Doron Drusinsky-Yoresh

Heterogeneous State Machine Control

Automotive electronics have taken on a life of their own. Their complex interprocessing requires engineers to mix classic control with continuous modeling tasks. Doron shows how statecharts ease the difficulties.

Clearly, automotive electronics have gone through a major evolution within the last decade. Processors now perform dozens of tasks that are completely new or were previously managed using analog or mechanical technologies.

The well-known advantages of digital control and computation include flexibility, diversity, low cost, and rapid development. With the proliferation of digital processors in vehicles, new challenges are emerging, like management, synchronization, and scheduling of heterogeneous computational tasks.

In this article, I describe how complex state-machine control tasks, classical control tasks, and continuous modeling tasks combine to achieve the heterogeneous solution necessary in practical automotive applications.

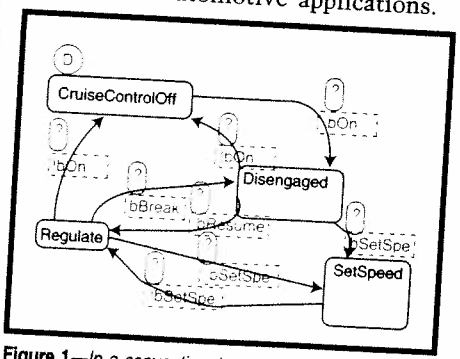

Figure 1—*In a conventional cruise-control state diagram, there is no state nesting. The cruise-control state machine maps input to output sequences.*

..., they describe the p... ...ssion of the system with time, as inputs keep arriving into the system and outputs need to be produced by it.

For example, the state diagram in Figure 1 depicts a simple cruise-control machine, which changes states as inputs from other parts of the car are accepted.

Such state diagrams are well-known, well-accepted, highly visual, and intuitive. Their ability to describe finite and infinite sequences, combined with their visual appeal, has made them one of the most commonly accepted formalisms in the electronics industry.

State diagrams are easier to design, comprehend, modify, and document than a textual approach. But, they haven't changed much over the last 30 years, and they're limited when applied to modern reactive applications.

One problem is that state diagrams are flat. They can't handle modern top-down design and information hiding. Top-down design concepts require interactive software so the user can manipulate and browse complex designs.

Another disadvantage is that state diagrams are purely sequential, whereas applications are not. Modern state-machine controllers need to react to signals going to and coming from a plurality of entities in their environment. Also, a single state-machine controller might be responsible for many independent or partially independent conceptual tasks, thereby performing "conceptual concurrence."

Compensating for these limitations are statecharts, designed by David Harel [1]. While addressing the hierarchy and concurrence problems of state diagrams, statecharts retain the visual and intuitive appeal inherent to state diagrams.

The statechart in Figure 2 describes a more advanced cruise-control design than the state diagram of Figure 1. Note how the *CruiseControlOn* state encapsulates lower level states, thereby creating state hierarchy.

An important property of such hierarchical representation is that the high-level transition (labeled *!bOn*) from *CruiseControlOn* to *CruiseControlOff*

regardless of the actual present state within *CruiseControlOn* and its descendants.

In fact, if the design is enhanced with more states within *CruiseControlOn* later in the design process, the high-level transition automatically encapsulates these newly added states, without any other explicit change to the design.

Note how the contents of the *Regulate* state are pushed onto a separate design page to provide more design space as we dive into lower levels of hierarchy. I'll discuss concurrence—the ability to describe multiple, simultaneous, or independent substatecharts—later when I extend the cruise-control example.

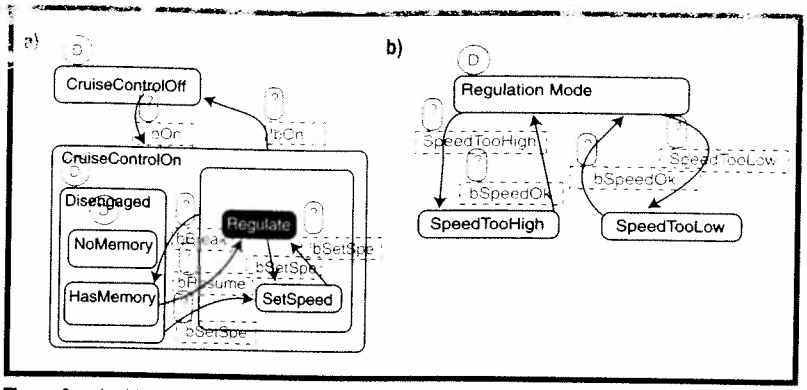

Figure 2a—*In this cruise-control statechart, state nesting exists within the* CruiseControlOn, Active, Disengaged, *and* Regulate *states. The* Regulate *state has nested states that are currently hidden from view.* **b**—*The contents of the* Regulate *state provide yet another example of state nesting.*

A HETEROGENEOUS EXAMPLE

Automotive applications often require that complex statechart designs be integrated with other types of computations, thereby creating heterogeneous (or hybrid) systems.

In my example, the cruise-control statechart of Figure 2 is extended to account for an event in the transmission box where the gear shifts from D (drive) to N (neutral), while cruise control is regulating (i.e., is in the *Regulate* state) the speed.

This event forces the statechart to change state to *Disengaged*. Indeed, it's a common problem in cruise-control–equipped vehicles that when the gear changes to N, the cruise-control logic attempts to accelerate, even though the transmission is not engaged. This action sharply increases the engine RPM.

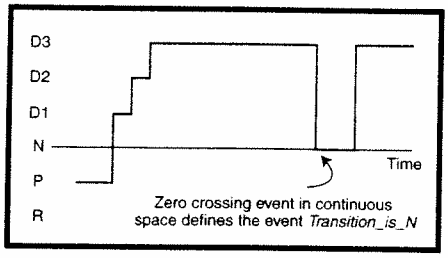

Figure 4—*This diagram illustrates a continuous waveform for the transmission box, from which the* Transmission_is_N *condition is generated.*

Figure 3 includes a simple modification to Figure 2, where the *Transmission_is_N* event causes a transition from the *Active* to *HasMemory* state. This transition disengages the cruise-control operation when the gear is set to N.

As illustrated in Figure 4, the *Transmission_is_N* event is generated from a continuous model, where the event occurs on a zero-crossing of the continuous signal. The interesting—and potentially difficult—aspect of this enhancement occurs with the real-world implementation of a statechart as a block of code and the progression of time.

Whether the statechart is implemented in C, C++, Java, or some other form of code, it will be invoked (i.e., called or activated) repeatedly in the real world. Each time it is invoked, it computes its new state or states (if the statechart has hierarchy or concurrence) given the previous state(s).

This repeated calling scheme can be periodic, as illustrated by the shorter arrows in Figure 5, or based on an event (e.g., every time a certain interrupt occurs).

There will always be periods of time between successive invocations of the statechart block of code in the real world. More often than not, the change in gear event (i.e., the gear shifting from D to N) occurs in some time slot between two successive invocations of the statechart code, as pictured in Figure 5.

There will be some period of time between the gear event and the next closest invocation of the statechart code that can use this information and disengage the cruise control. Let's denote this time period as an "unstable period."

Clearly, it's important that an event generated from the continuous representation of the transmission box trigger an invocation of the statechart code. Otherwise, the statechart of Figure 4 behaves improperly during its unstable period. This ability is the essence of a heterogeneous design, for simulation and implementation purposes alike.

The statechart in Figure 6 introduces concurrence. In this modified cruise-control example, there are two separate threads (depicted as dashed boxes)—the original *CruiseCntl* statechart and, simultaneously, the *Transmission* thread, which is entirely event based. *Transmission* changes

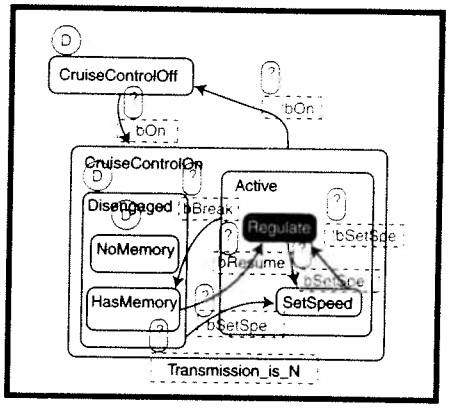

Figure 3—*This is the top-level enhanced cruise-control statechart. Note the* Transmission_is_N *transition from the* Active *state to the* HasMemory *state. The condition* Transmission_is_N *is generated by a continuous model.*

states as events from the continuous transmission model of Figure 4 occur.

Transmission constrains the allowable state changes in the transmission box. This thread performs a task that is mostly independent of the *CruiseCntl* thread, so it is described as concurrent.

However, some dependencies exist, such as the *Transmission* thread forcing *CruiseCntl* into the *Disengaged/HasMemory* state when the transmission's state becomes N. Visual synchronization—the ability to visually show dependencies between threads—is another powerful feature of statecharts.

HETEROGENEOUS DESIGN

As I mentioned, designing critical automotive control systems requires a

[illegible] necessary to support high-fidelity process behavioral modeling, graphical software programming, design verification via simulation, and software specification and implementation via automatic code generation.

One helpful tool is BetterState, which aids in statechart design, code generation, and visual debugging. It supports all features of Harel statecharts, plus some extra capabilities like visual synchronization, visual priorities, critical regions, and mixing flowcharts and statecharts.

BetterState's automatic code generator produces code in C, C++, Java, Perl, Visual Basic, VHDL, Verilog HDL, Delphi, and special real-time OS (RTOS) code for embedded controllers.

As well, developers can use certain features to customize their code generator to write special-style code or code in another language. For example, a code generator that makes assembly code for the Motorola 68k processor is available.

Visual-debugging tools enable you to observe the statecharts' behavior in run time using state animation. This animation works even when the generated code is on a target embedded CPU.

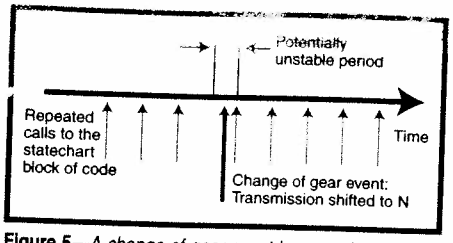

Figure 5—*A change-of-gear event happens in a potentially unstable period.*

To achieve the support goals mentioned above, I interfaced BetterState with MATRIX$_x$—a visual modeling, design, simulation, and implementation tool for large-scale control systems.

Each algorithm is described in MATRIX$_x$ and then automatically coded in C. The set of generated procedures is introduced in BetterState and activated as defined by the statechart. The combination of the MATRIX$_x$ algorithm and the statechart design creates the complete controller logic.

With the integrated environment, the design can be shared among team members without requiring any transfer [illegible] verification and validation. This phas[illegible] is first performed using the MATRIX$_x$ SystemBuild simulator.

The controller model can be tested against a process behavioral model also designed in MATRIX$_x$. Interactive animation tools can debug and interact with the design. Once the simulations execute satisfactorily, the automatic code generation can be performed.

The automatic implementation makes optimized code both from SystemBuild and BetterState. Concurrence and hierarchy describe a conceptually huge Cartesian product state space representing all combinations of states within concurrent threads anywhere in the state hierarchy.

However, the compact code generated by the code generator doesn't blow up state space at all. Rather, its size grows linearly with the number of transitions in the statechart. Time saved in code generation lets developers focus on software architecture and unit testing.

BENEFITS

Using such an heterogeneous environment for automotive control system design and implementation creates a work environment that facilitates workgroup communication, code reusability, and code-error reduction, while maintaining compact and efficient code.

More attention can be given to critical aspects of embedded software design such as software architecture and testing, which results in better quality and helps meet time-to-market agendas.

CODE GENERATION FOR RTOS

Many, if not most, real-time applications contain a reactive component that deals with sequences of inputs from and outputs to the environment surrounding the application. For example, a cellphone application requires a reactive component that deals with sequences of push-button events made by the user or with sequences of network commands received over the channel.

Such reactive components are almost always designed using a state-machine approach. Statechart and state-diagram code generation for a real-time OS needs to be robust in the presence of interrupts and other forms of reentrancy.

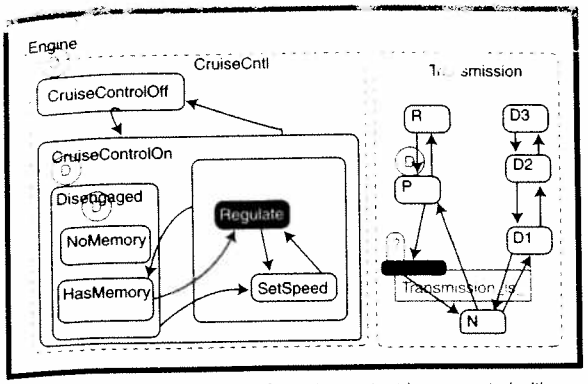

Figure 6—*This time, the Cruise Control statechart is augmented with concurrence (condition names are omitted for clarity). Note how* CruiseCntl *and* Transmission *are independent substatecharts in their own right.*

Typical code generators, whether for statecharts or conventional state diagrams, assume that the generated code (typically wrapped inside a function) completes executing before being called again.

This assumption is valid for simulation purposes. However, it doesn't hold when the generated code is invoked by interrupts or some other preemptive mechanism, where the current execution might be interrupted by another call due to a subsequent interrupt.

To address such concerns, BetterState offers a robust code generator tailored for pSOS and other RTOSs.

AN EMERGING STANDARD

Statecharts have been chosen as the de facto and de jure standards in many industries (e.g., SEMI, CASE/OOD, aerospace, and EDA). And recently, statecharts have been emerging as an important language in the automotive market as well.

In this article, I've shown how statecharts can be used in a heterogeneous design environment that combines statecharts and state-machine control with continuous-time control simulations and code generation.

It lets designers describe, simulate, and implement automotive applications such as engine control and antilock braking as well as complex cruise-control and body-logic applications.

Doron Drusinsky received his Ph.D. from the Weizmann Institute, Rehovot, Israel. He developed statechart CAD tools and DSP applications for Sony, and in 1993, he founded R-Active Concepts and developed BetterState. You may reach him at doron@isi.com.

REFERENCE

[1] D. Harel, "Statecharts: A Visual Approach to Complex Systems," *Science of Computer Programming*, **8**, 231–274, 1987.

SOURCE

BetterState, MATRIX$_x$
Integrated Systems, Inc.
201 Moffett Park Dr.
Sunnyvale, CA 94089
(408) 542-1500
Fax: (408) 542-1955
www.isi.com

I R S

407 Very Useful
408 Moderately Useful
409 Not Useful