DEFINITIONS FOR MODELLING AND SIMULATING CONCURRENT SYSTEMS[1]

Meurig Beynon, Mark Norris[†], Mike Slade

(RR124)

A novel activity-oriented approach to the modelling and simulation of concurrent systems is described. A method of reasoning about the perceptions and capabilities of an agent is illustrated. A brief evaluation, and a comparison with other approaches is included.

[1] presented at the IASTED conference: Applied Simulation and Modelling '88.

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

[†] British Telecom Research Laboratories
Martlesham Heath
Ipswich
IP5 7RE, UK

May 1988

Introduction

The ever increasing move towards distributed computing systems has greatly increased the complexity of their design. Many different methods for modelling and simulating such systems have been proposed. There is a significant distinction between a modelling perspective based upon an "event-oriented" world-view, in which the emphasis is upon describing the abstract behaviour of a system, and an "activity-oriented" world view, where the objective is to describe the actions of the objects comprising the model and the conditions for these actions to be performed [1]. In the systematic development of sound concurrent systems software, there is in general an important need to adopt both viewpoints: to abstractly reason about the behaviour of a system and at the same time formally describe the rôles of the participating agents. This can be achieved *either* by developing an activity-oriented specification in such a way that we can formally describe the system behaviour, *or* by developing an event-oriented specification in such a way that we can infer an explicit representation for the concurrently acting agents.

We consider an activity-oriented approach based upon a new language for the description of concurrent systems of interacting agents - the LSD notation ([2,3]). LSD is founded upon a novel "definitive (definition-based) programming" paradigm, and has several unusual characteristics. A distinctive feature is that the perceptions and capabilities of agents are explicitly modelled. This is important in at least two respects:
1) it becomes possible in principle to consider the implications of altering the perceptions of agents (e.g. when designing protocols for a blind user of a telephone system), or their capabilities (e.g. when assigning privileges to the users of a computer system).
2) 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 dialled digits are appropriately registered no matter how rapidly the user dials a number").
LSD also includes features for modelling the implications of actions that are - or are deemed to be - instantaneous.

The paper is in three principal sections: an introduction to LSD, an example of an LSD model of a very simple system, and a discussion of the behavioural issues. In the concluding section, the potential rôle of LSD as a specification notation, and its relationship to other methods of modelling and simulating concurrent systems is briefly examined.

§1 An introduction to LSD

In this section we introduce LSD, and give some of the motivation behind its development (c.f. [2,3]). The term **process** has previously been used in LSD - as in SDL - to refer to an autonomous agent that acts sequentially according to a particular protocol, but such terminology is inconsistent with [1]. The nearest equivalent to the LSD or SDL process is the **object** of [1] (c.f. [4,5]), but the LSD process is significantly different from the object of an object-oriented paradigm, and we prefer to adopt the term **agent**.

The concept of an LSD agent has its origins in a simple paradigm for user-computer interaction [3]. In modelling the interaction between the user and the computer, we shall be concerned with representing the user's knowledge of the current state of the system, and their privileges to change this state. As has been argued in detail elsewhere [3], a particularly effective way to represent this knowledge is via an acyclic system of functional relationships between variables. In such a system, the variables that are explicitly defined serve as parameters in the defining expressions for implicitly defined variables. By inspecting the system of variable definitions, the user can infer the current status of the interaction, and precisely predict the effect of changing a parameter. Such a 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.

The central concepts of LSD are derived by viewing 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. To explain this more fully, the different characteristics of the variables in the user's view must 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 the **derivates**. Within an agent's view, a derivate is always consistent with its definition. The other variables known to an agent have no such definition, and can be classified as follows. There are those variables such as the *cost* of components, that are subject to change beyond the control of the agent; these will be called the **oracles**. There are also variables, such as the *price* at which the final product is sold, that are conditionally under the agent's control; these will be called the **state** variables. The same variable may serve as both state and oracle.

The semantics of oracle, state and derivate variables is subtle, and we shall be content with informal interpretations. To say that a variable x is bound to the agent a, acts as a state variable for the agent b, and is an oracle for the agent c, corresponds to saying that "x denotes an entity that is extant while the agent a is instantiated", that "the agent b is conditionally able to alter x in some way", and that "the activity of the agent c depends in some way upon the entity x that is subject to change beyond its control". For example, if x is the colour of the house-next-door (agent a), then my neighbour (agent b) may repaint the house under appropriate circumstances, and I (agent c) will have an idea of what the colour x is at any time. The variable x is bound to a in as much as x has no meaning if the house is burnt to the ground. For my neighbour to repaint the house, some precondition such as "having a paint brush" must be met. The variable x is an oracle to me in that it is subject to change beyond my control. It should be noted that a shared variable interpretation of state/oracle pairs is not necessarily appropriate. It may be that my neighbour's house is repainted whilst I am away on holiday, so that I act in accordance with an inappropriate assumption about its present colour. To emphasise the orthogonality of these concepts, observe that an owned or state variable will not necessarily be an oracle: my neighbour may be colour blind, whilst the relevant activity in which the house participates is most probably independent of its colour. Naïvely, we might formulate the guiding principle that "the activity of an agent is determined by its perception" (c.f. viewpoint modelling in CORE [6]).

In LSD, a concurrent system is modelled by a family of agents, each having its characteristic oracle, state and derivate variables. Identifying and classifying the variables of an agent in this way is the first step towards specifying the relationship between an agent action and the associated system response. 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
}
```

- where each derivate of an agent is defined (without cyclic definition) by an algebraic expression in terms of its derivate, oracle and state variables, and each guarded_command takes the form of a guard (i.e. a boolean condition expressed in terms of derivate, oracle and state variables), together with an associated list of actions, each of which either redefines a state variable, or invokes another agent instance.

As in SDL, 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. All variables are bound to specific instances of agents - perhaps to agents that are permanently instantiated throughout the simulation. There is an important distinction between owned variables - those bound to the agent itself - and those bound to another agent; this is reflected syntactically in the specification of an agent by prefixing each owned variable identifier by a "#". Since a state variable for an agent may be owned by another agent, one agent can act directly to change the values of variables owned by another agent. In this respect, our approach differs radically from the object-oriented programming paradigm.

In using LSD for modelling and simulation of a concurrent system, we first specify an appropriate set of abstract agents, then instantiate specific instances of agents. An agent's behaviour is determined by its protocol: each agent acts sequentially, and each step of the simulation comprises appropriate parallel redefinitions of variables and invocations of agents. At any point in the simulation, each agent instance 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 guarded command whose guard is presently true. Note that the choice between guarded commands with true guards is non-deterministic, and the rate of execution of the associated sequence of actions is unspecified.

There are many sources of ambiguity and potential conflict in the model of behaviour as described, but this is not an oversight. The motivating idea is that an LSD specification identifies those aspects of the system behaviour that depend upon the interrelated capabilities and perceptions of its participating agents, but that a precise analysis of system behaviour in general involves additional considerations - c.f. §3.

§2 An illustrative example: a remote control system

The characteristic features of an LSD model will be illustrated by an elementary example. Consider a mechanical toy that moves in discrete steps along a track of length 2L+1 in response to input signals from a remote control unit. For simplicity, assume that the toy is initially placed at the centre of the track, and moves left or right one step at a time. As an initial specification of the LSD agents, consider:

```
agent toy()
{
state           (int) #position=0;
state-oracle    (int) #switch=0;
derivate        (bool) #on_track = abs(position)≤L;
protocol
                on_track and switch = 1  -> position=|position|+1; switch = 0;
                on_track and switch = -1 -> position=|position|-1; switch = 0;
}

agent remote_control()
{
oracle     (int) #button;
state      (int) #output = 0;
protocol
           output = 0 and button =1 -> output = 1;
           output = 0 and button =-1 -> output = -1;
           output != 0 and button = 0 -> signal(output); output=0;
}

agent signal(direction)
{
state       switch;
protocol
            true -> switch = direction; LIVE = false;
}

agent operator()
{
oracle      position;
state       button;
protocol
            position<L -> button=1; button=0;
            position>-L -> button=-1; button=0;
}
```

To set up the model, three agents are instantiated: the toy(), the remote_control(), and the operator(). These are all present throughout the simulation, but the remote_control() also invokes instances of the signal() agent from time to time. The specification of these agents illustrates many of the features of the LSD notation. Note in particular: the use of "| |" for expression evaluation; the invocation of signal() agents in the protocol for the remote_control() agent; the special boolean variable "LIVE" in the signal() agent, whose value determines whether or not the agent instance is live.

The specifications of the participating agents above will be interpreted with reference to the underlying principles outlined in §1. We shall also identify some of the significant behavioural issues to be considered in more detail in §3.

The toy() process has a variable to record its current *position*, and a register *switch* that is set to initiate movement. The condition for the toy to be on the track is defined by a functional relationship, viz "the absolute value of *position* is at most L". Notice how this use of a derivate illustrates the modelling of instantaneous side-effects: the value of the boolean variable *on_track* is defined in terms of its position. Whilst the toy is on the track, it responds to a non-zero value in the *switch* register by moving in an appropriate direction and resetting the *switch*. Notice that all these variables are bound to the toy() agent: were the toy to be removed, these variables would cease to have any meaning. Only the toy itself can act to affect its position, but the *switch* register is also set by an external agent, viz the signal() agent, and is accordingly specified as a **state-oracle**. An important behavioural issue is the possible interference between the signal() and the toy() agents through concurrent assignment to this register.

The rôle of the signal() agent is to set the *switch* register in the toy() agent. Each such agent is instantiated with a parameter to indicate the direction of movement of the toy that is to be invoked, and in principle there may be several signal() agents extant at any one time. The presence of several signal() agents would necessarily complicate the problems of potential interference over the setting of the *switch* register. Notice that enforcing synchronisation upon a signal() agent is impossible according to the specification above, since no provision is made for the signal() agent to reference the value of an external variable.

The remote_control() agent has the most complex specification. There are two variables bound to it: a *button* that is under the operator's control, and a register *output* that serves as a local memory. The remote control responds to button pushes, and has a two stage protocol, whereby it first records which button has been pressed in the register *output*, then proceeds to send an appropriate signal to the toy when the button is released. Notice that, with the behavioural interpretation described above, the remote control does not *necessarily* respond at all to a button push.

The operator() agent is aware of the current position of the toy, and can press a button on the remote control to initiate movement of the toy to the left or right. Pushing and releasing the right button is modelled by the assignments "button=1; button=0". Note that nothing in the operator()'s protocol prevents arbitrarily rapid sequences of button pushes: the pre-condition for pushing a button is not directly affected by performing the action. Indeed, this pre-condition would not in practice be enforced upon an operator - acting according to the specified protocol merely corresponds to "acting with discretion so as to ensure that the toy at all times remains on the track". A protocol that more realistically modelled the operator's capability would remove the pre-conditions upon button pushing entirely, thereby admitting the possibility of the toy being driven off the track.

There are several ways in which the operator's protocol can be elaborated to model different perceptions. In one model - appropriate for a young child, for instance - the operator would only perceive whether or not the toy() is on the track at any time, and be permitted to restore the toy() to the track. For a blind operator, it might be appropriate to model the current position of the toy() as remembered rather than perceived, introducing owned variables for this purpose. The details are left to the interested reader; our simple example illustrates the basic concepts required for this purpose.

§3 Behavioural aspects of the remote control system

As explained above, the primary rôle of an LSD model is to describe the independent asynchronous behaviour of concurrently acting sequential agents, each responding to its perceived view of the system. As our example illustrates, the relationship between the authentic value of a variable and its value as viewed as an oracle by another **agent** is open to a variety of interpretations in general, and need not be simply that of a "shared variable". This is accommodated in our model by regarding each agent as actively evaluating its guards and executing guarded commands - even where such activity is fictional. The behavioural analysis of a system then generally requires additional assumptions about the "speed" at which each agent detects a true guard and processes a command whose guard is perceived to be true, e.g. to ensure that parallel redefinitions of variables are non-interfering. In effect, an LSD specification makes explicit the preconditions (in terms of perceived values) for an agent to perform an action. This is sufficient to constrain the sequencing of actions, but does not encompass real time concerns.

By way of illustration, it may be important to support non-deterministic behaviour on the part of a user, yet to demand deterministic responses from other agents participating in a system. For instance, a telephone user is unlikely to put down the receiver whilst awaiting a connection, even though replacing the receiver when it is offhook is always a valid option in the user's protocol (c.f. [3]). It is equally inappropriate to assume that the telephone responds non-deterministically and intermittently to a user picking up the receiver. The LSD model also makes it possible to distinguish between synchronisation through perception (as in "pouring fresh tea into a guest's teacup when it is empty"), and synchronisation through independent assumptions e.g. about speed of propagation of values or execution of actions (as in "assuming - when replacing a broken light-bulb - that turning off the light switch will first render the light socket safe").

The limitations of the LSD model from a behavioural perspective are quite apparent in our illustrative example. Taking a pessimistic procedural view, it may be assumed that the participating agents sample their guards non-deterministically, and from time to time react to the identification of a true guard by embarking on the associated sequence of actions at a finite but irregular rate of execution. To assume more would be inconsistent with the behaviour exhibited for instance by the operator(), who can be expected to assess the position of the toy and respond after an indeterminate delay by depressing one or other button for an unspecified amount of time. In our example, such assumptions serve to model several different communications protocols, each associated with a variable that acts both as a state or derivate for one process and as an oracle for another. As our example illustrates, they are also essential; the singular patterns of behaviour cannot otherwise be eliminated without radically altering the perceptions of the participating agents.

The "intended" behaviour of the remote control system might be described thus: the user notes the position of the toy and presses an appropriate button, the remote control registers the button that is pressed, and on its release sends a signal to the toy that sets an appropriate switch, causing the toy to move and to reset the switch. This LSD specification highlights problems to be resolved in realising the intended behaviour.

The first problem is that pressing a button need not be registered by the remote control at all. An omniscient malevolent operator could in principle ensure that redefinitions of the button variable are precisely synchronised with attempts by the remote control agent to evaluate the guards in its protocol. It is also clear that erratic execution of the guarded commands within the remote_control() protocol can lead to responses that reflect the sequence of buttons pushed by the operator in a quite unpredictable fashion i.e. so that only a small proportion of the button pushes generate a signal. In practice, the design of a remote control unit will be such that the speed at which the operator can manipulate the button is slow compared with the rate at which the control samples the button oracle and acts to generate a signal. It may also be contrived that the duration of a physical button push is only partially reflected by the button oracle - e.g. is normalised so that the value of the button oracle remains non-zero for a fixed time interval. What is significant is that any more sophisticated solution would require a more elaborate interface e.g. a buffer to record a sequence of button pushes, or a "ready light" bound to the remote control - defined as a derivate by the functional relationship "output=0 and button=0" - that served as an oracle for the operator.

Suppose then that the remote control is in some way equipped to deal with every operator input, and is guaranteed to respond by sending an appropriate signal. The interpretation of the behaviour of signal() agents is such that a signal - once instantiated by the remote control - may take an arbitrarily long time to affect the switch setting. This is not altogether unreasonable: the direction in which a signal is sent may be very critical in determining when - and even whether - it arrives. A real environment could be contrived in which signals were alternately sent directly across a room or routed via a satellite, for instance, when the order of arrival of signals at the toy would be uncertain. An additional problem is that of potential interference between the signal() and the toy() processes over setting and resetting the switch register. The toy() agent is free to move according to its protocol at an unspecified speed, and to reset the switch register at any time after moving, whilst the signal() agent is totally oblivious to the status of the toy() agent. There can accordingly be no guarantee that resetting the switch by the toy() agent escapes conflict with an incoming signal.

In practice, there are several possible approaches to these problems. The issues of reliability and sequencing of signals are ordinarily resolved by making reasonable assumptions about the operating environment for the system. It is to be expected that all signals travel at the same speed for instance, and that they set the switch register in the order in which they are sent. In using a remote control for a television, the lack of a response to a button push within a short period of time is normally assumed to indicate misdirection of the signal. Indeed, it might be more appropriate to elaborate the protocol of the signal() agent to indicate that the orientation of the signal by the operator is significant in determining whether the switch setting takes place.

Potential conflicts on switch setting can be handled in many ways. One simple but relatively unsatisfactory solution is to admit the possibility of conflict. When operating a remote control for a television, it would not be unreasonable to expect a non-deterministic response to two very rapid channel switches, on the grounds that changing TV channels is normally done in response to what is observed on one channel. This assumes of course that attempting to simultaneously assign 0 and 1 to the switch register does not lead to some irrevocable error condition. Were that to be a problem, it would also be possible to separate the state-oracle switch within the toy() agent into state and oracle components; this would prevent the signal() gaining direct access to the switch register, and allow only for sequential access by the toy() agent. As with the resolution of conflicts over the setting of the button variable and the output register in the control unit, other possibilities are to ensure that the toy() executes its protocol sufficiently rapidly to guarantee that the switch is always reset prior to the next signal, or to introduce a buffer between the signal() and toy() agents. A similar device to the "ready light" described above for the remote control might also be attached to the toy to indicate to the operator() agent that a further signal can be sent. This type of feedback also helps when concerned with the issue of signal failure.

§4 The rôle of LSD as a specification notation

It may fairly be said that LSD provides an **agent-oriented** view of interactive and concurrent systems. At least for the present, our approach is most relevant to the task of modelling concurrent systems in which the specific form and disposition of the principal agents is 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. A formal specification of the system behaviour requirements within the LSD framework (c.f. JSD [7]), would require more comprehensive techniques.

Two approaches might be considered for this purpose. A natural way to eliminate singular behaviour from the system in §2 is to introduce a boolean state variable *has_moved* that is owned by the toy() agent, and acts as a state-oracle for the operator. The operator can press the button only if *has_moved* is true, assigns *has_moved* to false after pressing the button, and waits for the toy to reset the variable after it has completed a move. Notice that this method of specifying intended behaviour is outside the scope of the engineering design problem; it is a way of formulating a constraint upon the operator. Elaboration of this technique of using flags to constrain the interaction between agents leads to a framework somewhat resembling Numerical Petri Nets [8], but departs from the semantics of **state** and **oracle** variables as informally illustrated above. (A formal protocol designed to prevent an incoming signal from interfering with the resetting of the switch register by the toy() agent could hardly make use of a realistic oracle to the signal() agent. A second approach to synchronisation involves the introduction of an environment variable time used as an oracle by the participating agents. This is more consistent with the semantics of state and oracle variables, but cannot obviate the need to make assumptions about the time required for execution of actions by participating agents.

The difficulty encountered in formulating a theoretical framework for simulation of concurrent systems [1,9] is surely linked with the problem of relating the activity-oriented and the event-oriented perspectives. The key problem is to devise a method for modelling concurrent systems that can be used both to reason about the activity of agents and the behaviour of the system as a whole. In this section, we shall briefly contrast LSD with other approaches: the use of CSP [10,11], and the use of an object-oriented programming paradigm (OOPP) [12,13].

CSP 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." We do not dissent from the view that consideration of the agents responsible for events unnecessarily complicates the abstract specification of the behaviour of a concurrent system, but our experience with LSD suggests that giving more careful consideration to the rôle of agents has very significant implications in an activity-oriented approach, and may be helpful when reasoning about system behaviour.

CSP permits the representation of system behaviour as a family of traces, each comprising a sequence of interleaved events. It is not generally easy for the designer of a concurrent system to relate such a behavioural specification to given constraints about the configuration of processors, or the nature of the participating agents. Nor is it easy within such a framework to correlate the system behaviour with local modifications to the perceptions and capabilities of these agents.

It is perhaps more appropriate to compare LSD with the archetypal activity-oriented approach to simulation, viz that based upon an OOPP [12]. 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 that he/she sends a message to the telephone telling it to pick up its receiver. The concept of specifying the side-effects of actions through derivates also has some advantages - c.f. [14], where the limitations of the OOPP as a medium for abstractly specifying geometrical relationships between the components of a robot arm are exposed.

It is nonetheless plausible to argue that most concurrent systems are abstractly "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 may be physically short but may yet introduce problems of synchronisation. At this stage, it is not clear to what extent the use of LSD can assist the specification of such systems. At first sight, it would appear that any advantages of LSD are forfeit if we eliminate the possibility of direct action of one agent upon another, or of "synchronised side-effects". What is certain is that an LSD specification for such a system would closely resemble a specification based upon the OOPP: the only way in which one agent could change the status of another would be by first changing the value of a variable known as an oracle to that agent, causing it to respond according to its private protocol. At the same time, we should ideally need to improve upon the OOPP, since its semantics is obscure where concurrency is concerned [15]. Our present expectation is that a method for the development of a system by successive refinement may prove successful for this purpose. The prospects for such a method are difficult to assess at this stage, but probably depend on being able to apply an anthropomorphic approach: first constructing a model in which communication is represented via artificial oracle / state / derivate variables, then systematically eliminating derivates by substituting appropriate protocols and refining state/oracle pairs by introducing buffers etc to derive a specification at a lower level of abstraction.

§5 Conclusion

We have illustrated how an LSD specification can capture the logical framework of permissions governing the actions of agents in a concurrent system. Such a specification technique allows a separation of concerns: isolating aspects of the system behaviour that depend upon assumptions about execution times and environmental factors from logical constraints on agent actions. Our present research is directed at developing the techniques needed to reason formally about the behaviour of an LSD model. We anticipate that our agent-oriented approach has particular relevance for issues such as the development of fault tolerant systems.

Acknowledgements

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

References

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

[2] W M Beynon, M Norris, *Comparison of SDL and LSD*, SDL'87: State of the Art and Future Trends, North-Holland 1987, 201-209

[3] W M Beynon, *The LSD notation for communicating systems*, Computer Science Research Report number 87, Warwick University, 1986

[4] D Belsnes, B Møller-Peterson, *Rationale and Tutorial on OSDL: an object-oriented extension of SDL*, SDL'87: State of the Art and Future Trends, North-Holland 1987, 413-426

[5] R Braek, G Hasnes, *New Pathways for SDL*, SDL'87: State of the Art and Future Trends, North-Holland 1987, 401-412

[6] G P Mullery, *Controlled Requirements Expression* LNCS 1985 ISBN 3-540-15216-4

[7] M A Jackson, *System Development*, Prentice-Hall International, 1983

[8] G R Wheeler, *Numerical Petri Nets - A Definition*, Telecom Australia Res Labs Rep #7780

[9] B P Ziegler, *Theory of modelling and simulation*, Wiley 1976

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

[11] W H Kaubisch, C A R Hoare, *Discrete event simulation based on communicating sequential processes*, Computer Science Research Report, The Queen's University, Belfast 1978

[12] K Nygaard, O J Dahl, *The development of the SIMULA languages*, History of Programming Languages, ed Wexelblat, Academic Press, NY 1981

[13] M Stefik, D G Bobrow, *OOP: themes and variations*, AI Magazine 6(4), 40-62

[14] T Tomiyama, *Object-oriented Programming for Intelligent CAD Systems*, in Intelligent CAD Systems 2: Implementation Issues, Springer Verlag (to appear 1988)

[15] P America, *OOP: a theoretician's introduction* EATCS Bulletin 29, 1986, 69-84