these designs simply compute a throttle value. Only my feedback-loop model and Ward & Keskar's two designs consider the rate at which speed should be increased or the system's response characteristics. Higgins' design has an explicit entity (Set Throttle Pressure Summing Point) whose elaboration might reasonably address this question.

This comparison shows that, although architectural style strongly influences the resulting design, it does not fully determine the result. It is not surprising that different designers use the same approach and get different results, since all design methods allow considerable room for individual judgment. There are, however, systematic differences in the kinds of questions designers are led to ask by different architectures.

Notwithstanding the label associated with an architectural style, the designer usually develops several models. Sometimes, but not always, the relation among these system views is well-defined. The examples in this article show that designers need a systematic means of establishing the relations among multiple views.

This comparison also highlights two open research problems: How can a designer select the architectural style most appropriate to the problem at hand, and how can a designer maintain the consistency of different views of a design? I am currently working on issues in both these areas.[12-13] ◆

ACKNOWLEDGMENTS

My interest in cruise-control problems began in a reading group on software architecture that I organize with David Garlan. Through Marc Graham's efforts, Will Tracz presented an avionics example that led me to look at cruise control. Suzanne Bell's Inspec search yielded a rich harvest of papers. Discussions with Bill Wulf, Roy Weil, Jeannette Wing, Gene Rollins, David Garlan, Nancy Mead, and colleagues at Fisher Controls have improved my understanding of the designs. This research was supported by the Carnegie Mellon University School of Computer Science; by a grant from Siemens Corporate Research; and by the Wright Laboratory, Aeronautical Systems Center, US Air Force Materiel Command, and the Advanced Research Projects Agency under grant F33615-93-1-1330. Views and conclusions contained in this document are mine alone and should not be interpreted as representing the official policies, either expressed or implied, of Wright Laboratory, the Department of Defense, the US government, Siemens Corp., or Carnegie Mellon University. All the figures in this article were reproduced with the permission of the copyright holder; I thank each of them for their cooperation.

REFERENCES

1. M. Shaw et al., *Candidate Model Problems in Software Architecture*, WWW address http://www. cs.cmu.edu/~ModProb/.
2. G. Booch, "Object-Oriented Development," *IEEE Trans. Software Eng.*, Feb. 1986, pp. 211-221.
3. J. Kirby, "Example NRL/SCR Software Requirements for an Automobile Cruise Control and Monitoring System," Tech. Report TR 87-07, Wang Inst. of Graduate Studies., Boston Univ., Tynesboro, Mass., 1987.
4. J. Brackett, "Automobile Cruise Control and Monitoring System Example," Tech. Report TR 87-06, Wang Inst. of Graduate Studies, 1987, distributed by Cadre Technologies, Providence, R.I.
5. W. Yin and M. Tanik, "Reusability in the Real-Time Use of Ada," *Int'l J. Computer Applications in Technology*, No. 2, 1991, pp. 71-78.
6. A. Birchenough and J. Cameron, "JSD and Object-Oriented Design," in *JSP and JSD: The Jackson Approach to Software Development*, J. Cameron, ed., IEEE Press, New York, 1989, pp. 293-304.
7. S. Smith and S. Gerhart, "STATEMATE and Cruise Control: A Case Study," *Proc. Computer Software and Applications Conf.*, IEEE CS Press, Los Alamitos, Calif., 1988, pp. 49-56.
8. J. Atlee and J. Gannon, "State-Based Model Checking of Event-Driven System Requirements," *IEEE Trans. Software Eng.*, Jan. 1993, pp. 24-40.
9. D. Higgins, "Specifying Real-Time/Embedded Systems using Feedback/Control Models," *Proc SMC XII: Twelfth Structured Methods Conf.*, Structured Tech. Assoc., 1987, pp. 127-147.
10. M. Shaw, "Beyond Objects: A Software Design Paradigm Based on Process Control," Tech. Report CMU-CS-94-154, Carnegie Mellon University, Pittsburgh, 1994.
11. P. Ward and D. Keskar, "A Comparison of the Ward/Mellor and Boeing/Hatley Real-Time Methods," *Proc SMC XII: Twelfth Structured Methods Conf.*, Structured Tech. Assoc., 1987, pp. 356-366.
12. D.Garlan and M. Shaw, "An Introduction to Software Architecture," in *Advances in Software Eng. and Knowledge Eng., Volume 2*, V. Ambriola and G. Tortora, eds., World Scientific Publishing, Singapore, 1993.
13. M. Shaw, "Architectural Issues in Software Reuse: It's Not Just the Functionality, It's the Packaging," *Proc. IEEE Symp. Software Reuse*, IEEE Press, New York, 1995.

Mary Shaw is associate dean of professional education programs and Alan J. Perlis professor of computer science at Carnegie Mellon University. She is also a member of the Human Computer Interaction Institute at CMU. She is author or editor of six books and more than 100 papers and technical reports. Her research interests include software architecture, languages, specifications, and abstraction techniques. Her past projects include software architecture and technology transition at SEI.

Shaw received a BA in mathematics (cum laude) from Rice University and a PhD in computer science from CMU. She is a fellow of the IEEE and the American Association for the Advancement of Science, a member of the ACM, and part of WG 2.4, System Implementation Languages, of the International Federation of Information Processing Societies. Additional biographical information is on Shaw's www home page: http://www.cs.cmu.edu/~shaw/

Address questions about this article to Shaw at School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213-3891; mary.shaw@cs.cmu.edu.