

A Process for Specifying Black Box Behavior, Demonstrated in a Case Study

Stephanie White and Herbert Warner
Advanced Technology and Development Center
Northrop Grumman Corporation

Abstract

Prior to designing a system, customers and contractors should agree on required black box (externally apparent) system behavior. To define this behavior, practical, precise, design independent methods are needed. This paper describes results of a case study¹ in which formal event-based approaches are used, demonstrating that a combination of history based traces and guarded event-action statements is practical for defining black box behavior. Externally apparent modes (states) simplify the specification to promote human understanding. The specifier is allowed to use both traces and event-action statements in a single specification, as requirements that define sequential events are best specified using traces, and requirements that are conditional are best specified using event-action statements. Graph generation from the model, as opposed to graph definition makes this type of specification easier to define and maintain.

Problems with current methods

Methods commonly used in industry to model system requirements and design are not adequate for specifying externally apparent (black box) behavior. A specification defining such behavior should identify what a system does (the response) when it receives a specific input (the stimulus). The stimulus may be singular, or may consist of a sequence or set of stimuli. The same can be said for the system response. High level models and methods for specifying interfaces (e.g. context diagrams) do not relate system stimuli to system response. Detailed models that show the transformation from input to output include design decisions. Black box specifications should not include design decisions, as they make the specification harder for customers to understand, and limit designers.

Engineers frequently use Context Diagrams and Interface Control Documents (ICDs) to specify the interface between a system and its environment. Normally (e.g. in structured analysis) Context Diagrams specify information flowing to/from the system, events

taking place in the environment that affect the system, and actions caused by the system that affect the environment, but do not specify this information in sufficient detail or relate stimuli to response. Interface Control Documents specify interfaces in detail, identifying data limits, units of measurement and format, but also do not relate input to output.

Object Oriented Analysis includes Use Case Analysis and Object Interaction Diagrams (graphic traces) that specify the execution of scenarios. Booch [BOOC 94] warns that there are too many scenarios to specify: "no amount of time will be sufficient". Results of our Chrysler Cruise Control System Case Study indicate that the use of guarded event-action statements, canonical traces, and externally apparent states would drastically reduce the number of scenarios, making a complete black box specification feasible.

Traces define black box behavior

The literature, [BART 78, HOAR 85, HOFF 88, PARN 92, WANG 94] describes traces as a method for specifying interfaces. A trace starts with the initial system state, and consists of an alternating sequence of environmental stimuli and system actions that ends with a system action. This sequence of events is equivalent to a sequence of state transitions, and leaves the system finite state machine in a particular state. The trace $E_1. E_2. E_3. \dots E_n$ (where "." means concatenation) denotes both the sequence of events and the final state caused by the event sequence. Since only the output values are externally apparent, an object's externally apparent behavior is defined if the outputs are defined for every trace.

Many traces denote the same state and are thus equivalent. The set of all traces are divided into equivalence classes, where each class contains all traces that start with the initial object state, and end with a particular state, say S_i . That is, all traces that start in state S_0 and end in state S_1 are in one equivalence class, and all traces that start in state S_0 and end in state S_2 are in another. In each equivalence class, a trace called the canonical trace is chosen to represent the class, and is normally the more concise trace.

For a stack, two equivalent traces are:

- (1) push a, push b, push c, pop
- (2) push a, push b

¹ Research funded by Office of Naval Research (ONR) under Engineering of Complex Systems Technology Program, Contract N60921-94-C-0073.

The trace push a, push b would be chosen as the canonical trace.

Wang and Parnas write: "To successfully write a proper trace specification, one of the most important and creative steps is to choose the canonical trace set. Wisely chosen canonical traces can greatly simplify the specification." The concept of canonical traces can reduce costs and increase quality of system testing, even during informal system development. Engineers should choose scenarios with maximum coverage, such that no two scenarios are "equivalent".

A person defining requirements as traces can specify a starting state for a trace that is different from the initial state. This simplifies the representation. A state S can be specified by the set of traces T_i , $i = 1$ to n . The trace T_i can be expressed as a state S_k followed by a set of traces T_j , $j = 1$ to m . As the sequence is generated in reverse, the trace will eventually include the initial state S_0 . A tool could generate the entire trace from the partial traces.

For a trace for the cruise control problem, we may start from the state S_n : Engine On & Speed Okay For Cruise Control Mode. In the Chrysler, this is any speed greater than or equal to thirty miles per hour. Some other trace has to indicate sequences that create the state, starting with the initial state S_0 : Engine Off.

Event-action statements specify black box behavior

In many situations, it is convenient to specify high level system behavior using Event-Action statements with Condition Guards [HENI 80]. These statements, which henceforth will be called Event-Action Statements, are of the form:

Event while Condition \rightarrow Action
A simple condition consists of a noun, a relationship .R, and an associated value:

Condition: Noun .R. Value

(e.g. Accelerator Pedal = Pressed, Date > July 1, 1995, Carol is a member of the Project). A condition can also be compound. Compound conditions are composed of simple conditions, using the operators OR and AND, for example the compound condition: C1 OR C2, (e.g. Accelerator Pedal = Pressed OR Brake Pedal = Pressed). An event is a condition becoming True or False. An action is an event that responds to a stimulus.

Extensions to event-action statements

We have extended Event-Action statements to produce more than one action. These sets of actions may

- (1) happen at approximately the same time and order is unimportant,
- (2) happen at approximately the same time but order is important,

(3) happen at different times, and order is unimportant, but all actions happen prior to some other event,

(4) happen at different times but order is important.

We use the following notation for each situation:

Case 1:

Actions happen at approximately the same time and order is unimportant.

We use the word "AND" to indicate the fact that order is unimportant, and that the two events happen at approximately the same time:

Event while Condition \rightarrow Action 1 AND Action 2

|Action 1 - Action 2| < t indicates the timing requirement.

Case 2:

Actions happen at approximately the same time but order is important. In this case there is a required sequence, and we write:

Event while Condition \rightarrow Action 1

\rightarrow Action 2.

(Action 2 - Action 1) < t indicates the timing requirement. In this case the absolute value notation is unnecessary as the order of the events is known.

Case 3:

Two actions happen at different times and their order is unimportant, but both actions happen prior to a third action.

Event while Condition \rightarrow Action 1 AND Action 2

\rightarrow Action 3.

In this case Action 1 and Action 2 can occur in any order, but both have to complete prior to Action 3. Action 1, Action 2, and Action 3 are partially ordered. If there is a timing requirement, we include one, but timing may not be required in this situation.

Case 4:

Actions happen at different times but order is important. In this case, as in Case 2, there is a required sequence, and we write

Event while Condition \rightarrow Action 1

\rightarrow Action 2.

If there is a timing requirement we include one, but timing may not be required in this situation. Sometimes a sequence of Event-Action Statements are needed to express a particular behavior. In this case, we write

Event while Condition \rightarrow Action 1

Event while (Condition which is a result of Action 1)

\rightarrow Action 2

In [WHIT 95] we use these extensions to model the Chrysler Cruise Control System.

Specification containing traces and event-action statements

Event based notations (traces and event-action statements with condition guards) are used for specifying complex system behavior as they provide good support for specifying temporal information, especially important for Real-Time systems. For

example, event-based methods can identify maximum time between two events, maximum time for a sequence of events to take place, synchronization of events within a specified tolerance, worst case response to stimuli, worst case environment response, duration of time before or after an event, and data senescence.

Specifiers should be able to use the constructs they prefer, and methods should be able to accommodate diverse constructs. For example, specifiers should be able to list a sequence of events that cause an action, or alternately, list an event and existing conditions that cause an action. Methods and automated tools should accept both views. We modeled the externally apparent behavior of a Chrysler Cruise Control System to illustrate that in some cases guarded event-action statements are superior for representing system behavior, and in other cases traces are a more natural representation. We substituted states for sequences of events to simplify the specification, a mathematically sound practice since states represent system history and ordered or partially ordered sequences of events describe that history.

Process

The process for specifying externally apparent system behavior is a well-defined process, consisting of six steps.

- (1) To aid in consistency analysis, define externally apparent states of objects in the environment.
- (2) Define system states that users can recognize when operating the system, that is the externally apparent system states. Group these states according to system activity (e.g. throttle control is "off", "on", "suspended") or output (e.g. throttle value = formula 1, throttle value = formula 2).
- (3) Define the set of allowable state transitions.
- (4) Define actions that the system and environment objects perform.
- (5) Specify all externally apparent system behavior in terms of the system states defined in (2), traces, and guarded event-action statements.
- (6) Verify that all behavior is complete and consistent using model checking techniques. Analysis techniques developed by the author [WHIT 87] perform model checking and determine inconsistencies in extended state transition tables. These methods can be augmented for a model consisting of both guarded event-action statements and traces. Alternately, guarded event-action statements can be automatically transformed to a set of traces, for analysis purposes. In addition, system tautologies (statements that are always true about the system) can be defined and used to check the traces and event-action statements.

Cruise Control System (CCS) case study

Our Cruise Control System (CCS) case study demonstrates that the use of a combination of event-action statements and traces is practical for modeling high level system behavior for medium size systems. The complete CCS model is specified in [WHIT 95]. In this paper we provide a few examples, to demonstrate the process and techniques.

The first step is to define the states of the environment and system objects to aid in consistency analysis. For example the states of the CCS control slide are "Off", "On", and "Resume". The physical states of the Speed Set button are "Pressed", and "Released". When the CCS is active, virtual states include "Tapped", and "Held" as this button is used not only to set a speed, but also to increment speed and decrement speed. The set vehicle speed is either fast enough to engage the CCS (i.e. over 30 mph), or too slow. We call these states "speed_hi" and "speed_lo".

The second step is to define system states which users can recognize when operating the system (externally apparent states). A table is created that specifies states, state descriptions, and conditions that are true in each state. For the Chrysler CCS, there are seven states:

- (1) "OFF"; the CCS is inoperative.
- (2) "ON_inact_speed_null"; the Operator turned the ignition on while the control slide was in On position. The CCS cannot be activated from this state.
- (3) "ON_inact_speed_low"; the Control Slide is set to On but the set speed is too low for activation. The CCS cannot be activated from this state.
- (4) "ON_inact_speed_hi"; the Control Slide is set to On and the set speed is high enough so that CCS can be activated from this state.
- (5) "ON_act_suspend-acc"; the CCS is suspended because the accelerator pedal is pressed; the CCS will return to the active state when the accelerator is released, conditions permitting.
- (6) "ON_act_suspend-set"; the CCS is suspended because the Speed Set Button is held, but will return to the active state when the button is released, other conditions permitting.
- (7) ON_active; the CCS is applying accelerator pressure to control the vehicle speed.

For each state, conditions are specified that are always true in that state. For CCS = "Off", true conditions are: Control Slide = OFF OR Ignition SW = OFF, and Remembered Speed = Null. For CCS = "ON-active", true conditions are Ignition SW = ON, Control slide = ON, Remembered Speed > 30, SpeedSETbutton <> Hold, accelerator pedal = released, brake pedal = released, clutch pedal = released, and vehicle speed > 28 mph (allowing for a 2 mph tolerance in Speed Control).

Third, a state transition matrix specifying all possible transitions is constructed, see Fig. 1 for a subset of this matrix.

State Name	#	To				
		4	5	6	7	
ON_inact_speed_hi	4		X		X	
ON_act_suspend-acc	5	X			X	
ON_act_suspend-set	6	X			X	
ON-active	7	X	X	X		

From

Fig. 1. Subset of CCS State Transition Matrix

In the matrix all possible transitions for the CCS are listed. For example, this matrix indicates that it is possible to transition from (7) "On-Active" to:

- (4) "ON_inact_speed_hi", the CCS is inactive because the brake or clutch pedal was pressed;
- (5) "ON_act_suspend-acc", the CCS is suspended because the accelerator pedal is pressed; or
- (6) "ON_act_suspend-set"; the CCS is suspended because the Speed Set Button is held.

Fourth, we specify actions that environment objects and the system performs. For example, the operator sets the speed button, and presses the accelerator, brake or clutch. The system makes sure that vehicle speed is controlled so that it is approximately equal to Remembered Speed, and determines when this control should be operable.

Fifth, we specify the event-action statements and traces.

According to the car Owner's Manual one action the user wishes to perform is to Activate the Cruise Control System.

The owner manual specifies:

"When the vehicle has reached the desired speed, push the SET button to move the control slide to the ON position. This will establish memory and activate the system. Remove your foot from the accelerator."

This statement is not precise. Several assumptions have to be made:

- (1) Speed is captured when the SET button is pushed. Button release time does not matter.
- (2) If any pedal is depressed, the Cruise Control System is not in control of the car's speed.
- (3) For the Cruise Control System to operate, vehicle speed must be greater than 30 miles per hour.

This specification requires two event-action statements, which result in a simpler specification than the equivalent trace. Cruise Control System activation is achieved by Event-Action Statement 1 followed by Event-Action Statement 2, or by Event-Action Statement 3, as follows:

Accelerating to desired speed:
Event Action Statement 1 (phase 1)

```
(speed SET button pressed)op_ev
while
  (CCS = Off)sys_cond
  AND [(vehicle speed > 30mph)env_cond
  AND (accelerator = pressed)op_cond
  AND (brake pedal = released)op_cond
  AND (clutch pedal = released)op_cond]
--> (control slide position = ON)sys_ev
--> (remembered speed = current vehicle speed)sys_ev
--> (CCS = ON_act_suspend_acc)sys_ev
```

Event Action Statement 2 (phase 2)

```
(accelerator pedal = released)op_ev
while
  [(CCS = ON_act_suspend_acc)sys_cond]
--> (CCS = ON_active)sys_ev
```

Already at desired speed and no pedals pressed: Event Action Statement 3

```
(Speed SET button pressed)op_ev
while
  (CCS = Off)sys_cond
  AND (vehicle speed > 30mph)env_cond
  AND (all pedals = released)op_cond
--> (control slide position = ON)sys_ev
--> (remembered speed = vehicle speed)sys_ev
--> (CCS = ON_active)sys_ev
```

The specifier may decide to use a Behavior Net rather than Event-Action Statements or Traces. Behavior Nets identify events rather than functions but are otherwise similar to F-nets in [ALFO 85]. The Behavior Net in Fig. 2 corresponds to Event-Action Statement 1 followed by Event-Action Statement 2.

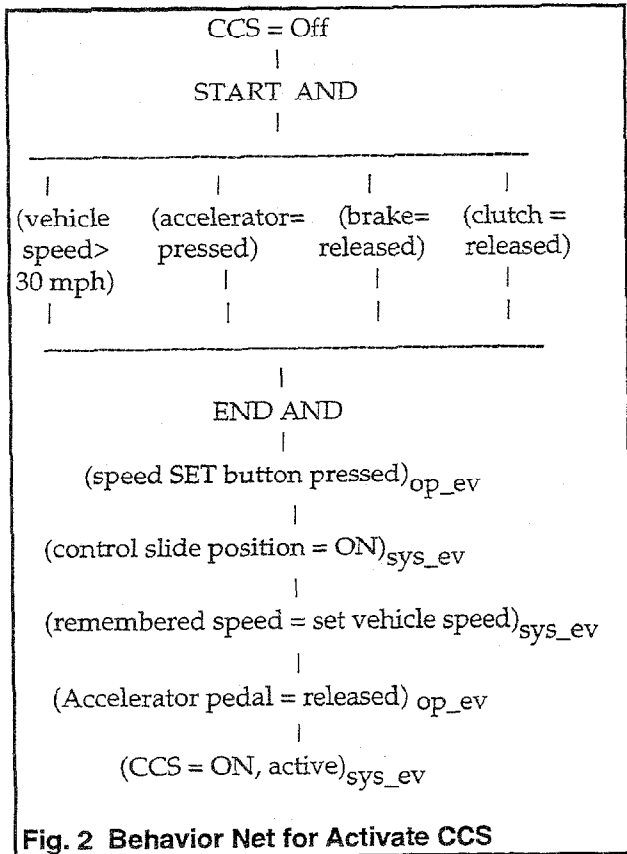


Fig. 2 Behavior Net for Activate CCS

Another CCS requirement follows:

The car operator wants to accelerate for passing, while the cruise control is active. The manual specifies:

"Depress the accelerator as you would normally. When the pedal is released, the vehicle will return to the set speed."

Event-action statements result in a more complex specification than traces. Two statements are needed for the two phases

Event-Action Statement (phase 1)

```

(accelerator pedal = pressed) op_ev
WHILE
  (CCS = ON_active) sys_ev -->
  (CCS = ON_act_suspend_acc) sys_ev
  
```

** vehicle accelerating under driver's control of accelerator pedal

Event-Action Statement (phase 2)

```

(all pedals = released) env_cond
  
```

** begin vehicle slowing

```

WHILE (CCS = ON_act_suspend_acc) sys_ev
--> (CCS = ON_active) sys_ev
  
```

** at original vehicle speed

```

--> (vehicle speed = set vehicle speed)
  
```

** takes time to accomplish

The trace is simpler than the two event-action statements as there is no need to be concerned that additional events have occurred between the two separate statements.

Trace

```

Start State: (CCS = ON_active)
(accelerator pedal = PRESSED) op_ev ;
(CCS = ON_act_suspend_acc) sys_ev ;
  
```

** vehicle speed= increasing

```

(accelerator pedal = released) op_ev ;
  
```

** vehicle speed= decreasing

```

(CCS = ON_active) sys_ev ;
(vehicle speed = set vehicle speed) env_ev
  
```

** takes time to accomplish

Timing is also an aspect of such specifications. The manual specifies:

"When the system is activated, tapping the Speed SET button will increase the speed settings by 2 mph."

Assume a Speed SET button "tap" is a press-release sequence of not greater than ST seconds.

In the Event-Action Statement that follows "T(event A) - T(event B) ..." indicates the time interval between initial event B and the following event A.

Event-Action Statement

```

([ T(Speed SET button = released)
- T(Speed SET button = pressed) <= ST]) op_ev
  
```

** a press-release sequence that qualifies as a "tap"

```

WHILE
  (CCS = ON_active) sys_cond
  -->
  
```

```

(remembered speed = remembered speed + 2 MPH) sys_ev
  
```

** no change in the system state, but the speed it will hold is 2 MPH more

The sixth and last step is specification analysis. Here we use tautologies to check the specification. An analyzer can check that all of the event-action statements and traces are consistent with the tautologies, and warn of any inconsistencies.

For the Cruise Control System, the following tautologies (universal truths) exist:

(1) The Operator automatically positions the Control Slide to "On" when pushing the SET button; thus these two events are simultaneous.

This is written formally as:

ALWAYS [(SET button pressed) \rightarrow (control slide position = ON)]

ALWAYS $\{T(\text{speed SET button pressed})_{\text{op_ev}} - T(\text{control slide position = ON})_{\text{sys_ev}}\} < \text{epsilon}$,

where $T(\text{event})$ is the time at which that event occurs, and epsilon is a very small positive number.

(2) The conditions that are always true for a particular state represent tautologies. For example, when the Cruise Control System is Off, "remembered speed" is null. This is written formally as

ALWAYS [(CCS = OFF) \rightarrow (remembered speed = null)].

Also, when the Cruise Control System is On and Active, the remembered speed is a vehicle speed over 30 miles an hour.

This is written formally as

ALWAYS [(CCS = On_Active) \rightarrow (remembered speed > 30 mph)].

(3) When a condition is always true for a particular state, alternate conditions are always false. Since

ALWAYS [(CCS = On_Active) \rightarrow (remembered speed = hi)] and
Remembered speed has values "null", "low", or "hi".

the following is true:

NEVER [(remembered speed = null) AND (CCS = ON_active)],
NEVER [(remembered speed = low) AND (CCS = ON_active)].

From the operator's point of view, this tautology states that if the operator does not provide a valid speed via the SET button, the Cruise Control System cannot be in control.

Any trace or event-action statement that negates a tautology is illegal. A tool could generate traces from Event-Action Statements to simplify the analysis.

Conclusions

High level system behavior is behavior that is apparent in the system environment. It is important when modeling such behavior, to use methods that are design independent. These methods should encourage the designer to provide only externally apparent behavior at this level. The paper demonstrates the use of guarded event-action statements, traces, and behavior graphs in modeling the externally apparent behavior of a real system, a Chrysler automobile Cruise Control System. The use of this combination of formal methods looks very promising. Future research should apply behavior graphs more extensively. Larger, more complex applications should be modeled.

References

- (ALFO 85) Alford, M., "SREM at the Age of Eight: The Distributed Computing Design System", *IEEE Computer*, Vol. 18 No. 4, April 1985, pp. 36-46.
- (BART 78) Bartussek W. and Parnas D., "Using Assertions About Traces to Write Abstract Specifications for Software Modules", *Information Systems Methodology*, in *Lecture Notes in Computer Science* 65, pp. 211-236, 1978.
- (BOOC 94) Booch, G., *Object Oriented Analysis and Design with Applications*, the Benjamin Cummings Publishing Co., Santa Clara, CA., 1994
- (HENI 80) Heninger K., "Specifying Software Requirements for Complex Systems: New Techniques and Their Application", *IEEE Trans. on Software Engineering*, Vol. SE-6, Jan 1980, pp. 2-13.
- (HOAR 85) Hoare, C.A., *Communicating Sequential Processes*, Prentice Hall, 1985.
- (HOFF 88) Hoffman, D. and Snodgrass, R., "Trace Specifications: Methodology and Models", *IEEE Transactions on Software Engineering*, Vol. 14, No. 9, September 1988, pp. 1243-1251.
- (PARN 92) Parnas, D. L. and Wang, Y., *The Trace Assertion Method of Module Interface Specification*, CRL Report No. 244, McMaster Univ., May 1992.
- (WANG 94) Wang, Y. and Parnas, D. L., "Simulating the Behavior of Software Modules by Trace Rewriting", *IEEE Transactions on Software Eng.*, Vol. 20., No. 10, Oct. 1994.
- (WHIT 87) White, S. *A Pragmatic Formal Method for Computer System Definition*, Ph.D. Dissertation, University Microfilms, Ann Arbor, Michigan, 1987.
- (WHIT 95) White, S. and Warner, H., *VISIBLE, A Method, for Specifying System Behavior*, Report No. ECS-95-001, Northrop Grumman, Nov. 1995.