Agent-oriented Modelling for a Vehicle Cruise Control System

W M Beynon
I Bridge
Y P Yung

Dept of Computer Science
University of Warwick
Coventry CV4 7AL
UK

ABSTRACT

Specifying a vehicle cruise controller is a standard case study for software development methods. We apply a new agent-oriented modelling technique to this problem to animate the requirement for testing. Our approach relates reactive system modelling to the identification of characteristic sets of observations. Its relationship to other approaches is briefly discussed.

BACKGROUND

Good engineering practice involves understanding the relationship between general scientific and engineering knowledge and a particular physical construction. Relevant knowledge includes properties of materials (e.g. Young's moduli, expansion coefficients, melting points), physical laws, characteristics of components (e.g. response times, power output) and mechanical principles. General knowledge of this kind can be acquired by experiment independently of a specific engineering task.

Analysing and solving a particular engineering problem involves identifying how to apply general knowledge. For instance, predicting safe limits for loading requires knowledge of both general characteristics of the construction material and of where the greatest stresses occur. Relating the performance of an engineering device or system to general scientific knowledge and environmental assumptions in this way is a most important aspect of engineering. Understanding this relationship enables the engineer to justify the design, to make modifications, to diagnose faults and to enhance reliability.

Current software specification practice doesn't really support engineering in this sense. The goal of software specification is to define a formal mathematical model for system behaviour that can be interpreted independently of the physical system to be built. In a recent paper [9], Harel has argued that effective methods for developing reactive systems can be based upon mathematical models of this nature. By implication, this suggests that it is not necessary to construct computer models that closely reflect the architectural and physical characteristics of reactive systems in order to analyse their behaviour effectively in engineering terms. Techniques for deriving programs from abstract specifications of their behaviour in the simpler context of "one-person programming" [9] have probably been a major influence in shaping this view.

In this paper, we present an approach to reactive systems modelling that has more in common with traditional engineering. An engineer will relate the successful development of a system to precise knowledge of how components interact and of what physical laws operate. We shall describe and illustrate a method of modelling a reactive system that is closely linked with the task of eliciting and documenting this engineering knowledge. The principles we adopt for knowledge representation are outside the logicist framework; in particular, we invoke classical physical models rather than "naive physics". Our approach can be viewed in two ways: as an informal method of animating requirements for testing or, more pretentiously and contentiously, as a principled alternative method of reactive systems specification. A fuller discussion of this issue, in the form of a commentary on [13], appears in [1].

Our case study is a vehicle cruise control system that has been used as an example in previous software specification studies [6,7,8]. Our model has been extended to include the vehicle dynamics. It is specified using the agent-oriented

notation LSD [2,3] and animated using the Scout-Donald-Eden environment for visualisation (cf [4]).

Our approach can be viewed as an extension of the scenario-oriented approach of Deutsch, but uses a different underlying computational paradigm that is not object-based. This paradigm shares the original philosophy behind object-oriented programming, as in Simula [5]: programming is a particular form of system description. We follow Deutsch's dictum [7]: "fundamental scenarios describing the behaviour of objects are more significant to the system engineer than the characteristics of objects in isolation". The essence of our method is faithful modelling of the behaviour of objects in combination as reflected in indivisibly associated changes in observations. [Here "observation" is used in a broad sense – closer to its scientific meaning, rather than its everyday meaning – to refer to anything that could conceptually be observed by experiment.] The principles behind this approach will be described in more detail in connection with our example.

1. THE VEHICLE CRUISE CONTROLLER

The vehicle to be modelled has a cruise controller that maintains a set speed under varying load conditions. The cruise control system is an embedded system within the vehicle – the primary objective of our model is to specify its operation in the context of the "engine-vehicle-driver" environment. The user plays the role of the driver in the animation. Engine and vehicle have approximate dynamical models adequate for animating representative transactions.

Our specification makes use of the agent-oriented modelling notation LSD [3] – see Listing 1. In this specification the agents that define the dynamic model are: throttle_manager, engine, vehicle, speed_transducer and environment. These agents are complemented by driver and cruise_cutout agents that define the interface.

The throttle_manager determines the engine throttle position. Its mode of operation (off, manual or automatic) is determined by the status of the engine and cruise controller. In manual mode, the throttle position is specified by the accelerator position. In automatic mode, when the speed is being maintained by the cruise controller, the throttle position is determined by a control law based on the discrepancy between the speed as measured and the intended cruise speed.

The engine model is grossly simplified: vehicle traction power is presumed directly proportional to throttle position. This discounts effects such as time lag between changes in throttle position and traction power, gearbox transmission efficiency, engine characteristic torque vs rpm profiles, gear selection criteria.

The vehicle model is specified by the Newtonian equations of motion. Retarding forces due to wind, rolling resistance and braking act on the vehicle. The road gradient is an environmental factor that may advance or retard motion. The traction force is determined by the engine power and the vehicle speed. These forces determine the vehicle acceleration; this is integrated to determine the speed. This discounts the effect of braking by the engine.

The speed_transducer model presumes that a transducer on a wheel emits one pulse per revolution; the speed is measured by a counter/timer that estimates pulse-rate. With this model of the speed transducer, it is possible to assess the effect of quantising the measured distance into multiples of the wheel circumference and of finite counter length.

The environment is modelled by a road gradient parameter. Gradient is treated as a random environmental factor, though it is a function of vehicle position and direction.

Listing 1 illustrates most of the principal features of the LSD notation. The **agent** specifications define the characteristics of agent instances that are in general dynamically created and destroyed during simulation. The simulation model comprises one persistent instance of each agent apart from the autoThrottle_manager(); this agent exists when its special variable LIVE is **true** i.e. when the throttle is operating in automatic mode (throttleStts==tsAuto). The value of initAutoThrottle is initialised to the current position of the accelerator on each invocation of this agent. Appropriate initial conditions for simulation presume that the car is at rest on level ground with the engine switched off.

The **state** variables[1] are the variables owned by an agent. Owned variables represent *observations* associated with an agent instance, in a sense to be developed below. Variables bound to an agent instance may be directly manipulated by another agent: for example, the driver agent can change the engine status (esOn / esOff) – cf an object-oriented paradigm.

The LSD specification can be viewed as describing the stimulus-response patterns in the application (cf [7,8]). The **oracle** variables represent values to which an agent responds. These variables are typically – but not necessarily – owned by other agents. Their values may be transmitted to the agent in several ways: e.g. via a communication channel, a mechanical linkage, or through sensory input. The variable throttlePos in the engine agent is an example of such a variable.

The stimulus-response patterns for an agent are modelled in two ways, according to how stimuli and responses are coupled. The **derivate** variables are used to represent stimulus-response relationships that are indivisibly coupled in a sense to be explained. The definitions of physical forces acting upon the vehicle are of this nature; any change in the speed of the vehicle is reflected in a simultaneous change of wind resistance and traction force.

---

[1]The syntactic conventions adopted for the LSD notation in this paper differ from those used in previous papers, but the essential concepts are unchanged.

Looser coupling of stimulus and response can be modelled in an agent **protocol**. This consists of a set of guarded actions, each of which takes the form of an enabling condition and an associated sequence of variable redefinitions or agent instance invocations. Each guarded action is viewed as expressing a privilege to act: if an enabling condition pertains, a particular action may be performed. In interpreting a protocol for simulation purposes, application specific assumptions are invoked to model the way in which an agent exercises its privileges for action [3]. There is no general principle to decide which action to perform when there is non-determinism (i.e. two or more enabling conditions hold), nor is it always appropriate to presume that a privilege that is enabled will be exercised. The driver agent specification illustrates this principle.

```
typedef
   cruiseStts_Type  = enum (csOn, csMaintain, csOff)
   throttleStts_Type = enum (tsOff, tsMan, tsAuto)
   engineStts_Type = enum (esOn, esOff)

agent vehicle {
   const
      mass = 1500 /* total mass of car & contents [kg] */
      windK = 10   /* wind resistance factor [N m-2 s2] */
      rollK = 100   /* rolling resistance factor [N m-1 s] */
      gravK = 9.81 /* acceleration due to gravity [N m2 s2] */
      brakK = 150  /* braking constant [N m-1 s] */
   state
      actSpeed  : analog   /* actual speed */
      accel     : analog   /* acceleration */
      windF     : analog   /* wind resistance force */
      rollF     : analog   /* rolling resistance force */
      gradF     : analog   /* gradient force */
      tracF     : analog   /* engine traction force */
      brakF     : analog   /* braking resistance force */
      brakePos  : analog (0.0, 1.0)   /* normalised */
      accelPos  : analog (0.0, 1.0)   /*   position   */
   oracle       brakePos
   derivate
      windF   = windK * actSpeed2
      rollF   = rollK * actSpeed
      gradF   = gravK * mass * sin ( gradient * π / 200 )
      brakF   = brakK * actSpeed * brakePos
      tracF   = enginePower / actSpeed
      accel   = (tracF–brakF–gradF–rollF–windF) / mass
      actSpeed  = integ_wrt_time (accel, 0)
}

agent speed_transducer {
   const
      wheelDiam     = 0.45    /* wheel diameter [m] */
      wheelCirc     = π * wheelDiam
                              /* wheel circumference [m] */
      countPeriod   = 0.2     /* counter/timer period */
      maxCountVal  = 65535  /* 16-bit counter */
   state
      measSpeed : analog
      pulseRate    : analog  /* wheel revs / sec [s-1] */
      countVal     : analog  /* counter/timer value [s-1] */
   derivate
      pulseRate    = actSpeed div wheelCirc
                              /* integer division */
      countVal     = (pulseRate * countPeriod) mod
                        maxCountVal
      measSpeed = (countVal * wheelCirc) / countPeriod
}

agent throttle_manager {

   agent autoThrottle_manager {
      const  initAutoThrottle = |accelPos|
      state
         deltaAutoThrottle   : analog
         autoThrottle        : analog
         speedErr            : analog
      derivate
         LIVE = (throttleStts == tsAuto)
         speedErr = cruiseSpeed - measSpeed
         deltaAutoThrottle = ((GainK * speedErr) –
                        autoThrottle) / TimeK
         autoThrottle = integ_wrt_time (deltaAutoThrottle,
                        initAutoThrottle)
   }

   const
      GainK = 0.5  /* auto throttle controller gain */
      TimeK = 2.0  /* auto throttle controller time const. */
   state
      throttleStts   : throttleStts_Type
      throttlePos    : analog (0.0, 1.0)
                        /* normalised position */
   oracle
      measSpeed, cruiseSpeed,
      cruiseStts, engineStts, accelPos
   derivate
      throttlePos =   (throttleStts == tsOff) ? 0.0 :
                      (throttleStts == tsMan) ? accelPos :
                      (throttleStts == tsAuto) ?
                      min(max(autoThrottle, accelPos), 1)
   handle  throttleStts
   protocol
      (engineStts == esOff) → throttleStts = tsOff
      (cruiseStts != csMaintain) ∧ (engineStts == esOn) →
                      throttleStts = tsMan
      (cruiseStts == csMaintain) ∧ (engineStts == esOn) →
                      throttleStts = tsAuto
}
```

```
agent engine {
   const    maxEnginePower = 74500
                        /* watts (i.e. approx 100 hp) */
   state
      engineStts     : engineStts_Type
      enginePower    : analog
   oracle
      throttlePos    : analog
   derivate
      enginePower = maxEnginePower * throttlePos
}

agent environment {
   state    gradient : analog (-25.0, 25.0)
                        /* realistic gradient [%] */
   derivate
      gradient = rand ( analog (-25.0, 25.0) )
}

agent cruise_cutout {
   const    minCruiseSpeed = 20.0   /* miles/hour */
   state    cruiseStts     = csOff     : cruiseStts_Type
            cruiseSpeed    = minCruiseSpeed : analog
   handle   cruiseStts
   oracle   cruiseStts, brakePos
   derivate
      braking = brakePos != 0.0
   protocol
      braking ∧ cruiseStts == csMaintain →
         cruiseStts = csOn                  /* brake */
}

agent driver {
   const    maxCruiseSpeed  = 70.0    /* miles/hour */
            minCruiseSpeed  = 20.0    /* miles/hour */
   handle   engineStts, cruiseStts, cruiseSpeed
   oracle
      engineStts, cruiseStts,
      cruiseSpeed, measSpeed, brakePos
   derivate
      brakePos = user_input (brakePos_Type)
      accelPos = user_input (accelPos_Type)
   protocol
      engineStts == esOff → engineStts = esOn      /* on */
      engineStts == esOn → engineStts = esOff      /* off */
      cruiseStts != csOff → cruiseStts = csOff
                        /* switch off cruise controller */
      cruiseStts == csOff → cruiseStts = csOn
                        /* switch on cruise controller */
      cruiseStts == csMaintain → cruiseStts = csOn
                        /* return to manual control */
      cruiseStts == csOn → cruiseStts = csMaintain
                        /* resume to cruiseSpeed */
      cruiseStts == csOn → /* maintain the current speed */
         cruiseSpeed = | measSpeed |;
         cruiseStts = csMaintain
      cruiseStts != csOff ∧ cruiseSpeed < maxCruiseSpeed
         → cruiseSpeed = | cruiseSpeed | + 1
                        /* increase cruiseSpeed */
      cruiseStts != csOff ∧ cruiseSpeed > minCruiseSpeed
         → cruiseSpeed = | cruiseSpeed | − 1
                        /* decrease cruiseSpeed */
}
```

Listing 1

2. INTERPRETING THE LSD SPECIFICATION

An LSD specification is intended to document the way in which the behaviour of a system depends upon the characteristics and interrelationship of its components. This information is expressed in terms of the observations of the system that define the role of each agent. An "observation" refers to a measurement that could in principle be tested by experiment. The constants and state variables of the vehicle agent in Listing 1 are illustrative examples.

The way in which stimulus-response pairs are modelled reflects the convention that the designer adopts in making observations of the system. A **derivate** variable is appropriate when this convention precludes observing the system in a state in which the variable and its defining formula have distinct values. An action models a situation in which the system can be observed after a stimulus has been received (i.e. when an enabling condition is satisfied), but prior to response (i.e. before a commitment to act has been made).

The classification of variables in an LSD specification reflects an important distinction between different kinds of observation. The behaviour of the vehicle depends intrinsically upon its mass: no engineering is needed to ensure that the speed of the vehicle is affected by a change in mass. In contrast, the position of the brake affects the dynamics of the vehicle via a mechanism for communication, such as a mechanical linkage. The classification of brakePos as an oracle to the vehicle agent indicates that it is potentially an unreliable observation, subject to qualification as the system model is refined.

The specification of the speed_transducer illustrates the refinement principle. This transducer is the engineering component that converts the actual speed of the vehicle into a measured speed that is communicated to the throttle-_manager via the oracle measSpeed. A more primitive form of the cruise control specification might model the actual speed of the vehicle as a parameter that could be directly observed by the throttle_manager. By introducing the speed_transducer agent to the specification, it is possible to relate the speed of the vehicle as recorded to observations that correspond more accurately to the engineering model – for example, to assess the effect of changing the period or register size of

the counter/timer upon the accuracy and range of speed measurement.

Functional dependence is a fundamental aspect of measurement and observation. Basic physical laws, such as Hooke's Law and Newton's Second Law of Motion, express functional relationships between simultaneous observations. In LSD agent specifications, derivates are used to represent such relationships. For example, the derivate for the variable accel in the LSD specification of the vehicle agent expresses how Newton's 2nd law defines the acceleration of the vehicle. In other contexts, a derivate may represent an idealised relationship, as in the definition of the engine power output in the engine agent, where a change in the throttle position is deemed to have a simple instantaneous effect.

In understanding a system in engineering terms, a further distinction must be made between observations that are conditionally under the control of an agent, and those that are beyond its control. In an LSD specification, the variables that an agent can redefine are implicit in its protocol: for example, in Listing 1, the throttle_manager agent alters the status of the throttle to reflect manual or automatic control. Variables conditionally under an agent's control are classified as **handles**.

There are two principal respects in which the LSD modelling paradigm differs from an object-oriented approach. In general – as is appropriate in a realistic engineering model – an agent can directly change the value of a variable owned by another agent. What is more, derivates can express the way in which indivisible change of state is propagated across object boundaries, as typically happens in a mechanical system.

In our modelling approach, the primitive abstraction is the observation rather than the object. As explained in [2], the indivisible updating of sets of dependent values in a spreadsheet-like model is a powerful way of expressing the systems of transformational methods that define objects in an object-oriented framework. In effect, we can regard observations as a low-level abstraction: objects are apprehended and described in terms of observations of them.

The connection between an LSD specification and potential observations of a physical system serves two functions. On the one hand, the specification documents the engineering requirement and directly reflects information about the engineering application. The conventions for specifying variables in LSD enable us to record where transmission of information between agents is crucial to correct operation of the system, even if the transmission mechanism is unspecified – as in the case of the oracle brakePos in Listing 1. It is also a routine matter to adapt the specification to reflect "what if?" experiments, as in simulating brake failure, changing wind resistance, modifying the characteristics of the counter/timer or investigating worst-case scenarios.

On the other hand, the LSD specification can not be interpreted in isolation from the engineering application. The use of variables in the specification is similar to that in a spreadsheet; values are interpreted in connection with external observations that are primary intuitions of a procedural nature. [In contrast, mathematical variables do not admit a procedural interpretation, and a mathematical model of behaviour is unaffected by the substitution of definitions for variables.] The LSD specification is incomplete and ambiguous as a behavioural model; it must be interpreted with reference to conventions not explicit in the model. For example, Listing 1 does not specify how the vehicle speed, as calculated by the speed_transducer, is transmitted to the throttle manager. Both actSpeed and autoThrottle are defined by an integral, but one of these models a continuous physical relationship, the other a relationship that must be maintained by a control circuit that embodies an integrator. In animating the model, these ambiguities are resolved by making appropriate assumptions about the environment and engineering framework in which it operates.

3. ANIMATION AND VISUALISATION

In animating LSD specifications, scripts of definitions resembling the relationships between cells that underlie a spreadsheet are used to describe a state-transition model. The principles behind such animation, as they apply to concurrent systems in which there is a high degree of non-determinism, have been described elsewhere [3]. In animating agent actions, it is in general necessary to refer to their intended interpretation. For instance: agents typically operate at different speeds; communication of data between agents can follow many different patterns; plausible models of action must be devised where there is non-determinism. In the case of the cruise control specification, these problems are minimised by the event-driven nature of the activity of the engineering components.

Our chosen medium for animation is the EDEN interpreter [4]. Two principal features of EDEN are required: scripts of definitions over real variables, and actions – procedures triggered by changes to the values of variables in these scripts. There is a close correspondence between the LSD specification in Listing 1 and the EDEN animation; the derivates determine the definitive script that defines the state of the system and actions are used to implement protocols.

The interpretation of analog variables is a subtle point in animating the LSD specification. An **analog** variable has a value of type "**real** function of time"; in animation, its value is represented by a real number, but the definition of an **analog** variable may involve reference to the history of its values (as e.g. in the definition of actSpeed and autoThrottle). In

the animation, the values of analog variables are necessarily digitised, irrespective of their interpretation. For example, the integration used to define actSpeed and autoThrottle is explicitly defined with reference to a simulated clock. In this context, there is an important distinction between approximation to an exact value, such as physical laws of motion specify for actSpeed, and simulation of a value computed by an engineering device, such as an integrator of a particular sampling rate specifies for autoThrottle.

The use of EDEN for animation has practical advantages for visualisation. The definitive notations SCOUT and DoNaLD are implemented as filters to EDEN; this makes it relatively easy to represent the state of the cruise control system to the user in an appropriate graphical form [4]. Figure 1 depicts the interface to the cruise control system as defined by these tools. The buttons on the display conform to the specification for the cruise controller as specified in Booch [6]. The user interacts with the animation in the role of the driver agent – as specified in Listing 1 at an appropriate level of abstraction. For example, pushing a button corresponds to redefining the status of the engine or the cruise controller.

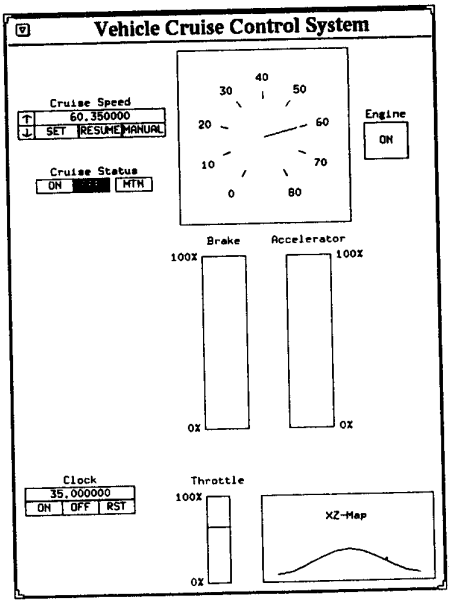

Figure 1

In a recent paper [9], Harel recommends the use of *visual formalisms* for describing conceptual models of reactive systems. Our approach offers the same advantages that Harel ascribes to a visual formalism: it offers a specification that is both executable and amenable to visualisation. A brief comparison between our contrasting approaches is instructive.

Harel distinguishes his conceptual model from a physical model of the system. For him, a major concern in visualisation is representing the conceptual construct underlying a complex reactive system – a problem he has addressed with considerable success by using statecharts. Harel's conceptual model is an abstract mathematical model that is independent of the physical model, but captures its required functional and behavioural characteristics.

Harel's rationale for a formalism is expressed in the following terms [9]: "A prerequisite to being able to execute models of complex systems is the availability of a formal semantics for those models, most notably, for the medium that captures the behavioural view". An LSD specification is not a formalism in Harel's sense. Its behavioural interpretation is ambiguous until implicit assumptions about the physical system to which it refers have been considered. However much the specification is refined, there will always be assumptions about the physical system that have not been made explicit. In this respect, the specification resembles an engineer's justification of why a particular physical system functions correctly. This justification may appeal to physical laws, to familiar properties of standard mechanical components, but rests ultimately upon faith in the reliability of patterns of behaviour that are suggested by experimental observation.

Despite this informality, the methods proposed in this paper appear well-suited to achieving the practical objectives that Harel identifies. The guiding principle is that a reactive system model can be developed in terms of a characteristic set of observations selected by the designer. This establishes a close connection between requirements analysis and testing: the same set of observations that is used to guarantee that the system fulfils requirements can also serve as an effective basis for testing. Selecting appropriate observations to be made of a system is a good basis on which to initiate visualisation; it identifies the parameters that have to be displayed to the designer to convey the system state. It also provides a natural technique for refinement; a particular concern identified by Harel.

CONCLUSION

This paper has described and illustrated a new approach to the specification of reactive systems. The objective of this approach is to enable the engineer to construct a system model that closely reflects the physical characteristics, capabilities and interrelationship of its components. The underlying principle is to express the system behaviour in terms of values of variables that represent potential experimental observations upon which the validity of the design depends. Animation and visualisation of such specifications can be performed using state-transition models that combine actions (resembling rules in rule-based systems) and definitive representations of state (resembling interconnections between cells of a spreadsheet).

ACKNOWLEDGMENTS

We are indebted to Steve Russ for useful discussions and helpful comments on the content of this paper.

REFERENCES

1. Beynon, W. M., "Programming principles for the semantics of semantics of programs," Computer Science RR#205, Warwick Univ., January 1992

2. Beynon, W. M., Russ, S. B., Slade, M. D., Yung, Y. P., "Programming as modelling: new concepts & techniques," Proc ISLIP'90, Queen's Univ. Kingston, 1990

3. Beynon, W. M., Norris, M. T., Orr, R. A., Slade, M. D., "Definitive specification of concurrent systems," Proc UKIT'90, IEE Conf Publications 316, 1990, 52-57

4. Beynon, W. M., Yung, Y. P., "Definitive Interfaces as a Visualisation Mechanism," Proc GI'90, Canadian Inf Proc Soc, 1990, 285-292

5. Birtwistle, G., Dahl, O-J., Myrhaug, B., Nygaard, K., Simula Begin, 2nd ed., Studentliteratur, Lund, Sweden, 1979

6. Booch, G., "Object-Oriented Development," IEEE Trans Software Engineering, SE-12(2), 1986, 211-221

7. Deutsch, M. S., "Focusing Real-time Systems Analysis on User Operations," IEEE Software, Sept 1988, 39-50

8. Deutsch, M. S., "Enhancing Testability with Scenario-Oriented Engineering," 6th Int Conf on Testing Computer Software, (Washington D.C.), 1989, 1-12

9. Harel, D., "Biting the Silver Bullet: Towards a Brighter Future for System Development ," IEEE Computer (to appear Jan 1992)

10. McDermott, D., "A critique of pure reason," Comput Intell, **3**, 1987, 151-160

11. Smith, B. C., "Two lessons of logic," Comput Intell, **3**, 1987, 214-218