Research report 163

PROTOCOL SPECIFICATION IN CONCURRENT SYSTEMS SOFTWARE DEVELOPMENT

MEURIG BEYNON, MIKE SLADE
SIMON YUNG

(RR163)

A protocol specification technique that relates concurrent system behaviour to agent activity and interation at a high level of abstraction is described. This expoits a concurrent programming technique based upon a synthesis of two new programming paradigms: "agent-oriented" and "definition-based" programming. Application of the method to modelling and simulating activity at a railway station is described. Its potential for concurrent systems software requirements specification is assessed.

Department of Computer Science
University of Warwick
Coventry CV4 7AL
United Kingdom

October 1990

Protocol Specification in Concurrent Systems Software Development

Meurig Beynon, Mike Slade, Simon Yung

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

Abstract

A protocol specification technique that relates concurrent system behaviour to agent activity and interaction at a high level of abstraction is described. This exploits a concurrent programming technique based upon a synthesis of two new programming paradigms: "agent-oriented" and "definition-based" programming. Application of the method to modelling and simulating activity at a railway station is described. Its potential for concurrent systems software requirements specification is assessed.

Keywords

protocol specification, software requirements, concurrent programming, concurrent systems modelling and simulation

1. Introduction

1.1. Background

Protocol specification plays a central role in computer systems engineering. Experience in designing distributed systems has shown the need for systematic methods of specification and analysis to address all aspects of the interaction between concurrently acting processes.

The models used for protocol specification are typically either *state-based* or *sequence-based* [23]. A state-based model is described in terms of the states of the interacting processes, as typically represented using finite state machines, Petri nets or statecharts. A sequence-based model describes the required global behaviour of a system in terms of admissible execution sequences, as typically represented using notations such as RSPL [23], RPL [12] or CSP [13].

Protocol specification techniques have been most successfully applied to specific subproblems of digital systems design, such as describing the interaction between two components communicating over an unreliable channel. With current techniques, it is

sometimes possible to analyse the behaviour of small critical subsystems comprehensively. When designing such subsystems, it is ideally useful to be able to relate state-based and sequence-based views, so that the abstract behaviour of the system can be understood in terms of the activity of its components.

The potential scope for applications of protocol design and specification has been progressively enhanced as distributed processing applications have increased in significance and complexity. At every level of abstraction, it is the protocols for interaction between processes that define the relationship between the global behaviour of a concurrent system and that of its component parts. Conventional techniques for protocol specification and analysis can describe this relationship at low levels of abstraction on well-defined critical subsystems. This paper considers protocol specification and analysis techniques to support the designer at a higher level of abstraction as is required, for instance, in the development of a specification for complex concurrent system software.

1.2. Protocols and the Specification of Concurrent Systems Software

The phases involved in the development of large software systems are traditionally conceived in terms of the *waterfall model* [21]. This paper is concerned with the initial phases of the development life-cycle, viz:

- *Software Requirements Specification*: the activity that leads from an analysis of the software problem to a complete specification of the external behaviour of the system to be built, and
- *Design*: the activity that leads first to the decomposition of the software system into its architectural components and then to the specification of their computational roles.

Where concurrent systems software is concerned, these two phases are respectively intended to produce an abstract characterisation of global system behaviour (such as is typically expressed in a sequence-based protocol specification model) and a complementary explicit description of the activity and interaction of system components

(such as is typically represented in a state-based protocol model).

The objective of the requirements capture and design phases for concurrent system software is a synthesis of state-based and sequence-based views similar to that arising at a different level of abstraction in the detailed design of a specific subsystem. It is natural to seek ways of enhancing traditional protocol specification techniques so that they can be applied to Software Requirements Specification. The techniques for specification of external system behaviour described in [12] include several state-based modelling techniques originally developed for specification at a low-level of abstraction, for instance. The nature of the modelling required in the early phases of software design differs in several significant respects, however:

- *size*: building a state-transition model that captures the behavioural requirement of a complex system is infeasible with techniques developed for small subsystems
- *comprehensibility*: it is essential that the models of software developed in the requirements specification and design phases are intelligible to the designer and correspond closely to the system as conceived at a high level of abstraction
- *modifiability*: there is essential interaction between the requirements specification and design phases in developing specifications for large systems so that the system models can be incomplete and must be easy to enhance or revise.

1.3. Nature and Orientation of our Approach: an Overview

In this paper, the initial phases of concurrent system software development are viewed as a form of protocol specification. The concept of protocol differs in some respects from what is appropriate when considering interaction between components at a low level of abstraction. Following established practice in requirements specification, the behaviour of a concurrent system is related to the actions of agents (see for example [15). Whereas it is common to consider a low level protocol as operating between processes, the distinction between a computational process and the agent performing the computation becomes significant at a higher level of abstraction. Making this distinction will enable us to treat as separate concerns synchronisation based upon

communication between agents and synchronisation determined by the relationship between agents - such as their relative speed of execution.

Our specification method exploits an unconventional form of high-level programming. Informally, it may be regarded as a style of programming with two different manifestations, for which the designations Jekyll and Hyde are not inappropriate. The part of Mr Hyde is played by *agent-oriented* programming (a distant relative of object-oriented programming) and that of Dr Jekyll by definition-based or *definitive* programming (a distant relative of functional programming). Agent-oriented programming involves identifying the agents in a system and describing the interactions between them in terms that are easy to relate to the requirements of the application. The language LSD is introduced to express agent-oriented programs; LSD programs, like object-oriented programs, invite concurrent interpretation but have a quite ambiguous operational semantics. To derive a formal operational model, an agent-oriented program has first to be transfomed into a definitive program. This transformation process invokes additional application-oriented information about the scenario for agent action. The result is a simulation program executable on the Abstract Definitive Machine. The most significant feature of this style of programming is that all control over the changes of state in the simulation model is derived with direct reference to the application.

The paper is in three main sections. §2 examines the motivation for our protocol specification method with reference to object-oriented programming and describes the principles underlying its state-transition model. §3 explains the role of the tools we introduce and sketches their application to modelling and simulating a concurrent system. §4 reviews our approach in relation to the classification of protocol specification methods, current trends in software requirements specification and broader issues concerning modelling, computation and cognition.

The ideas described in this paper have been developed as part of a broad programme of research into definitive (definition-based) programming [5,6,7,8,9,10]. Because of the scope of this paper, the background supplied by this related work could not be described in detail. Though the paper is intended to be self-contained, much

useful supporting material can be found in other references. Of particular relevance are [6,7,8] which respectively consider the specification notation LSD, the abstract definitive machine model, and concurrent programming in a definitive programming framework. Some aspects of the method described in this paper are considered in greater detail in [9]; these are illustrated with reference to a telephone specification.

2. Principles for Protocol Specification at a High Level of Abstraction

2.1. Protocols, Processes and Agents

The traditional concept of *protocol* is well expressed by the following definition, taken from Merlin [18]:

"Given a system of cooperating processes such that cooperation is done through the exchange of messages, a protocol is the set of rules which governs this exchange".

This concept is suited to a particular type of concurrent system: a totally distributed system, in which the only way for one component to influence the behaviour of another is to send a message. In practice, most research on formal specification and verification of protocols has focussed on cooperation of a quite restricted - though fundamental - nature. For instance, cooperation has typically been concerned with reliable transmission of data or the distributed detection of termination - issues so primitive that the significance of the transmitted data or the terminating algorithm is irrelevant.

A significant step towards relating the above concept of protocol to computation at a higher level of abstraction is represented by concurrent object-oriented programming (OOP). In concurrent OOP, the protocols for exchanging messages between objects typically have greater algorithmic significance - they determine whether a distributed algorithm is being correctly executed. Specifying the concurrent behaviour of object-oriented programs raises complex issues beyond the scope of protocol specification methods that have been successfully applied at lower levels of abstraction [1].

Where concurrent systems modelling for requirements specification and design is concerned, we must adopt a concept of protocol that is well-suited to the way in which communication and interaction is informally conceived. Where Merlin refers to

cooperating *processes*, it is commonplace in requirements analysis to think in terms of cooperation between *agents* [15,17]. By way of clarification, a process is here to be understood as "the execution of a sequential program" [2], and an agent as "that which performs this execution". In our development, this distinction between the computational agent and the computational process associated with it - to be further explained and developed below - becomes significant. In particular, whilst it may be possible to ignore the distinction where the role of the agent is narrowly prescribed (as for instance in identifying the *reader* with the *reading process*), it is apparent that protocols do not relate in the same way to agents as to processes at higher levels of abstraction.

In OOP, the role of agents is to an extent played by *active objects* that change state or send messages autonomously. Conventional OOP is unsuitable for concurrent interpretation; it does not make explicit provision for protocols between objects, so that the synchronisation of messages is unspecified. The motivation for our approach is similar to that behind extensions to OOP that have been designed for concurrent execution, such as the actor model of computation [16] .

Our approach resembles an actor model in that the state of a concurrent system is recorded by variables bound to agent instances that are dynamically created and destroyed. In other respects it deviates from an OOP paradigm:

- no concept of information hiding is presumed
- message passing is only one of the ways in which agents interact.

Examples to illustrate modes of interaction outside the scope of the OOP paradigm are given below. The principal motivating ideas are:

- direct action of one agent upon another is sometimes appropriate: cf the distinction between requesting a child not to step into the road and physically restraining it
- there are commonly situations in which a single action on the part of an agent leads in one indivisible transition to simultaneous state changes in several objects.

It is sometimes argued that a totally distributed computational model, in which all communication is by message passing, is the only model directly relevant to digital

system design. It is true that at some level of abstraction all interaction between electronic components can be regarded as a communication of values, but there are other considerations:

- In the development of a specification, it may be helpful to presume that an agent has direct control over values bound to another (cf the manipulation of aircraft positions by an air traffic controller, as discussed in [15]).
- Embedded systems require good models of their environment if they are to operate effectively and this environment may incorporate elements (for example mechanical components) whose interaction is not most effectively modelled using message passing.
- The capabilities of computer-based systems increasingly resemble those of real-world agents that can directly perceive changes of state in their environment (for example through sensory input) and act directly to change the state of objects in their environment (for example by performing physical actions).

These considerations justify our interest in applying our protocol specification methods to modelling and simulating general concurrent systems.

2.2. Object-oriented, Agent-oriented and Definitive Programming

The above discussion motivates a variant of object-oriented programming that can appropriately be called *agent-oriented* programming. In object-oriented programming, the representation of computational state is simplified through the identification of generic pieces of local state. The problem in concurrent interpretation is then to represent synchronised changes that occur in independent pieces of local state in a conceptually indivisible action. In agent-oriented programming, the emphasis is upon faithful modelling of indivisible transitions irrespective of what pieces of local state they affect. The implementation of this style of programming requires new techniques for representing states and transitions.

Our approach to protocol specification at a high-level of abstraction will be introduced and illustrated informally with reference to modelling and simulating the

activity surrounding the departure of a train from a station. The state of the model will be represented by variables bound to a set of agent instances. Agents are of two kinds:

- *active* agents that can perform automonous actions to change the system state
- *passive* agents that serve to record state information but cannot perform actions.

The driver, the guard, the station master and the passengers are active agents: they are able to change the system state, for example by releasing a brake, or opening a door. The doors of the carriages are passive agents: the state of the system is in part determined by whether doors are open or closed. (The term "object" would perhaps be more appropriate than "passive agent", but for its technical meaning in OOP.)

The distinction between agents and OOP objects reflects a different approach to modelling state-transitions. In an object-oriented system, each object is responsible for changing its state: thus a door object "opens itself" by performing an internal action or *method*. To simulate a passenger opening a door, a passenger object sends a message to a door object that invokes the appropriate method. Such an action may occur whilst the station-master is monitoring the train to confirm that all doors are closed. In that case, the fact that a door has been opened will have to be relayed to the station-master, who must register that there is now a door open. This must be implemented by programming the door object to send a message to the station-master object to invoke the appropriate change of state.

It is possible to interpret the activity in the OOP model in terms of actual system behaviour. The message passing paradigm is particularly appropriate if the carriage doors are semi-automatic and respond to passengers pressing a button. In that case, there is a point in time when the passenger has invoked opening of the door but it is not yet open. In much the same way, though a door has been opened, there is a point in time before this fact is registered by the station-master, who then imagines that all doors are still closed. With such interpretation, OOP potentially leads to very subtle models of system state, but it is impossible to reason about concurrent behaviour in the system without additional information about how sending, receiving and responding to messages is synchronised.

In the agent-oriented approach, priority is given to modelling the desired global state-transitions in a system as faithfully as possible without regard for information hiding. It may not be realistic to suppose that the station-master instantaneously registers the fact that a door is opened, but for many practical purposes it is convenient to assume this. It may be both realistic and convenient to suppose that a passenger can directly act to change the state of a door. An agent differs from an object in this respect: the actions of one agent can directly effect changes of state in another. Where an agent action results in a conceptually indivisible propagation of state changes involving several agents, as typically occurs in a mechanical system, this is to be modelled as a single transition.

Providing an appropriate computational framework for the representation of indivisible state-changes in a concurrent system requires an additional concept. In our approach, this is supplied by exploiting the essential mechanism that underlies the spreadsheet, in which changing a single value simultaneously changes all dependent values *as if in one indivisible transition.* The station-master's knowledge of whether all doors are closed is expressed by a *definition*:

$$\mathbf{all\_doors\_closed} = \neg\mathbf{open\_door}_1 \wedge \neg\mathbf{open\_door}_2 \wedge \ldots \wedge \neg\mathbf{open\_door}_N$$

that is to be interpreted as asserting that the value of the variable **all_doors_closed** is defined by the formula on the RHS. (The definition concept resembles the combinational function of the protocol specification language SAN [23], though a different kind of useage is envisaged, as will become clear below.)

In the computational model used to represent system state, the variables associated with agents may have their values defined by formulae in this fashion. In effect, the entire system state is represented by a set of definitions of variables. A typical agent action is represented by the redefinition of a variable. Opening the first door becomes

$$\mathbf{open\_door}_1 = \mathbf{true}$$

for instance. A significant feature of the computational model is that this redefinition may or may not change the value of the variable **all_doors_closed** in one and the same indivisible transition, according to whether another door is already open. This is a

powerful means of modelling the fact that the indivisible transition from one system state to another associated with a particular action is context dependent.

The ideas outlined above derive from an extended programme of research into a "definitive" (definition-based) programming paradigm [5,6,7,8,9,10]. In this style of programming, computation is represented using *definitive state-transition* (DST) models, in which

- computational state is represented by a set of definitions
- transition from one state to another is represented by one or more redefinitions.

DST models have great expressive power. Redefining a variable in the context of a set of definitions can resemble supplying a different parameter to a function defined by a functional programming script, or perhaps even modifying the script [24]. The discussion of protocols below is to be interpreted with reference to a computational framework in which a concurrent system is modelled in the following way:

- the entire system state is represented by a set of definitions of variables
- an agent action is represented by the redefinition of a variable
- transition from one system state to another is modelled by parallel redefinition associated with concurrent action by one or more agents.

As a simple illustration, a scenario in which two doors are opened concurrently is represented by the performing the pair of definitions:

$$\mathbf{open\_door_1 = true},\ \mathbf{open\_door_2 = true}$$

in parallel. These ideas will be further developed in connection with simulation techniques to be described below.

3. Protocol Specification using LSD and the ADM

3.1. Introducing LSD and the ADM

The behaviour of a concurrent system is to be related to the protocols adopted by the agents and the context in which these are executed. In concurrent OOP it is not sufficient to specify how objects act but also how their actions are synchronised through message processing. Agent-oriented programming introduces more powerful

abstractions to represent communication and synchronised interaction between agents. It is still typically necessary to complement agent descriptions with information about relationships between agents, but the hypothesis that must be made regarding interprocess synchronisation are weaker and more easily comprehended in application-oriented terms.

In justifying a high-level protocol, there are two separable concerns:

- the abstract characteristics of the agents, and how they act in isolation

  By way of illustration, the station-master must be able to detect whether a door is open and be able to close an open door. When the station-master sees that all the doors are closed and the train is due to depart, he will blow a whistle. The role of the station-master can be described in such terms with reference to conditions he can perceive and actions he can perform - independently of the context for action.

- what assumptions must be made about the context for interaction.

  The consequences of executing the protocol cannot be predicted without additional assumptions. Issues such as the reliability of the station-master's perceptions and the appropriateness of his actions depend upon the speed of his responses relative to other agents, the nature of the communication between agents and the scenario governing the selection of agent actions. A typical departure protocol will prove unsatisfactory if a passenger decides to disembark and board the train repeatedly, for instance. Conflicts may also arise, as when the station-master attempts to close a door as it is being opened by a passenger, or two passengers attempt to pass through a single door simultaneously.

The abstract characteristics of an agent will be specified using a special-purpose notation called LSD. An LSD specification for an agent describes

- the aspects of the system state to which it can respond – its *oracle* variables
- those aspects it can conditionally change – its *state* variables
- the circumstances under which state-changing actions can be performed – its *protocol*, consisting of a set of guarded possible sequences of redefinitions.

It also includes definitions – represented by *derivate* variables (such as the definition

$\mathbf{all\_doors\_closed} = \neg\mathbf{open\_door}_1 \wedge \neg\mathbf{open\_door}_2 \wedge \ldots \wedge \neg\mathbf{open\_door}_N$

discussed above) that can express the different ways in which agent actions are to be interpreted in state-transition terms according to context.

An LSD specification correlates possible actions of an agent with perceived states of the system and provides the basis for simulations of the behaviour that exploit DST computational models. It constrains the behaviour of the system model so that

- all state-transitions are associated with action on the part of one or more agents
- every agent action is consistent with its perceived knowledge of the system state.

The limitations of such a specification as an operational model are readily apparent [6,9]: it does not specify how accurately an agent perceives values, when it chooses to act, or its speed of execution relative to other agents. (The aspects of agent behaviour described by an LSD specification in some respects resemble those described using *deontic logic* in [17].)

To provide an operational interpretation of an LSD specification requires simulation under suitable hypotheses about the mode of execution. The identification of these hypotheses is an auxiliary part of the specification process. It is not important for the station-master to register every time a door is opened whilst a train is at the station, but it is important that he correctly perceives that all doors are closed as the train departs. Guaranteeing correct perception presumes facts about the context for interaction that may appear self-evident but are beyond the scope of the LSD specification. For instance, the fact that the station-master has continuous visual access to the state of the doors is significant and must be reflected in the manner in which their perceived state is maintained in the model.

The simulations of system behaviour associated with an LSD specification exploit a newly developed computational model: the Abstract Definitive Machine (ADM) [7,8]. In the ADM, computational state is represented by a set of definitions that changes dynamically as redefinitions are performed. The state information required to determine when a redefinition is to be performed is itself encoded within the set of definitions. An ADM program is specified by a set of entities. An entity comprises a set of definitions

and a set of actions. A typical action is a guarded sequence of redefinitions. In execution, the definitions and actions associated with currently instantiated entities determine the contents of definition and action stores D and A. In each machine cycle, actions in A whose enabling conditions are true in the state specified by the definitions in D are performed in parallel.

3.2. Illustrative examples

In this section we illustrate how LSD and the ADM can be used

- to formulate a possible protocol for the departure of a train from a station
- to simulate passengers boarding and leaving a train.

The specifications are not entirely independent but their inter-relationship is easily understood by inspecting the common variables.

The departure protocol takes the following form:

as the train approaches the station
    the guard applies the brake to stop the engine
after the train has waited at the station for an appropriate interval of time
    the station-master checks and shuts the doors
when all doors are closed
    the station-master whistles to call the attention of driver and guard
the guard waits for the station-master to raise his flag
    to signal the release of the brake
the driver gives the ready signal to the station-master who raises his flag
after the brake is released
    the guard signals the station-master by raising a flag
the station-master signals the driver to start the engine.

LSD specifications for the Station-Master (SM), guard, driver and train agents appear in Figure 1. A "#" symbol before an identifier designates a variable that is owned by the agent. The status of the SM's flag sm_flag is bound to the SM. The SM specification is annotated to show the significance of the states and oracles. The oracles for an agent typically represent perceived values of variables whose "authentic value" is bound to another agent. The status of a door for instance is not necessarily what the SM imagines it to be. A state variable that is subject to change by another agent is

designated as a *state-oracle*. A state variable is normally treated as an oracle; when it is necessary to distinguish between the perceived value and the authentic value of a variable that is under the control of an agent (as when the SM cannot remember whether he has whistled) it may appear both as an oracle and a state.

The LSD specification can be given an informal operational interpretation by imagining that each agent acts asynchronously, repeatedly checking the enabling conditions for actions and then non-deterministically selecting an enabled action to be performed sequentially. Since the real departure protocol is designed to operate in just such a fashion, it is easy to simulate by animating the specification in Figure 1. The LSD specification effectively describes the way in which communication is used to transfer control between the agents.

The LSD specifications required to simulate passengers boarding and leaving the train appear in Figure 2. The oracles to the passenger include knowledge of travel status: a passenger responds to arrival at the destination by following an alighting protocol. The derivate LIVE designates that passenger agents cease to be passengers when they have left their destination station. A derivate such as alighting has a significant role in specifying the activity of the passenger agent, it represents a boolean condition that is monitored continuously and of which the passenger has instant and accurate knowledge. This aspect of the specification is relevant for instance in building a robot to emulate a passenger.

For simplicity, each passenger agent has been associated with a particular door. The door is specified so that passengers pass through the door one at a time: the protocol of the door agent non-deterministically selects a passenger from the set of passengers queuing to pass through it. It may be observed that the use of definitions enables the position of passenger p, represented by the variable pos[p], to be interpreted in many ways. The position of a passenger may signify to the guard that there is a queue at a particular door, for instance. For purposes of the simulation, such a process of interpretation may be deemed to require no time. This use of definitions also serves to synchronise a passenger's passage through the door with another's entitlement to

occupy the door.

It is noteworthy that the specifications in Figure 1 and 2, though related, are easy to develop separately and link together. The role of the station-master can be defined first with reference to whether doors are open, without regard for the agents responsible for opening and closing them. The additional requirement that the departure protocol should take account of whether there are passengers queuing can be added subsequently. Such considerations are significant for the incremental development of a specification. A further stage of refinement might involve reconsideration of the choice of around[d] as an oracle for the station-master, who is not in a position to detect passengers about to disembark.

A formal operational interpretation of LSD specifications is given by simulation using the ADM. Full details of how such a simulation is described are beyond the scope of this paper; only the essential ideas will be outlined (cf [9]). In the transformation of an LSD specification into an ADM program, each agent is modelled by an entity that is specified using the set of oracles, states and derivates to which the agent refers:

- derivates are transformed into definitions
- the agent protocol is transformed into a set of mutually exclusive guarded actions to be performed with delays between commitment to action and successive redefinitions that reflect the semantics of the associated agent actions.

Communication between agents is modelled using additional entities so that *either* an oracle is defined to have its authentic value *or* is updated by appropriate actions. Devising the ADM program corresponds to specifying a scenario within which to perform simulation taking account of assumptions concerning the relative performance of agents and the nature of the communication between them.

Simulation in the ADM can make use of probabilistic methods to generate random delays, but it also possible to reflect the truly non-deterministic nature of the agent specifications by allowing the designer to select the sequence of parallel actions to be performed in simulation [9]. The close correspondence between state-transitions in an ADM simulation program and state-transitions in the system being modelled makes it

easy for the designer to interpret and intervene. The ADM can detect interference between actions associated with multiple redefinition of variables, such as occurs when two passengers open a door simultaneously. More interesting forms of interference, such as occur when two agent actions require conflicting contexts of definitions for their articulation, can also be detected. Such conflicts can be resolved by elaborating the model or dealt with by the designer as they arise during simulation [9].

The specification in Figure 2 illustrates how the requirement that passengers pass through a door one at a time can be enforced. Another kind of conflict arises when a passenger alighting from the train delays shutting the door until such a time as the next passenger alights. Only by assuming unrealistically prompt closing of the door after alighting and slowing up the activity preparatory to alighting is it possible to avoid this problem with the current specification. This conflict can be resolved by elaborating the specification in one of two ways:

- enhancing the sensitivities of passenger agents so that they never rush through doors nor close doors without first checking there is no-one in the way. (This involves a strong but unquantifiable commitment to prompt closing of doors when it is perceived to be safe: in practice, it is impossible to guarantee that this response is sufficiently fast to eliminate conflict without specifying a scenario.)
- enhancing the model to reflect the physical impossibility of a door passing through a person in transit. Conflicts of this kind have been considered in connection with interference in mechanical systems in [10].

As the above example suggests, the role of LSD in requirements specification is more closely linked to the identification and interpretation of conflict than to its resolution. In simulation using the ADM, the problems of recovery encountered in many conventional simulation systems are eliminated since the computational state is relatively stable and transparent to the program designer. Our discussion of the examples above is intended to illustrate the different ways in which the use of LSD and the ADM can assist in clarifying why the execution of a protocol may lead to conflict. Imaginary scenarios in which real agents and objects have bizarre characteristics play an

important role in this process of clarification. Analysing the specifications in Figure 1 and 2 leads us to consider doors that can pass through passengers, passengers who can move at the speed of light or station-masters who can see through carriages. Our approach offers some formal support for this common informal process of analysis by proof and counter-example.

4. Concurrent Systems Software Requirements Specification as Protocol Specification

In this paper, software requirements specification for concurrent systems software has been viewed as a generalised form of protocol specification. This generalisation depends upon adopting a perspective upon protocols in which synchronisation due to communication between agents and that due to characteristics of the system of agents as a whole are treated as separate concerns.

In [23], Venkatraman and Piatkowski classify protocol specification methods. Under this classification, the approach considered above has:

- a state-based orientation
- a multi-component structure
- a specification philosophy that can be either meta-implementation or monitor.

As a state-based technique, it supports concurrent execution. Component connectivity is integral to component specification in so far as state-oracle pairs are identified in the LSD specification, but the nature of communication between components is determined with reference to ADM simulations. Combinational functions and discrete state-machine components are built-in in LSD, but more sophisticated component types, such as programmable clocks, have to be programmed for the ADM. Other issues raised in [23], such as the comprehensibility and operational nature of the specifications will be discussed in detail below.

Software requirements specification aims at rigorous description of the intended external behaviour of a concurrent system. For relatively small conventional systems, an approach such as that discussed by Danthine in [11] is effective. The system designer identifies the list of acceptable message sequences through the simulation of a

basic scenario and error recovery scenarios. This behaviour is then directly described using a formal protocol model that becomes the basis for the detailed design and implementation. The list of acceptable message sequences is typically described using a sequence-based model that is realised by a state-based model reflecting the architectural components of the system.

Direct characterisation of the behaviour of a large concurrent system in sequence-based terms is unrealistic. The development of such a specification demands a detailed knowledge of the required behaviour of the system that can only be acquired through an iterative process of preliminary design, simulation, analysis and revision. The size of a system is not the only consideration in this respect; for embedded systems, where there is interaction with external objects, devices or people, the required behaviour is developed in conjunction with assumptions about the manner in which free agents in the system are liable or entitled to act.

The techniques described in this paper are well-matched to a philosophy for software development in which software requirements specification and design are seen as inseparably linked [15]. Characterising the external system of behaviour of the system proceeds in parallel with decomposition of the software system into its architectural components and the specification of their computational roles. This requires the development of a state-based (ie operational) model for the system that:

- can be interpreted by the designer in terms of agents acting in the application
- can be used for simulating and analysing system behaviour.

The more strictly a state-based model fulfils these conditions, the better. Ideally all aspects of the state-transition behaviour of the model should be interpretable in application-oriented terms. The state-based models that are amenable to analysis in restricted contexts do not have the expressive power to represent large systems in ways intelligible to the designer; those, such as concurrent OOP, that are most easily interpreted in application-oriented terms present formidable problems in simulation and analysis.

The state-based approach advocated in this paper has promising characteristics. An

agent-oriented model is well-suited to the way in which a designer initially conceives the requirements specification for a concurrent system (cf [12] and [15]). The DST models upon which the ADM relies can represent exceedingly complex state-transition systems. They also have characteristics that are useful in simulation [9] and may be amenable to analysis. The relationship between an LSD specification and an associated ADM simulation is based upon perceived characteristics of scenarios for execution. This establishes a correspondence between the behaviour of the system model in simulation and the parameters that the designer must either choose or identify in the application, such as what agents can do and when, how fast they act, and how fast and reliably messages are communicated.

The modelling techniques discussed in this paper relate to several interesting philosophical issues:

- The use of LSD and the ADM is concerned with distinguishing that part of system behaviour that depends upon the actions of agents in isolation from that dependent upon their inter-relationship in context. This relates to sequence-based and state-based views in protocol specification [23], event-oriented and activity-oriented views in simulation [19] and holistic and reductionist theories [14].
- LSD uses agents as an essential means of developing a specification of system behaviour: all state changes are attributable to the actions of agents. The relevance of causality has been questioned in connection with event-oriented modelling using CSP [13] and with relativistic mechanics [22].
- The usefulness of LSD and the ADM in requirements specification is related to the validity of a cognitive thesis: that the consequences of agent actions can be effectively represented using DST models. Other issues, such as modelling the perceptions and capabilities of agents, also establish connections with research on cognition and computation [20].

5. Conclusion

The protocol specification techniques introduced in this paper address issues that lie

at the focus of many strands of current software research. We believe that our approach represents a promising new direction of development of particular relevance to the search for better general-purpose concurrent programming techniques [3].

Experience with our present prototypes proves that our approach can serve a useful function in the preliminary stages of system design, prompting the designer to probe the qualities and limitations of protocols through specification and simulation. Of particular importance is the way in which the validity of a protocol can be related to the perceptions and capabilities of the participating agents and the characteristics of the environment in which they operate. To exploit these qualities fully, our present prototypes must be developed into more powerful and versatile software tools. This requires a better environment for the development and execution of ADM programs, to include techniques for semi-automatic generation of programs from an LSD specification.

Future work must address the formal representation and analysis of concurrent system behaviour and consider the issues involved in the implementation of system components.

Acknowledgements

We are indebted to numerous colleagues for encouragement, guidance and criticism. Particular thanks are due to Mark Norris of British Telecom Research Laboratories for helping to promote this work and contributing motivating ideas. We also wish to acknowledge the role that the financial support of British Telecom has played in assisting this research - this is not to imply any endorsement of the views expressed, for which the principal author takes full responsibility.

References

[1] America P et al *Operational Semantics of a Parallel Object-Oriented Language*, CS–R8515, CWI Amsterdam 1985

[2] Andrews G R, Schneider F B *Concepts and Notations for Concurrent Programming*, Computing Surveys, Vol 15 (1), 1983, 3-43

[3] Baldwin D *Why we can't program multiprocessors the way we're trying to do it now*, Tech Rep 224, CS Dept, Univ of Rochester 1987

[4] Balzer R, Goldman N *Principles of good software specification and their implications for specification languages*, Software Specification Techniques, International CS Series, Addison-Wesley 1985, 25-39

[5] Beynon W M, Norris M T *Comparison of SDL and LSD*, Proc SDL'87, ed R Saracco, P A J Tilanus, North-Holland 1987, 201-209

[6] Beynon W M, Norris M T, Slade M D *Definitions for modelling and simulating concurrent systems*, Applied Simulation and Modelling, Proc IASTED ASM'88, Acta Press 1988, 94-98

[7] Beynon W M, Slade M D, Yung YW *Parallel computation in definitive models*, CONPAR 88, BCS Workshop Series CUP 1989, 359-366

[8] Beynon W M, *Parallelism in a definitive programming framework*, Advances in Parallel Computing Vol 2, North-Holland 1990, 425-430

[9] 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

[10] Beynon W M *Evaluating definitive principles for interactive graphics*, New Advances in Computer Graphics, Springer-Verlag 1989, 291-303

[11] Danthine A A S *Protocol Representation with Finite-State Models*, IEEE Trans on Comms, COM-28 (4), 1980, 632-643

[12] Davis A M *A comparison of techniques for the specification of external system behaviour*, CACM(31) 1988, 1098-1115

[13] Hoare,C A R *Communicating Sequential Processes*, Prentice-Hall 1985

[14] Hofstadter, D R *Godel, Escher, Bach*, Vintage Books Edition 1980

[15] Johnson W L *Deriving Specifications from Requirements*, Proc 10th Int Conf on Software Engineering, Singapore, 428-438, 1988

[16] Lieberman H *Thinking About Lots Of Things At Once Without Getting Confused*, MIT AI Laboratory Memo #626, 1981

[17] Maibaum T S E et al *A Logic for the Formal Requirements Specification of Real-Time Embedded Systems*, Alvey Project SE 015 Report R3, 1987

[18] Merlin P M *Specification and Validation of Protocols*, IEEE Transactions on Communications vol Com-27, 1979, 1671-1680

[19] Nance R E *The time and state relationships in simulation modelling*, CACM 24(4), 1981, 173-179

[20] Pylyshyn Z W *Computation and Cognition: Toward a Foundation for Cognitive Science*, MIT Press 1984

[21] Royce W *Managing the development of large software systems: Concepts and Techniques*, Proc 9th Int Conf on Software Engineering, IEEE 1987, 328-338

[22] Russell B *ABC of Relativity*, 3rd Edition, Allen and Unwin 1969

[23] Venkatraman R C, Piatkowski T F *Formal Protocol Specification Techniques*, Proc 5th Int Workshop on Protocol Specification, Verification and Testing, IFIP 1986

[24] Bird R, Wadler P *Introduction to Functional Programming*, Prentice-Hall 1989

```
agent sm() {                                    // The station master:
oracle      (time) Limit, time;                 // knows the time to elapse before departure due
            (bool) guard_raised_flag;           // knows whether the guard has raised his flag
            (bool) driver_ready;                // knows the driver is ready
            (bool) around[d]; (d = 1 .. number_of_doors)
                                                // knows whether there's anybody around doorway
state       (time) #tarrive = |time|;           // registers time of arrival
            (bool) #can_move;                   // determines whether the driver can start the engine
            (bool) #whistle = false;            // controls the whistle
            (bool) #whistled = false;           // remembers whether he has blown the whistle
            (bool) #sm_flag = false;            // controls the flag
            (bool) #sm_raised_flag = false;     // remembers whether he has raised the flag
state-oracle
            (bool) door_open[d]; (d = 1 .. number_of_doors)   // partially controls the doors
derive      (bool) #ready = ∧ (¬door_open[d]) | d = 1 .. number_of_doors);
                                                // monitors whether all doors are shut
            (bool) #timeout = (time – tarrive) > Limit;       // monitors whether departure is due
protocol    door_open[d] ^ ¬around[d] –> door_open[d] = false; (d = 1 .. number_of_doors)
            ready ^ timeout ^ ¬whistled –> whistle = true; whistled = true; guard(); whistle = false;
            ready ^ whistled ^ ¬sm_raised_flag –> sm_flag = true; sm_raised_flag = true;
            sm_flag ^ guard_raised_flag –> sm_flag = false;
            ready ^ guard_raised_flag ^ driver_ready ^ engaged ^ ¬can_move –> can_move = true;
}

agent guard() {
oracle      (bool) whistled, sm_raised_flag;
state       (bool) #guard_raised_flag = false; #guard_flag = false;
state-oracle
            (bool) brake;
derive      LIVE = engaging || whistled;
protocol    engaging ^ ¬brake –> brake = true;
            sm_raised_flag ^ brake –> brake = false; guard_flag = true; guard_raised_flag = true;
            guard_flag ^ ¬sm_flag –> guard_flag = false;
}

agent driver() {
oracle      (bool) can_move, engaged, whistled;
            (position) at;
state       (position) to;
            (bool) running; #driver_ready = false;
state-oracle
            (position) from;
protocol    engaged ^ whistled ^ ¬driver_ready –> driver_ready = true;
            engaged ^ from <> at –> from = | at |; to = | next_station_after(at) |;
            engaged ^ can_move –> driver_ready = false; running = true;
}

agent train() {
state       (bool) #running = true; #brake = false;
            (bool) #door_open[d] = false; (d = 1 .. number_of_doors)
            (position) #from = station1; #to = station2;
            (position) #at = some_position;
derive      (bool) #engaging = running ^ to == at; #leaving = running ^ from == at;
            (bool) #engaged = ¬running;
protocol    engaging ^ ¬alarm –> alarm = true; guard(); sm();
            leaving ^ alarm –> alarm = false; delete guard(), sm();
            brake ^ running –> running = false;
}
```

Figure 1

```
agent passenger((int) p, (int) d, (position) _from, (position) _to) {
// passenger p is intending to travel from station _from to station _to
// and he will access through door d of the train
oracle      (position) at;
            (bool) queueing[d];
state       (position) #from[p] = _from;
            (position) #to[p] = _to;
            (int) #door[p] = d;
            (bool) #pos[p] = 1;
state-oracle
            (bool) door_open[d];
derivate    #alighting[p] = at == to[p] ^ -2 ≤ pos[p] ≤ 0 ^ engaged;
            #boarding[p] = at == from[p] ^ 0 ≤ pos[p] ≤ 2 ^ engaged;
            #join_queue[p,d] = door_open[d] ^
                                (alighting[p] ^ pos[p] == -1) || (boarding[p] ^ pos[p] == 1);
            LIVE = ¬(at == to[p] ^ pos[p] == 2);
protocol    at == to[p] ^ pos[p] == -2 -> pos[p] = -1;
            alighting[p] ^ ¬door_open[d] -> door_open[d] = true;
            alighting[p] ^ pos[p] == 0 ^ door_open[d] ^ queuing[d] -> pos[p] = 1; pos[p] = 2;
            alighting[p] ^ pos[p] == 0 ^ door_open[d] ^ ¬queuing[d]
                                -> pos[p] = 1; door_open[d] = false; pos[p] = 2;
            boarding[p] ^ ¬door_open[d] -> door_open[d] = true;
            boarding[p] ^ pos[p] == 0 ^ door_open[d] ^ queuing[d] -> pos[p] = -1; pos[p] = -2;
            boarding[p] ^ pos[p] == 0 ^ door_open[d] ^ ¬queuing[d]
                                -> pos[p] = -1; door_open[d] = false; pos[p] = -2;
}

agent door((int) d) {
oracle      (int) pos[p], door[p]; (p = 1 .. number_of_passengers)
derivate    #queuing[d] = there exists p such that join_queue[p,d] == true;
            #occupied[d] = there exists p such that (pos[p] == 0 ^ door[p] == d)
            #around[d] = there exists p such that (door[p] == d ^ -1 ≤ pos[p] ≤ 1)
protocol    queuing[d] ^ ¬occupied[d] ^ join_queue[p,d] -> pos[p] = 0; (p = 1 .. number_of_passengers)
}
```

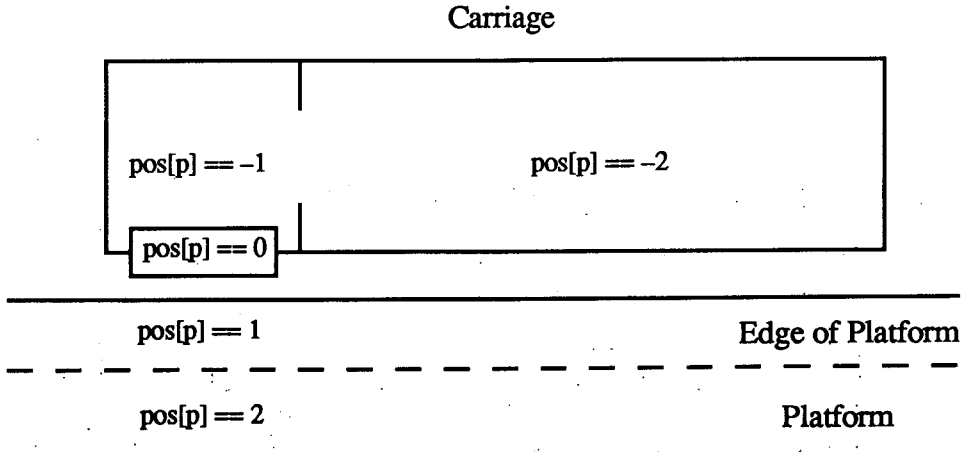

Figure 2