Definitive specification of concurrent systems

W M Beynon[1], M T Norris[2], R A Orr[2], M D Slade[1]

[1]Department of Computer Science, University of Warwick, Coventry CV4 7AL
[2]British Telecom Research Laboratories, Martlesham Heath, Ipswich IP5 7RE

Abstract

A new approach to the modelling and simulation of concurrent systems, characterised by the use of sets of definitions to represent states and transitions, is described. The simulation of system behaviour is derived from a specification of possible agent actions by taking account of relative speeds of response and operation. A distinctive feature of the approach is that the perceptions and capabilities of agents are explicitly modelled.

INTRODUCTION

As distributed computing systems become ever more sophisticated, the complexity of their design continues to increase [11]. In response to this trend, many different methods for specifying and analysing concurrent systems have been proposed [8]. Broadly speaking, two perspectives can be adopted when modelling a concurrent system: the one "event–oriented" (in which the emphasis is upon describing the abstract behaviour of the system as a whole), the other "interaction–oriented" or "activity-oriented" (where the objective is to describe the interactions between component parts of the system) [13]. Both of these approaches are now well-developed [2,17] and can provide valuable support in important aspects of design. Neither, however, provides a complete picture.

In the systematic development of concurrent systems software, there is in general an important need to bring both views to bear. Ideally we would like to reason about the abstract behaviour of a system and at the same time describe precisely how system components relate to each other.

This paper considers how these views might be combined. Our approach is particularly suitable for modelling concurrent systems in which the specific form and disposition of the principal system components, referred to here as *agents*, are known. Its immediate application is to the preliminary phase of the system design process, when the form of the protocols for the participating agents has to be developed

The behaviour of a complex system is typically first conceived by the designer in terms of the interactions between 'agents' [4]. To characterise the overall system behaviour, the designer must specify the privileges of agents in a framework that enables the consequences of actions to be faithfully represented. An agent's primitive actions cannot be defined simply in terms of the state-transitions they induce in the system. By way of illustration, entitling an agent to open a door may be an open-ended privilege to change the state of the system, depending upon the context in which the action is performed. It may be an entitlement to move - in a single atomic action that incorporates opening the door - any piece of furniture that is adjacent to the door. To maintain conceptual control over the design it is essential to avoid the need to specify each context-dependent entitlement separately. This motivates a computational model in which the transition associated with one and the same primitive action can depend upon the frame of reference.

Following [3], a definitive (definition-based) description of system state is adopted (cf [8]). The values of the characteristic variables in the designer's model of the concurrent system are specified implicitly, without cyclic definition, as functions of the values of other variables. The redefinition of a variable in the context of a system of interdependent definitions is an appropriate way to represent atomic actions in a concurrent system whose effect is to change several features of the system state simultaneously [4]. This is the essential principle underlying the spreadsheet. For example, through expressing the profit as a function of the price and number of sales of a product, it is possible to regard "changing the profit by side-effect when changing the price" as one indivisible action.

Within a definitive computational framework, agent privileges are specified in the special-purpose language LSD [3]. A distinctive feature of LSD is that the perceptions and capabilities of agents are explicitly modelled [5]. This is important in at least two respects:

- it is possible to consider the implications of changing the perceptions of agents (e.g. when designing protocols for a blind user of a telephone system), or their capabilities (e.g. when assigning dialling privileges to users);

- it is possible to distinguish between synchronisation based upon an agent's perception (as in "inserting coins when the display is flashing"), and synchronisation based upon implicit assumptions about the speed with which protocols are executed (as in "assuming that all digits are registered no matter how rapidly the user dials").

This paper develops the ideas introduced in [3], where the emphasis is upon illustrating the problems of characterising system behaviour using LSD alone. A constructive approach to the behavioural issues raised in [3] is adopted; a new computational model, supplied by the Abstract Definitive Machine (ADM) [6], is introduced to show how behaviours associated with an LSD specification may be simulated. There are four sections: the background to our method of systems modelling, an example to illustrate how LSD can be applied, a description of the simulation framework and an appraisal of our present progress.

BACKGROUND TO THE METHOD

In an event-oriented approach to modelling, a concurrent system is conceived in terms of its possible behaviours, each of which is defined by a sequence of events that may occur. An appropriate model of the system can be given in a mathematical expression that represents all possible behaviours. Whatever approach to modelling is used, some kind of formal representation of the possible behaviours of the system is essential in order to attain intellectual control over a design. It is otherwise impossible to determine the characteristic properties of the system e.g.when a deadlock can occur.

A comprehensive understanding of the system is needed to derive an adequate behavioural model. When first conceiving a concurrent system, a designer typically develops an informal interaction-oriented model based upon knowledge of how specific agents can act in specific states. Event- and interaction-oriented frameworks are linked via the concept of a comprehensive state-transition model of the system of which the designer initially conceives fragments. It is the behaviour of this state-transition model that has to be analysed and adapted by the system designer through modifying the components of the system, whether by reprogramming an electronic device, prescribing protocols for human users or identifying appropriate assumptions about the system environment. This entails developing an

abstract event-oriented model for the system behaviour that can be understood in its relation to the perceived interaction-oriented requirement. Only in this way is the designer able to relate the behavioural limitations of a partial design to the perceived role of the participating agents, so that an iterative process of validation and refinement of the model becomes feasible [2].

The concept of a comprehensive state-transition system model is elusive. To characterise all possible state-transitions from an interaction-oriented perspective presumes precise knowledge of what agents can act, when and how. This observation is of seminal importance in our design method, in which the first objective is the identification of the participating agents in a concurrent system and the privileges they conditionally have to change the system state.

The state-transition model of a concurrent system is also vast. Many representations for states and transitions have been developed as potential behavioural models [8]. Our method exploits a new computational model in which states are represented by sets of interdependent definitions of variables and transitions by sets of redefinitions to be executed in parallel. Exceedingly complex state-transition models can be succintly described using such *definitive* ("definition-based") representations [4] and they are potentially good candidates for behavioural models of systems. Definitive representations are well-suited to expressing the effects of agent actions as conceived by the system designer. They also have a significant role in enabling information about agent privileges to be effectively interpreted in state-transition terms.

In simulation programs, it is commonplace to introduce sets of characteristic variables whose values are explicitly specified to represent features of the system being modelled. Definitive representations also allow the values of characteristic variables to be specified *implicitly*, without cyclic definition, in terms of other variables and constants. In developing a model, the designer must confirm that the effect of possible scenarios for agent action upon these values is consistent with observed or intended patterns of behaviour. In a definitive framework, agent actions are modelled by *redefining* variables. A redefinition, unlike a conventional assignment, can have instantaneous side-effects upon the values of other variables.

The principles underlying our method will be illustrated with reference to a model of a telephone system to be developed below. Characteristic variables for the telephone system record conditions such as whether the phone is onhook or ringing, what tone is emitted by the earpiece; they also include call-status variables such as *isringing* under the control of the exchange. The boolean variable *onhook* is explicitly defined, ringing is defined as *onhook* **and** *isringing*, and *tone* is implicitly defined in terms of call-status variables. A typical action is represented by a redefinition e.g. replacing the receiver is defining *onhook* to be **false**. This can have an immediate side-effect e.g. changing the value of *ringing*. In one transition, two actions might be performed concurrently, as when a user lifts the receiver to answer a call at the same instant that the caller puts the phone down. In this case, two variables are redefined concurrently.

An agent's privileges are modelled with reference to which variables can be conditionally redefined. The user can pick up the phone and so redefine *onhook*, but has no control - for instance - over what tone the telephone emits when the callee engaged signal is received and so cannot redefine *tone*. The definition of *ringing* reflects the fact that redefining *onhook* has different implications for ringing status according to context. The use of an immutable implicit definition for the variable *tone* expresses the fact that the telephone tone is a characteristic of the device fixed by design that stands in the same relationship to the signal received in every simulation state.

A definitive state-representation has several important characteristics. The effect of redefining a variable may be to change many values in a conceptually indivisible fashion. The implications of the same redefinition depend upon the context supplied by the other definitions. Definitions can be introduced independently in any order. Adding new definitions to a particular definitive representation of state resembles adding a system of levers to a mechanical device, allowing indivisible propagation of state change from one component of the system to another. These characteristics enable definitive state representations to play a crucial technical role: mediating between an interaction-oriented model based upon the identification of agent privileges that is developed incrementally during the design process and a behavioural model that is implicitly specified by the effects of potential agent actions.

AN LSD SPECIFICATION FOR THE TELEPHONE

The function of an LSD specification is to describe the roles that can be played by the agents participating in a concurrent system. These roles are specified with reference to those features of the system state to which the agent can respond and those which it can conditionally change. The LSD specification is the basis for behavioural models of the system that are expressible in definitive state-transition terms, but many aspects of concurrent system behaviour cannot be faithfully captured by considering agents' privileges in isolation.

The interface between each agent and the system is specified using techniques originally applied to modelling the user-computer interface [7]. This interface is determined by the user's knowledge of the current state of the system and by the privileges the user has to change this state. The user's view is typically modelled by a set of definitions of variables in which the variables that are explicitly defined serve as parameters in the defining expressions for implicitly defined variables. In general, the user can change only certain parameters. By inspecting the set of definitions, the user can infer the current status of the interaction and see the effect of changing a parameter. This mode of interaction is most simply illustrated by the spreadsheet: the user may know (for instance) the profit as a function of the price at which the final product is to be sold, the number of sales, and the cost of components and services.

Developing the idea a little further, different classes of variables in the user's view can be distinguished. There are those variables - such as the profit - that are implicitly defined and are subject only to indirect changes of value; these will be called **derivates**. Other variables, such as the cost of components, are subject to change beyond the user's direct control; these will be called **oracles**. Finally, there are variables that are conditionally under user control (such as the price at which the final product is sold); these will be called the **state** variables.

The framework for specification in LSD is derived by regarding the user as an archetypal **agent** i.e. as one participant in a concurrent system, having a particular view of the system, and certain privileges to interact with the system. In LSD, a concurrent system is modelled by a family of agents, each having its characteristic oracle, state and derivate variables. A template for the specification of an LSD agent takes the form:

agent *agent_name* (*parameter_list*)
{
    oracle *list_of_oracle_variables*
    state *list_of_state_variables*
    derivate *list_of_derivates*
    protocol *list_of_guarded_commands*
}

As in the Specification and Description Language SDL [19], such a specification may correspond to one or more active instances within the concurrent system model, and agent instances can be dynamically created and destroyed. In an agent specification, the variables referenced by an agent are not necessarily, nor typically, bound to the agent. Each derivate of an agent has a non-cyclic definition, and each *guarded_command* takes the form of a guard together with an associated sequence of actions, each of which either redefines a state variable, or invokes another agent instance. The guards and definitions are expressed in terms of variables referenced by the agent.

In using LSD to model a concurrent system, the first step is the specification of an appropriate set of abstract agents. For simulation purposes, certain agents must then be instantiated. The principles will be illustrated with reference to the LSD specification of a rudimentary telephone system shown below.

```
agent user(U,S) {
    oracle (int) tone[S], (bool) ringing[S]
    state (bool) onhook[S], (int) dialled[S]
    protocol
    ¬onhook[S] → onhook[S] = true; dialled[S] = @,
    ¬onhook[S] ∧ (tone[S] == D) → dialled[S]=dialled_number,
    ¬onhook[S] ∧ (tone[S] == C) → speak,
    onhook[S] ∧ ¬ringing[S] → onhook[S] = false; dial(S),
    onhook[S] ∧ ringing[S] → onhook[S] = false; speak
}

agent telephone(S) {
    oracle      (bool) #onhook[S] = true,
                (int)  #dialled[S],
                (bool) #dialling[S] = false,
                (bool) #calling[S, dialled[S]] = false,
                (bool) engaged[S, dialled[S]],
                (bool) isringing[S],
                (bool) active[S]
    derivate
    (char) #tone[S] = D if dialling[S]:
                      E if calling[S, dialled[S]] ∧ engaged[S, dialled[S]]:
                      R if calling[S, dialled[S]] ∧ ¬engaged[S, dialled[S]]:
                      C if active[S]:
                      @ otherwise,
    (bool) #ringing[S] = onhook[S] and isringing[S]
}

agent exchange() {
    oracle      (bool) onhook[T], ...
                (bool) calling[S, T], ...
                (bool) #connected[S,T] = false, ...
                (bool) #answered[T] = false, ....
    derivate
    (bool) #isringing[T] = calling[?,T] ∧ onhook[T] ∧ ¬answered[T], ...
    (bool) #engaged[S,T] = calling[S,T] ∧ (ringing[T] ∨ ¬onhook[T]), ...
    (bool) #active[S] = connected[?,S] ∨ connected[S,?], ...
    (time) #T_dial = timeout_for_dialling,
    (time) #T_call = timeout_for_calling
}

agent dial(S)     {
    oracle      (int) dialled[S],
                (time) T_dial, time,
                (bool) onhook[S],
                (bool) calling[S,dialled[S]] = false
    state       (time) #t_start = |time|
    derivate    (bool) dialling[S] = ¬onhook[S] ∧ (time - t_start < T_dial),
                (bool) LIVE = dialling[S] ∧ ¬calling[S, dialled[S]]
    protocol    dialling[S] and valid(dialled[S]) → connect(S, dialled[S])
}

agent connect(S,T) {
    oracle      (bool) onhook[S], onhook[T], ringing[T], engaged[S,T],
                (time) T_call, time
    state       (time) #t_call = |time|,
                (bool) answered[T] = false,
                (bool) connected[S,T] = false
    derivate
    (bool) calling[S,T] = ¬connected[S,T],
    (bool) LIVE = ¬onhook[S] ∧ (
                      (calling[S,T] ∧ (time - t_call < T_call)) ∨
                          ringing[T] ∨
                      (connected[S,T] ∧ (¬answered[T] ∨ ¬onhook[T]))
                  )
    protocol
    ¬engaged[S,T] ∧ ¬connected[S,T] → connected[S,T] = true,
    ¬onhook[T] ∧ ¬answered[S,T] ∧ connected[S,T] → answered[T] = true,
    engaged[S,T] → delete connect(S,T)
}
```

The permanently instantiated agents in the system are the *exchange()*, *user()* and *telephone()* agents. The parameters for the *user()* are used for disambiguation: they specify the identity of the user and the telephone and distinguish references to different instances of the variables with the same name. In simulation, there is one instance of the *exchange()* agent, but there are many users and telephones. The above specification has been used to simulate a two user system, using telephones identified by their numbers S and T. For simplicity, only the variables of interest when simulating a call from S to T are explicitly represented in the specification of the *exchange()*. In the process of simulating such a call, *telephone(S)* invokes a new agent *dial(S)* that in turn invokes the agent *connect(S,T)*.

The oracles for the user are variables bound to the telephone: a boolean to indicate whether the telephone is ringing, and a variable with values to represent the various tones emitted by the earpiece. The user can determine whether or not the telephone is onhook, and can also enter a number into the dial register of the telephone. In the specification, this is viewed as an atomic action.

The oracles for the telephone represent information about its current state: whether or not it is onhook, the content of the dialling register, and other variables that record the status of a call that is in process. A definition of an owned oracle such as

oracle #dialling[S] = **false**

indicates that the value of the variable is from time to time conditionally under the control of a transient agent – in this case dial(S) – and will revert to **false** when this agent is no longer extant (cf the **state-oracle** of [3]).

In an agent specification the special variable LIVE indicates when the agent is extant. The *connect(S,T)* agent is deleted when the condition *engaged[S,T]* is encountered; it is otherwise extant whilst the connection between S and T is being established, whilst T is ringing and whilst the connection has been made and telephone T has not yet been answered or is offhook. The special variable *time* is used as a universal oracle associated with a *clock()* agent whose protocol consists of a single guarded command:

time = |time|+1,

where *|time|* denotes the value of the variable *time* in the current context.

The protocols of agents are readily interpreted in cognitive and informal operational terms, subject to important qualifications to be discussed below. For instance, when the phone is onhook the user can lift the receiver. If the phone is not ringing, this results in an invocation of the *dial()* agent. Whilst the phone is offhook and the timeout for dialling has not lapsed, the status of the variable *dialling[S]* is specified by the derivate in *dial()* and the dialling tone is emitted by the phone. This entitles the user to enter a number into the dialling register. If this number is valid, the *dial()* agent invokes the *connect()* agent, and so on.

Useful as these operational interpretations are, it is wrong to suppose that the LSD specification of agent privileges completely constrains the system behaviour. Any idea that the LSD specification has an authentic behavioural interpretation is an illusion arising from the fact that additional cognitive input is unconsciously introduced by the intelligent reader. In interpreting the user protocol, we do not assume that the user necessarily answers a ringing phone promptly - if at all. In contrast, for appropriate behaviour of the system, the *dial()* and *connect()* agents must be assumed to be relatively fast and predictable in execution and response. The telephone specification is no less consistent with pathological behaviours associated with scenarios such as the user frenetically lifting and replacing the receiver whilst dialling numbers at the speed of light, but these are not typically conceived by the designer (cf [3]). Methods of deriving appropriate simulations are the subject of the next section.

SIMULATING BEHAVIOUR IN THE ADM

The primary role of an LSD specification is to describe the interaction between agents in a concurrent system in terms of their privileges to perform actions. As explained in the introduction, the fact that these are to be interpreted in the context of definitive state–transition models is crucial, since it enables the same action to have different effects upon the state of the system in different contexts. This is the first step towards establishing a comprehensible relationship between the behaviour of the system and the concurrent execution of atomic actions by agents.

In practice, the relationship between an LSD specification and the simulation of system behaviour is nonetheless very complex. Determining agent privileges is not enough to characterise possible agent interactions, for a variety of reasons. In general, an LSD specification can be given many behavioural interpretations, most of which are inappropriate. The relevant issues are discussed at length in [3] and will be briefly reviewed here with reference to a framework within which an LSD specification can be given an appropriate behavioural interpretation.

Every behavioural interpretation of an LSD specification is based upon the behaviour of individual agents, as specified by the agent protocols. Informally, agent behaviour can be conceived in anthropomorphic terms. Each agent acts sequentially, and at any point in a simulation is either "in transition" – in the process of sequentially executing the actions in one of the guarded commands specified in its protocol, or in a "waiting" state pending commitment to the execution of any particular command whose guard is presently true. In guard evaluation, an agent refers to the values of referenced variables "as they are perceived". The choice between guarded commands whose guard is presently true is non-deterministic, and the rate of execution of the associated sequence of actions is unspecified. The motivating idea is that an LSD specification identifies the characteristics of the system behaviour that depend upon the interrelated capabilities and perceptions of its participating agents, but that a precise description of system behaviour requires additional assumptions.

Behavioural interpretations of an LSD specification can also be described in more precise - if more prosaic - computational terms. For this purpose, a state-transition model is used. The current state of a simulation is represented by a set of definitions in which each agent has a private copy of the variables it references, and the derivates supply implicitly defined variables. The private copies of variables are used to record the perceived values of variables; the redefinition of a state variable during the execution of a protocol assigns a new definition to the authentic occurrence of the variable, viz. that associated with the agent to which it is bound. The computational interpretation is rendered complete by introducing appropriate mechanisms to model the communication of the authentic value of a variable to the other agents that reference it.

The definitive state-transition model described above provides a generic framework for simulating the behaviour of a system in which the agents act according to specified protocols under appropriate assumptions about speeds of operation and the nature of communication. Consideration of the telephone specification above readily proves that devising a simulation from an LSD specification requires essential application-specific input from the designer that is based upon cognitive considerations. It is reasonable for instance to suppose that *dialled* serves as a shared variable for the *telephone()* and *dial()* agents, so that the perceived value can be defined to be equal to the authentic value. It would not be so reasonable to suppose that the lifting of the receiver was instantaneously detected by the exchange. An appropriate simulation must take account of unpredictable behaviour on the part of the user, such as picking up or replacing the receiver at any time. The *dial()* agent on the other hand should respond promptly and consistently when the user lifts the receiver.

The computational framework in which the designer must operate in developing an interaction-oriented specification can be conveniently described in terms of an appropriate machine model for context-sensitive parallel redefinition – the *abstract definitive machine* (ADM) [4,6]. In the ADM, the computational state is represented by a set of definitions D that is dynamically modified through redefinition of variables and the creation or deletion of definitions. The transitions to be performed in executing an ADM program are specified by a set of guarded actions A to be executed in parallel as and when the guards allow. Each action is a sequence of instructions that either redefines a variable, or leads to the instantiation or deletion of an entity comprising a set of definitions and actions. The ADM gives output by redefining variables whose values model the state of an output device.

An ADM program consists of a set of abstractly specified entities. Execution is initiated by instantiating appropriate entities. On each machine cycle the guards associated with actions in A are evaluated in the context specified by the definitions in D. If there is no interference, those actions that are associated with true guards are then executed in parallel. Evaluation required in a redefinition – as in interpreting an action such as:-

time = |time|+1

is performed in the same context as guard evaluation. Autonomous computation terminates when no action in A has a true guard.

An LSD specification can be transformed into several ADM programs, each associated with a different scenario for agent action. The parameters used in the transformation process reflect assumptions about the relative speeds at which agents operate and how closely the perceived value of a variable is linked to its authentic value. In practical implementation of a system, these assumptions in turn reflect physical and engineering considerations, such as "How fast can the user dial a number?".

Each LSD agent specification is transformed into an ADM entity description in which the actions are annotated to take account of the speed of an agent's response and execution. These annotations take the form of control parameters for the ADM simulation program to be chosen by the designer. In the present prototype, these parameters serve to assign a probability of selection to each guarded command and to introduce an element of random delay into its execution. The propagation of values from one agent to another is controlled by a similar technique. The result is a faithful computational image of a family of agents executing asynchronously, as informally described in anthropomorphic terms above.

The ADM computational model has several unusual features [6]. It allows the designer to interact fully with a simulation. In execution, an ADM program resembles an environment with an autonomous behaviour in which the designer is in principle privileged to intervene in an arbitrary fashion. Such intervention is feasible because the current state of the execution is recorded by stored definitions and pending actions that can be readily interpreted by the designer. The parameters supplied for the transformation of LSD specification can be

dynamically changed during simulation. It is also possible for the designer to take the part of the user agent, or indeed of any participating agent.

The precise representation of data dependency in the ADM has important implications for the detection of interference [6]. This is particularly significant in a modelling context, where the designer may not wish to prescribe solutions for potential conflicts such as two users fighting over the onhook status of a telephone receiver. Actions can interfere in several ways. The same variable may be redefined independently in concurrent actions, or the set of parallel redefinitions may introduce cyclic dependency. Such interference is detected during computation and the execution is suspended. In one possible mode of execution of the ADM, the designer can act as an auxiliary agent to resolve conflicts as they arise. By exploiting the graceful treatment of undefined values in the ADM, a similar technique can be used to handle user input. The value of a variable is only required when it is encountered in a guard or an evaluated sub-expression in a defining formula. The ADM can either be programmed for default action in these circumstances, or request input from the user.

EVALUATION AND COMPARISON

The approach described in this paper has been successfully applied to obtain simulations of the LSD specification of the telephone. At this stage of development, our methods serve best as an informal design aid, helping the designer to clarify the roles that agents play in influencing system behaviour. It is possible to show the implications of changing the user profile to increase the average length of calls for instance. For our approach to be fully effective, it is essential to move beyond simulation to methods of proving properties of the system behaviour subject to assumptions about the activity of agents. Without such methods, there can be no guarantee that the behaviour of the system meets the requirements of the designer.

Our approach nonetheless has several attractive features. The virtues of definitive representations for specifying transitions in a concurrent programming system have been discussed in detail elsewhere [4]. They include explicit representations of data dependency that can assist the identification of interference and expressive techniques for representing the synchronised propagation of state-changes through a system. As is in part illustrated by the telephone example, these techniques may be used as a means to replace a complex component of a system (such as the telephone exchange) by a simpler model with invisible internal behaviour. This can mean that a conceptually complex system of actions is represented as an atomic action at a higher level of abstraction, so that, for instance: "the telephone starts to ring at the moment that the connection is made". In the design context, this is an important method of enabling the decomposition of the system into component parts.

The use of definitive representations in conjunction with LSD is interesting in cognitive terms. Existing software systems supply empirical evidence that sets of interrelated definitions are well-suited both for representing data relationships in business applications and for modelling the movement of objects [12]. Knowledge that provides the basis for "commonsense reasoning" about the consequences of actions in our environment has to be conveniently represented in terms that take account of the context-dependent nature of action (the frame problem) and admit universal statements about possible states and transitions [9]: the ideas explored in this paper are clearly directly relevant. The use of LSD to represent a user protocol – when the concept of modelling perceptions and capabilities is quite legitimate – makes it possible to interpret cognitive considerations in computational terms, as advocated by Pylyshyn in [14].

Future research will be aimed at extending our present prototype system, essentially based upon an interpreter for the ADM, into a computer-aided system for concurrent systems modelling and simulation. Within such a framework, the designer will be able to specify the privileges for action of individual agents, then construct scenarios in which to simulate concurrent action of the entire system. Whether or not it proves possible to perform formal analyses of behaviour on our models, some significant advantages can be expected. The relationship between the behaviour of the system and the components of the specification can be readily interpreted by the designer. Synchronisation based upon interaction between agents and on timing considerations can be treated as separate concerns. The ADM model potentially supports exceptionally rich interaction in a design environment, effectively giving the designer the privileges to act as an omniscient omnipotent agent within a simulation. The techniques for animation of the simulation within a definitive computational paradigm are already well-developed [7].

Our limited experience with present tools so far confirms expectations. When analysing the output from a simulation, inappropriate behaviour can be linked either to a flaw in the LSD specification, or to an unsatisfactory choice of parameters for its transformation into an ADM program. Both types of defect can be interpreted in cognitive terms. In the telephone specification, for example, we are led to consider issues such as: "is there any purpose in not replacing the receiver when the engaged signal is obtained?" – an issue for clarification in the LSD specification, or "when making a connection does it matter if the fact that the line is engaged is not immediately detected?" – an issue concerning the speed at which values are communicated to and from the exchange.

Because of the cognitive input required when interpreting an LSD specification in operational terms, its conversion to an ADM program can never be fully automated. It is realistic to generate a skeletal model automatically, however, so that the process of introducing assumptions about speeds of execution and communication can be reduced to substitution of parameters. Alternative approaches to prescribing the behavioural interpretation of an LSD specification are mentioned in [3]. The introduction of flags to constrain the interaction between agents, as in Numerical Petri Nets [18] involves no essential operational extension of LSD, but does disrupt the cognitive framework. The introduction of an environment variable **time** can be accommodated within the ADM in a restricted sense (see for example the ADM simulation of a systolic array in [7]), but true real-time concerns raise many difficult issues beyond the scope of our present methods.

Our approach can be contrasted with an event-oriented approach. CSP [10] typifies the abstract behavioural perspective on concurrent systems. The philosophy behind the notation is that, where the behavioural view of a system is concerned ([10] p24): ".... there is no need to make a distinction between events which are initiated by the object and those which are initiated by some agent outside the object. The avoidance of the concept of causality leads to considerable simplification in the theory and its application." Our approach is consistent with this point of view in that the abstract computational model used for simulation has no agent concept. From our perspective, the role of the agent abstraction in system design and modelling is on the other hand essential.

CSP permits the representation of system behaviour as a family of traces, each comprising a sequence of interleaved events. Our behavioural interpretations of an LSD specification are expressed in terms of a non-interleaving concurrency model resembling asynchronous transition systems. Much work on the semantics of the ADM will be required before there can be any prospect of developing concurrent programs from an abstract specification of behaviour in our framework.

It is perhaps more appropriate to compare LSD with the archetypal interaction-oriented approach to simulation, viz that based upon an object-oriented programming paradigm (OOPP) [15]. The basic concepts of the OOPP - that the internal structure of an object is hidden from other objects, and that the only way in which one object can act to change the state of another object is by sending it a message

- are clearly well-adapted for programming in a totally distributed system. It does not seem that an OOPP serves the function of modelling real-world interactions as effectively as LSD however. It is surely appropriate to assert that the telephone user acts directly to take the telephone offhook for instance, rather than sends a message to the telephone telling it to pick up its receiver. Specifying the side-effects of actions through derivates also has some advantages - cf [16], where the limitations of the OOPP as a medium for specifying geometrical relationships between the components of a robot arm are exposed.

It can be argued that most concurrent systems are effectively "totally distributed" at a sufficiently low level of abstraction. That is to say, most systems are built up from digital components that communicate across channels that are physically short but nevertheless introduce problems of synchronisation. An LSD specification for such a system must closely resemble a specification based upon the OOPP: the only way in which one agent can change the state of another is by first changing the value of a variable known as an oracle to that agent, causing it to respond according to its private protocol. The synchronisation issues involved in transforming an LSD specification into an ADM program are similar to those that arise in studying the semantics of the parallel OOPP [1].

It may appear superficially that definitive specification methods are inappropriate when definitive state representations are absent from the final computational model. In fact, our design method seems well-suited to the incremental development of a complex specification by a process of refinement. For this purpose, we first construct a model in which some components and communication in a system are abstractly represented by derivate and state-oracle pairs, as in the specification of the exchange() agent in Figure 1. In the refinement process, these derivates and state-oracle pairs can then be replaced by communication channels and protocols to derive a specification at a lower level of abstraction. Such a use of functional abstraction as a simplifying device is a common informal design technique.

CONCLUSION

This paper has described substantial progress towards understanding the problems of relating the interaction-oriented and event-oriented perspectives on concurrent systems modelling, as they are identified in the context of a definitive approach to programming in [3]. The solutions proposed here are not yet sufficiently well-developed to meet the ultimate objective of relating the abstract behaviour of a system to the roles played by the participating agents as perceived by the designer. They have nevertheless already provided the basis for prototype systems that can assist the designer in the analysis of requirements and the development of a formal specification.

More research into abstract programming within the ADM and practical experience of the proposed design method is required to guide future work on tools. Parallel investigation into the semantics of LSD and the scope for generic methods of deriving ADM behavioural interpretations of an LSD specification is also essential before more practically useful design environments can be developed.

ACKNOWLEDGEMENTS

The authors would like to acknowledge the Director, Communications Systems Technology, Research and Technology, British Telecom for permission to publish this paper. They would also like to thank the many friends and colleagues who contributed to this work.

REFERENCES

[1] P America, *OOP: a theoretician's introduction*, EATCS Bull 29, 1986, 69-84
[2] R Balzer, N Goldman, *Principles of good software specification and their implications for specification languages*, Software Specification Techniques, International CS Series, Addison-Wesley 1985, 25-39
[3] W M Beynon, M Norris, M D Slade, *Definitions for modelling and simulating concurrent systems*, Proc IASTED conference ASM'88, Acta Press 1988, 456-468
[4] W M Beynon, *Parallelism in a definitive programming framework*, Proc Parallel Computing '89, Leiden, Sept 1989, (to appear)
[5] W M Beynon, M Norris, *Comparison of SDL and LSD*, SDL'87: State of the Art and Future Trends, North-Holland 1987, 201-209
[6] W M Beynon, M D Slade, Y W Yung, *Parallel computation in definitive models*, CONPAR'88, British Computer Society Workshop Series CUP 1989, 359-367
[7] W M Beynon, *Evaluating definitive principles for interactive graphics*, New Advances in Computer Graphics, Springer-Verlag 1989, 294-303
[8] A Davis, *A comparison of techniques for the specification of external system behaviour* Comm ACM Vol 31(9), 1988
[9] Ginsberg M L, Smith D E, *Reasoning about Action I & II*, Artificial Intelligence 35, 1988, 165-195 & 311-342
[10] C A R Hoare, *Communicating Sequential Processes*, Prentice-Hall 1985
[11] A A Kaposi, L A Jackson, *A systems approach to complexity management in designing information systems* BT Technology Journal Vol 4 no , 1986
[12] C Lewis, *Using the NoPumpG primitive*, Dept of CS & Cog Sci, Univ of Boulder
[13] R E Nance, *The time and state relationships in simulation modelling*, CACM 24(4), 1981, 173-179
[14] Z W Pylyshyn, *Computation and Cognition*, MIT Press, 1984
[15] M Stefik, D G Bobrow, *OOP: themes and variations*, AI Mag 6(4), 40-62
[16] T Tomiyama, *Object-oriented programming for intelligent CAD systems*, in Intelligent CAD systems 2: Implementation Issues, Springer-Verlag, 1989, 3-16
[17] K Turner, *A constraint-oriented style of specification in LOTOS* Proc FORTE'88, Stirling, Sept 1988
[18] G R Wheeler, *Numerical Petri Nets - A Definition*, Telecom Australia Res Labs Rep #7780
[19] *Functional Specification & Description Lang., SDL* CCITT Standard Z100, 1988