AGENT-ORIENTED MODELLING FOR ENGINEERING DESIGN

Meurig Beynon
wmb@uk.ac.warwick
Dept of Computer Science

Alan Cartwright
esraf@uk.ac.warwick
Dept of Engineering

University of Warwick, Coventry CV4 7AL, UK

ABSTRACT

We review an approach to engineering design and simulation that we have developed over several years. Our research exploits a fundamentally new method of programming that is ideally suited to creating computer-based design environments where state, state change and the agents of change can be modelled. The programming method is general-purpose and has significant connections with object-oriented, functional and spreadsheet programming paradigms. We consider its application as a basis for environments for modelling systems of interacting agents, as a mode of knowledge representation and as an implementation technique. Our approach is thus of interest from both a computer science and an engineering perspective.

INTRODUCTION

Models of state and agent-interaction have a significant role to play in many aspects of the design process. In this paper, we first describe general principles for agent-oriented modelling that we have developed, then indicate how they can be applied in constructing a computer-based multi-agent design environment.

Throughout the paper, we refer to a particular case-study – the development of a Vehicle Cruice Control System (VCCS) – that demonstrates the feasibility of our modelling system in principle. For more technical details of the VCCS case-study, see [4,6,8].

Our VCCS model illustrates three different levels of agent-interaction that are significant in the design process:

- in developing the engineering model, we represent knowledge about how the physical components of the cruise controller interact to define the state of the system in operation,
- in considering the VCCS model from a design perspective, we represent the different views and privileges accorded to the various participants in the design process that determine the current status of the design,
- in implementing the design environment, we apply an agent-oriented paradigm in which all programming is expressed as interaction between computational agents.

Whilst we recognise that much further development will be required to meet the realistic needs of practising designers, our method is potentially extremely powerful and versatile. For instance, it allows design and simulation to be performed in parallel (cf [9]), and offers good prospects for implementation on parallel architectures [1,2].

AGENT-ORIENTED MODELS AND PROGRAMS

Agents, Observations and Indivisible Relationships

Our approach is based on certain fundamental abstractions for constructing state-transition models for complex interactive systems [3]. We first analyse possible behaviours of the system in terms of the state-changing components or *agents*.

The agent concept is interpreted in different ways according to which aspect of the design process we address:

- the components of a complex engineering system can themselves be regarded as agents. By analysing the relationship between them, we represent engineering knowledge about how the behaviour of the entire system is related to that of individual components.The components of the VCCS include the *speed transducer* that generates a digital output representing the vehicle speed, the *speedometer* that conveys the measured speed to the driver, and the *driver*, who manipulates the brake, accelerator and the cruise control device.

- the specialist designers who participate in the design process can be viewed as agents. By analysing their interrelationship, we understand how to represent the design to the different participants and to identify where there are potential conflicts between their requirements. Our VCCS model enables us to perform experiments in the role of different designers. For instance, we can change the sampling rate of the speed transducer, redesign the profile of the vehicle, reconfigure the dashboard display and change the functionality of the buttons in the interface to the cruise controller.
- the computational elements required to construct a computer model of the engineering system, and to simulate its behaviour, can be regarded as agents. By representing them, we *both* record the current state of the design *and* prototype the system to be constructed in software in such a way that we can perform realistic experiments on a computational model. Our VCCS model is both a representation of an evolving design and an animation of the requirement. The process of analysing the interaction between the agents in a real vehicle cruise control system is used to construct a computer program to animate this interaction.

The interaction between an agent and its environment is modelled in terms of *observations*. Observations record an agent's view of the state of its environment. They are represented by variables whose values can be changed through actions by agents in the system. The different aspects of the design process involve different kinds of observations:

- the observations that define the engineering model correspond to the measurements that an engineer might make to explain the interaction between the engineering components. For instance, in the VCCS, the behaviour of the speed transducer is understood in terms of the relationship between its sampling rate, the measured speed it outputs and the actual vehicle speed. Of course, the relationship between observations does not necessarily account totally for an agent's behaviour. For instance, the layout of the dashboard display influences a driver's actions, but does not completely determine them.
- the interaction between specialist designers depends upon what each designer sees of the total design and what aspects of the design they are allowed to change. In the VCCS, the designer of the dashboard layout will wish to experiment with the model in the role of a driver, but will not be concerned with the method of integration used in the speed transducer.
- in the computational models we use for simulation, the observations associated with computational agents correspond to variables to which they respond and variables whose values they can conditionally change. Identifying these variables resembles specifying the interface between objects or modules in a conventional software engineering framework.

Our approach is distinguished from other agent or object-oriented approaches to modelling by the emphasis we place upon faithful representation of indivisible relationships between observations. An indivisible relationship between observations is established when changing one observation necessarily entails a complementary change to another. Such links between observations arise in each of the three aspects of the design process. For instance:

- in engineering models, they arise as mechanical linkages, as invariant relations between forces and acceleration, or in situations where it is convenient to deem that changes in observations propagate instantaneously.
- in interactions between designers, indivisible relationships link a modification made by one designer to its consequences for another. For example, a change to the shape of a vehicle will affect its wind resistance.
- in computer simulation, indivisible relationships define the associations between internal and external values, so that the state of the display is defined in terms of the values of internal variables, and is updated automatically as they change.

Dependencies between observations that are identified in this way are represented in our approach by sets of definitions or *definitive scripts*. (Throughout this paper, we use the term "definition" in a technical sense to refer to definitions of variables somewhat similar to those used to specify the values in a spreadsheet, and the term "definitive" to mean "definition-based". The use of sets of definitions to represent state is a common feature of many design systems, but is adopted as a convenient practical technique [10,11], rather than developed as a fundamental programming principle.) In a definitive script, the computation involved in ensuring that the values of variables accord with their definitions is hidden from the user, and – like the updating of cells in a spreadsheet – has no external interpretation. The result is a programming paradigm based on "agent-oriented modelling over definitive representations of state" [3, 4, 5] to be elaborated below.

Perceptions, Privileges and Animation

In specifying an agent-oriented model of a system, we first record and classify the observations associated with each agent using a special-purpose notation called LSD. In this process, we regard each agent in anthromorphic terms as having certain *perceptions* of state and certain *privileges* to change state. The values perceived by the agent are its **oracles**, and those that it can conditionally redefine are its **handles**. Indivisible relationships between observed values, as perceived by the agent, are recorded as **derivates** formally resembling the definition of a variable. The same variable may be referenced in different ways by different agents – for example, as an oracle for one agent and a handle for another. All such references are viewed as associated with an authentic value of the variable that is bound to a particular agent and accordingly defines part of the state of that agent. Such references to values are referred to as **states** of the agent for this reason.

An LSD specification is not directly executable: it merely describes the framework within which the agents can exercise their privileges to change the state of the system. The LSD specification includes a **protocol** for each agent that prescribes the enabling conditions that must be met before an agent can perform a state-changing action. Each such action comprises a sequence of one or more redefinitions of handles and invocations of agent instances.

LSD agent specifications supply the context for many of the state-changing activities associated with the design process. The extent to which they can be transformed into patterns of automatic computation depends upon their significance. For instance, in [4], we give the complete LSD specification for the components of the VCCS that is used to animate the behaviour of the system over time. Direct animation of this kind is possible only because engineering components typically react instantaneously and deterministically to stimuli. In contrast, an LSD specification for the dashboard layout designer might include the privilege to change the angular range of the speedometer display, and derivates to maintain indivisible relationships between the locations of the graphical elements that make up the speedometer. In this context, it would be unreasonable to invoke automatic criteria for this privilege to be exercised.

The Abstract Definitive Machine (ADM) was developed as a medium for animation from an LSD specification [2,3]. The ADM is an appropriate computational model in which to represent a variety of state changes, whether they are to be performed automatically or driven by the designer. An ADM program is defined by a family of entities, each of which comprises a set of definitions and a set of actions formally similar to the privileges in an LSD specification. In execution, a set of entities is instantiated. On each machine cycle, the enabling conditions associated with actions are evaluated in the context of the script defined by amalgamating the sets of definitions associated with currently instantiated entities. Actions whose enabling condition is fulfilled are then executed in parallel. The effect of this execution is in general to redefine several variables and to create and/or destroy instantiated entities.

In meeting the demands of design environments, we envisage extensions to the ADM as presently implemented. These involve introducing new mechanisms for specifying ADM entities:

- to define the presence of an instantiated entity as a boolean function of the ADM state,
- to define the *form* of an entity as a function of the ADM state.

Informally, the motivation for introducing these mechanisms is that a change of state can lead to an entirely new context for observation, or to a new perspective on observations of the same object. Some illustrative examples are given below.

APPLYING AGENT-ORIENTED MODELLING TO DESIGN

Representing Engineering Knowledge

From an engineering perspective, our approach can be seen as building upon traditional principles applied in constructing a large system. An engineer interprets the behaviour of a complex system in the light of knowledge about how each component behaves in context. This knowledge can be either theoretical or empirical in nature, but is ultimately based upon experimental observations. The relevant observations are those required to describe the interaction between each component of the system and its environment. This suggests that the most appropriate way to approach the description of an engineering design is to identify components and their associated experimental observations, and to represent the correlations between these observations that can be confirmed through experiment.

The programming principles we adopt are well-suited to modelling of this nature. Each component of the system is represented by an LSD agent whose specification identifies the experimental observations that would have to be made in order to explain the behaviour of the agent in context. The oracles and handles of an agent correspond respectively to the parameters which the engineer *measures* and *changes* when conducting an experiment to justify the design of a component. Definitive scripts capture the inviolable relations between parameters that are observed in an experiment.

The VCCS demonstrates the power of our modelling principles. The state of the system, as specified by a set of observations sufficient to define the roles of the engineering components in many different aspects, is represented by a definitive script that includes some hundred definitions. The observations specified in this script relate to the views of many different design agents. For instance, an observation may concern the configuration of the driver interface, the physical characteristics of the vehicle, or the mode of animation for the designer. Changes of state to the system are represented by redefinitions of appropriate variables. For instance, a single redefinition can transform the model interactively to change the vehicle mass, to adjust the sampling speed on the speed transducer, to redesign the speedometer, or to reconfigure the dashboard display.

In the VCCS model, many of the observations are associated with features whose existence is independent of system state. For example, each of the engineering agents is a persistent entity. In other simulations we have developed [3,5], the agents are more volatile, and the set of observations changes dynamically as entities are created and destroyed. The extensions to the ADM cited above are conceived for applications in which dynamic transformations of sets of observations are necessary. These transformations need not involve changing *what is being observed*, but can relate to the mode of observation or the dependencies between observations. In analysing the analogue to digital conversion involved in measuring the vehicle speed, we need to simulate sampling of the system at different frequencies – a new mode of observation of the inputs and outputs to the speed transducer. In simulation from a bond-graph model, a change in causality does not alter the set of observations, but changes the dependencies between observations.

Design as an Interactive Process

In the modern engineering design process, there is an ever greater need for interaction between specialists representing many different aspects of the engineering task. LSD specification in conjunction with definitive scripts provides a useful framework within which to represent different perspectives on the design process [9]. We can distinguish the roles of different design agents by identifying how they are privileged to change the state of the model through redefining variables in the script. For instance, one agent may specify the sampling rate of the speed transducer, another the relative positions of the components of the dashboard display.

In many respects, the ADM is well-suited to representing the interactions amongst design agents. For example, where a pair of simultaneous redefinitions would lead to conflict, the ADM can detect cyclic references, and suspend execution. Such conflicts can be resolved either by direct intervention of a super-user, who is privileged to redefine variables arbitrarily at any machine cycle, or by devising auxiliary agents that either constrain redefinition or arbitrate between possible alternatives. It is also easy to implement agents to monitor the design evolution and to flag exceptional conditions as they arise. An entity that defines the state of a screen component in terms of the parameters of the design suffices for this purpose.

We envisage the development of design processes and environments where definitive scripts and protocols for interaction complement or replace preliminary specification documents for communication between design agents. This is a natural extension of the principles used to describe the engineering model itself. In effect, a definitive script together with a protocol is an experimental interface by means of which the design agent can extend or refine the design.

Interaction between agents demands techniques for representing design knowledge that involve complex script manipulation and management. These include automatic methods of re-parametrisation, partial evaluation, replication from templates, customisation of designs through combination of stored fragments and systematic methods of storage and retrieval. The extensions to the ADM described above can assist the convenient transformation of definitive scripts. For instance, it is possible to imagine a geometric representation of the design space in which selection of a point determines the set of ADM entities appropriate for a particular design or design stage.

Implementation Issues

The use of definitive scripts has many advantages in implementation [1]. The fact that scripts express indivisible propagation of state-change through a complex system makes it easy to link components that are developed independently. In effect, definitive scripts play a role analogous to that of levers in a mechanical system, allowing components to be coupled in change. Scripts also make the dependencies between variable values explicit, so that the re-evaluation involved in changing state can be efficiently identified

Animation is an essential ingredient in the design of complex engineering systems that integrate electronic components with sensors, mechanical devices and human operators [10]. The VCCS shows the considerable potential for using our modelling techniques in this role, but also highlights the need for modes of implementation better suited to fast execution. We are currently investigating techniques for compiling scripts with appropriately constrained protocols for redefinition into conventional procedural programs.

Agent-oriented modelling is a general-purpose technique for describing the behaviour of concurrent systems [2]. As such, it can also be developed as a method of parallel programming. Definitive scripts have many qualities that suggest that they are a suitable basis for parallel programming:

- the state defined by a definitive script is independent of the order of the definitions,
- parallel redefinition of variables in a definitive script is a conceptually simple model of concurrent action,
- making the same redefinition twice has the same effect as making it once,
- a definitive script records the dependencies between variable values, and thus eliminates traditional problems concerning data dependencies and side-effects,
- definitive scripts can accommodate undefined values gracefully, and readily represent the effects of incomplete computational processes.

CONCLUSIONS

Using our approach, we hope to develop engineering design environments that effectively combine rich expressive power with efficient implementation on parallel hardware.

REFERENCES

1. W. M. Beynon, A. J. Cartwright *A definitive programming approach to the implementation of CAD software*, Intell. CAD Sys. II: Implementation Issues, Springer-Verlag 1989, 126-145
2. W. M. Beynon *Parallelism in a definitive programming framework* Parallel Computing 89, Advances in Parallel Computing Vol 2, North-Holland 1990, 425-430
3. W. M. Beynon, M. T. Norris, R. A. Orr, M. D. Slade *Definitive specification of concurrent systems* Proc UKIT'90, IEE Conference Publications 316, 1990, 52-57
4. W. M. Beynon, I. Bridge, Y. P. Yung *Agent-oriented Modelling for a Vehicle Cruise Controller* Proc. Eng. Sys. Design & Analysis Conf., ASME PD-Vol. 47-4, 1992, 159-65
5. W. M. Beynon, Y. P. Yung *Agent-oriented Modelling for Discrete-Event Systems* Proc IEE Coll. "Discrete-Event Dynamic Systems", Digest #1992/138 June 1992
6. G. Booch *Object-Oriented Development* IEEE Trans SE-12(2), 1986, pp. 211-221
7. A. J. Cartwright, W. M. Beynon Enhancing Interaction in Computer-Aided Design Proc Int Conference on Manufacturing Automation, HK, August 1992, 643-8
8. M. S. Deutsch *Focusing Real-time Systems Analysis on User Operations* IEEE Software, Sept 1988, pp. 39-50
9. M. Chmilar, B. Wyvill *A Software architecture for integrated modelling and animation* New Advances in Computer Graphics, Proc CGI'89, 257-276
10. D. Harel *Biting the Silver Bullet* IEEE Computer, January 1992
11. R. Popplestone, T. Smithers et al *Engineering Design Support Systems* IKBS Mail Shot 7, 1986