

Programming as modelling: new concepts and techniques

Meurig Beynon, Steve Russ, Yun Pui Yung

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

Abstract

The relationship between programming and modelling is considered. Fundamental programming concepts are examined from this perspective. Programming for embedded systems is approached via a method of concurrent system modelling in which the global behaviour of system can be related to the activity of its components.

1. Introduction

The demands made by modern computing applications lead us to re-examine the fundamental nature of programming. Intensional programming has drawn attention to the importance of developing a conceptual framework for programming that is rich enough to meet the challenges presented by new technology and uses for computers. The design of LUCID, for instance, reflects a broad vision about the nature of the programming task aimed at reconciling the perspectives of the priests, the wizards, the cowboys, the boffins, the handymen and the mystics [18].

It is widely acknowledged that the conceptual framework that guided the development of programming languages in the past is inadequate for modern applications. The evidence for this is particularly strong in applications such as embedded systems that involve concurrent activity and a high degree of interaction and communication. This is endorsed by analyses of programming methodology [3] and programming language design [1,2,18].

In this paper, we examine the problems posed by programming embedded systems in the light of an extensive programme of research into programming methods based upon representations of state by sets of definitions of variables resembling scripts ("definitive programming") [4,5,6,7,8,9]. We are led to adopt a perspective in which programming embedded systems is regarded as essentially equivalent to developing concurrent systems models in which the global system behaviour can be related to the activity of its components. In this approach, the concepts of "state" and "agent" play a fundamental role in determining the semantics of programs.

A motivating example

In building interactive systems for graphics, we have developed a line-drawing package that can be viewed as a kind of geometrical spreadsheet [8]. The role of the cells of the spreadsheet is played by variables whose values represent geometric entities such as points and lines, rather than scalars. These values can be specified explicitly or implicitly by formulae expressed in terms of the values of other variables in a non-cyclic fashion.

The representation of a geometrical object in our system is a set of definitions superficially resembling a functional programming script [10]. The essential difference in our perspective is that the values of variables in the script are to be interpreted as defining a *state* of the object. That is to say, certain changes to the script, such as involve the redefinition of particular parameters for instance, are to be considered as part of the geometrical model we have described. This is consistent with the "what if?" semantics of spreadsheet use, in which different possible states of a system are investigated.

A simple example of a script is the following description of a conventional door, as it might appear on an architectural plan:

```
- real    width = doorwidth
bool      open
line      door = [hinge, lock] // the line segment joining points hinge and lock
point     hinge = Lframe
point     lock = hinge + if open then (0, -width) else (width,0)
```

This description is to be understood with reference to a context in which the position of the left frame of the door and the width of the door in the enclosing wall is also prescribed. A simple context might be provided by the following additional definitions:

```
point     NW = (0,0)
point     NE = (100,0)
point     Lframe = NW + (20,0)
point     Rframe = Lframe + (doorwidth,0)
line      n1 = [NW, Lframe]
line      n2 = [Rframe, NE]
real      doorwidth = 20
```

The drawing that these scripts together prescribe is so trivial that it can be drawn in two or three instructions using a basic graphics package. What then is the significance of our scripts in relation to the task of architectural design?

The scripts have to be interpreted with reference to the possible transformations that can be performed upon the state through the redefinition of variables. In using the door, there are essentially two states of interest: the door is either open or closed. When interpreted in conjunction with the protocol

open -> open = false; not open -> open = true

the door script identifies the possible transformations that the user can perform upon the state of the door in a very precise manner. That is to say the script prescribes a transformation of the values of the variables that represent the door that corresponds exactly to what the door user can perceive to distinguish the door open from the door closed.

The script that represents the context for the door has a different status for the door user. In conventional use of a door, we do not change its width or the position of the hinge. This is a valid option for the architect, however, whose privileges to transform the context surrounding the door are ultimately circumscribed only by whether the door fulfils its intended function effectively.

The nature of programming

The purpose of our illustration is to draw attention to some fundamental issues about the nature of programming. The scripts above are most certainly not orthodox programs for drawing pictures. They differ very significantly from a procedural recipe for drawing the plan of a door in a wall, where the picture is built up through a sequence of changes of display state culminating in a static image. It is clear that the way in which a concept of state is exploited in such a drawing procedure has no connection whatever with the informal meaning of the image that is to be displayed. In contrast, the state represented by the script has a quite fundamental and essential relationship to the interpretation of the associated image. Indeed, if our representation of a drawing does not include such a representation of state, it is impossible to demonstrate the precise correspondence between transformations

of the image and behaviour of the real object being represented through which the intended meaning of the image is expressed.

It is clear that the approach to graphics that we are advocating has more in common with modelling than conventional programming for graphics. The difference is underlined by the fact that conventional graphics systems tend to focus upon developing very sophisticated methods of describing static images. Though this typically encompasses many sophisticated techniques for transforming graphical objects (e.g. through grouping or stretching) this does not provide a framework for specifying the precise set of transformations of an object that is consistent with its semantics.

What are the advantages of adopting a modelling approach to graphics? In the first instance, a model of an object can be validated with reference to the application far more easily than a recipe for describing an image. Models can also be far more conveniently redesigned and enhanced whilst respecting their semantics. The purpose of this paper is to consider how the principles that we have applied to graphics can be extended to address other kinds of programming task.

It will doubtless be remarked that the role of modelling in programming is already served by existing paradigms. Can't we take the scripts above and recast them into functional scripts incorporating higher-order definitions as necessary to allow us to encapsulate all the transformations that we intend to perform upon doors and their contexts? Isn't modelling applications using objects and transformations exactly what object-oriented programming is advocating? These issues will be touched on later; they have naturally been significant in guiding our research, and fuller discussion of them appears elsewhere [8]. Our purpose here is to emphasise that the way in which different programming paradigms have been applied is generally inconsistent with a unified view of programming. For instance, impure functional and logic programming use default techniques for operational interpretation to generate complex executable programs whose status with respect to the application in no way resembles that of an abstract model. This is the kind of consideration that motivates intensional programming [18].

It may also be argued that modelling is only appropriate in programming for certain applications. In our view, modelling is inconspicuous in many contexts because the programming problem that is being solved and the computational framework within which its solution is to be expressed have already been so thoroughly analysed that they do not have to be explicitly modelled. This becomes clear in applications such as programming for embedded systems where there are generally no routine generic methods to solve the algorithmic problems and the architecture of the computational agents themselves can be freely designed.

Embedded systems and concurrent systems modelling

Modern computing is increasingly concerned with embedded systems. Programmable devices have to be integrated into a concurrent system that comprises human, mechanical and electronic components. In developing embedded systems we must take account of all relevant information about the components of the system whose behaviour cannot be modified and complement these components with electronic devices programmed to cooperate effectively to achieve the desired effect. The components in such a system typically interact in ways that are difficult to conceive outside a state-based framework. Each component is designed and programmed to respond to changes in state in its immediate environment and itself reacts by changing the state of other components.

Concurrent systems modelling has an essential role in programming embedded systems. A component cannot be programmed to react appropriately to its environment unless it incorporates a representation of the state of the system of some description. There are in fact two complementary aspects of modelling involved in the programming task:

- inferring the abstract behaviour of the unspecified components of the system from the explicit form and operation of the prespecified components
- prescribing the explicit form and operation of the programmable devices to conform with the intended abstract behaviour.

Addressing these issues effectively is essentially equivalent to developing a method of concurrent system modelling whereby the global behaviour of a system can be related to the activity of its components.

2. Concurrent systems modelling in programming for embedded systems

The objective of our research is to develop techniques that can model the changes of state that take place in a concurrent system as faithfully as we can model the effect of opening or relocating a door. The models we require will necessarily be more complex in nature than the sets of points and lines with protocols for transformation given above. However, we believe that they can also be described in terms of possible transformations of state performed by agents that correspond as faithfully to possible state changes in the application. These state changes will generally have to be conceptually more complicated. For instance, they may have to be conceived as transforming an object that presently admits one set of transformations into an object that allows an entirely different set.

The process of constructing such a model is clearly a complex design task: it involves analysis of the environment in which the programmable components operate, and synthesis of the components with reference to how they are responsive to their environment and how they are programmed to react. The interaction between these two kinds of activity in the design process is similar to the essential intertwining of requirements specification and design postulated in [3,17]. Analysis of the application indicates limitations in the design of devices. When a device is modified, further knowledge about the environment typically has to be acquired through analysis of the application.

Fundamental principles

The illustrative example above leads us to conclude that in attaching a meaning to computer representations of objects three interrelated ingredients have a fundamental role:

- a state-based representation
- agents and their privileges to change state
- an exact correspondence between state changes in the application and the model.

If such concepts are relevant to the specification of static images, they must surely be still more significant in constructing programs that have an explicit dynamic behaviour.

An approach to programming that is "state-based" or "agent-oriented" is sure to be viewed with suspicion, especially when its mathematical foundations are as yet only informally described. In fact, as we have remarked above, it is clear that state concepts can operate in many ways, and that better methods of modelling state-transition systems are essential if we are to develop models of computation that meet the needs of those designing algorithms for modern architectures, where there is an essential need to take account of different measures of complexity. The introduction of agents is conceptually essential when considering concurrent processing. Some of the difficulties encountered in specifying the semantics of conventional programming languages also relate to a potentially open-ended use of tools to generate complex procedural recipes. An overview of our current progress towards the development of an appropriate programming paradigm is given below, together

with a simple illustrative example. More detailed discussion of the principles and techniques introduced, together with other examples, can be found elsewhere [5,7,8].

Analysis of embedded systems: the role of variables

Analysis of an embedded system is based upon knowledge of the acceptable system states and responses. It can take the form of experiments that are either physical or imaginary. The object of such analysis is comprehension of the system that encompasses knowledge about particular states and responses but is not typically expressed in state-based terms. For instance, it may take the form of general propositions about the behaviour of the system and constraints upon the kinds of state that can arise.

The operation of an embedded system is primarily conceived in terms of observations of characteristic measurable quantities that determine its current state. For example, we can record the position of a lever, the content of an electronic memory, the temperature of a thermocouple. As in intensional programming, we shall associate variables with such quantities ("characteristic variables"): these variables acquire different values as the state of the system evolves in operation.

The status of the characteristic variables is a key issue in our research. We surely perceive transitions in a system in terms of entities with a single identity but a value that is subject to change; such change within identity is a commonplace and essential part of our cognitive framework. The concept of a variable that can assume different values conflicts with the traditional notion of a mathematical variable, however [18,15,9]. This conflict is resolved in intensional programming by introducing variables whose values can be viewed statically as preconceived "streams of values in time" or dynamically as representing successive state of a computation. There are acknowledged difficulties in applying such an approach to embedded systems. One is concerned with how changes in values are synchronised [18], the other with a methodological issue: to what extent the changes in values of variables can be preconceived.

Our approach is pragmatic at this point. Like the intensional programmer, we wish to avoid the anarchic use of variables to which any value can be arbitrarily assigned [18]. Rather than appeal to a concept of history and future for the values of a variable we prefer to specify the acceptable transformations of value that the characteristic variables of an application undergo formally, thereby implicitly describing the possible values they can attain in any particular simulation of the system. During the process of analysis, the notional values that can be assigned to variables, corresponding to the acceptable states of the system, can be changed by altering the set of possible transformations. It is not yet clear whether this approach to disciplining the changing values of variables is expressive enough to handle issues such as synchronisation and at the same time can be made mathematically respectable. Even so, there are promising indications from our current practical experience and initial steps towards a formal computational model.

States and transitions

Our approach can be motivated by examining the cognitive framework in which embedded systems are typically conceived. We have remarked that the state of a system is represented by families of variables. In understanding the significance of these variables, it is important to recognise the role played by simultaneous observation. As we observe the ways in which the values of the characteristic variables change as the system changes from state to state there are certain invariant relationships between values that are preserved through synchronised change. A central objective of our research is the precise representation of these changes.

The following discussion concerning the difference between the status of a candle and its reflections in a mirror, paraphrased from Russell [16], provides useful motivation:

When we examine the changes in [certain] groups of objects: there are those which affect only some member of the group, and those which make connected alterations in all the members of the group. If you put a candle in front of a mirror, and then hang a black cloth over the mirror, you alter only the reflection of the candle as seen from various places. If you shut your eyes, you alter its appearance to you, but not its appearance elsewhere ... In [these] cases, you do not regard the candle itself as having changed; you find that there are groups of changes connected with a different centre or a number of different centres. When you shut your eyes [...] the centre of changes that occur is in your eyes. But when you blow out the candle, its appearance everywhere is changed; [...] the change has happened to the candle. The changes that happen to an object are those that affect the whole group of events which centre about the object.

The synchronised changes of values of characteristic variables are modelled in our framework in a manner consistent with Russell's account. The variables whose values define the reflection of a candle, for instance, would have values that are functionally defined in terms of the position of the candle, so that when the candle is moved the reflection also moves in one indivisible transition. Only certain variables have values that can be independently changed - those that in Russell's classification lie at a centre of change. The reflection of the candle can only be changed through moving the candle.

Representing indivisible groups of changes

To describe such groups of associated state changes formally we make use of definitive - ie definition-based - state transition (DST) models [7]. In a DST model, the values of a system of variables are defined either explicitly or implicitly in terms of other variables in a non-cyclic fashion. The defining formulae make use of data types and operators from an underlying algebra of values chosen to suit the application. A simple reflection might be defined by a set of points and lines that is a geometric transformation of a set of points and lines defining an object for instance. A transition from one state to another is specified by redefining the value of variable: such redefinition could represent relocation of the object with an associated change in its reflection.

The concepts that underlie DST models bear closer examination. They include the functions that are used to describe the dependencies between variable values. Notice that - as in a spreadsheet - it is inappropriate to think of computation being involved in maintaining functional relationships between characteristic variables: this accords well with a declarative programming ethos. We do not have referentially transparent programming framework on the other hand; in DST models, scripts are used to represent state information. This has similar practical consequences to the introduction of intensional variables - there is apparently no longer an essential need for higher-order functions.

There is an important distinction between our use of definitions and constraints: a pure constraint is a universal statement about a class of valuations of a set of variables, whereas a definition serves to express specific transformations between one valuation and another [9]. The state-based representation we adopt can also be contrasted with an object-oriented modelling approach, in which state-changes are distributed between objects representing generic pieces of local state. An OOP approach does not assist the representation of centres of change in Russell's sense, since propagation of state-change is modelled by message-passing.

Agents and privileges

DST representations are the means by which we represent the admissible transformations of systems of characteristic variables. They establish the link between analysis and synthesis, and have an ambiguous declarative / procedural quality. For example, a system of definitions can be used to express known relationships between entities whose values are as yet undefined (as in a traditional use of a spreadsheet). On the other hand, when the values of all variables are fully defined, a DST model can be directly interpreted in computational terms.

As developed above, our approach deals only with the description of groups of state-changes that - in Russell's terminology - are associated with a single centre of change. The use of DST models in this mode is associated with a restricted kind of user-computer interaction in which the computer records the state of the model and the user acts to change the state through a sequence of redefinitions, as in our initial illustrative example [8]. In typical applications, the transitions in a system are associated with several simultaneous groups of state-changes about several distinct centres. Such concurrent activity is described in DST models by introducing parallel redefinition of variables. It is also possible for there to be interference between actions performed at different centres of state-change, as when a seesaw is simultaneously depressed at both ends.

In the analysis of an embedded system, concurrency is conceived as simultaneous action of one or more agents. The introduction of agents is commonplace in requirements specification [11] and echoes a Newtonian perspective whereby change occurs only through the action of agents. (It is a curious fact that the quotation from Russell above is taken from a discussion that disputes the usefulness of the concept of agent in physical theories.) In our development, the agent concept has a fundamental role in characterising the transformations of variable values that are deemed acceptable. One agent may be privileged to blow out the candle, another to move the candle, yet another to move the mirror; in general such privileges are predicated upon enabling conditions pertaining to the current state. In formulating the protocols by which agents can act to change system state, we find it useful to identify those aspects of the system (represented by characteristic variables) to which the agent can respond and those which it can conditionally manipulate.

The notation we introduce for this purpose is called LSD. For each agent, an LSD specification records the variables to which the agent can respond (oracle), those variables that it can conditionally change (state), and those variables whose values are defined by functional relationships (derivate). It also includes a protocol specifying the redefinitions that the agent is permitted to perform and the associated enabling conditions that must be met.

Faithful state-transition models for embedded systems

We connect the use of DST representations with a form of modelling that establishes a precise correspondence between states and transitions in the application and states and transitions in the model. When the candle is moved, there is conceptually no point in time at which the reflection of candle has yet to move; this fact is captured using a DST model. In conventional approaches to simulation such faithful state-transition models are unusual. There are typically intermediate states in the model that are inconsistent with an external interpretation. Such inconsistencies account for many of the difficulties encountered when debugging procedural programs: they are also part of the motivation behind the use of invariants and assertions in program development. Identifying the states of the model that

are meaningful in the application is a crucial aspect of the validation process that is commonly left to the programmer to infer through informal insight.

The critical reader will be under no illusions about the difficulty of developing a faithful model of a complex embedded system; this goal is certainly beyond the scope of our current methods in many cases. We argue only that any solution to this modelling problem demands that proper consideration must be given to all the respects in which application-oriented information influences the form and nature of the model. Three primary concerns are significant here:

- indivisible actions have to be accurately characterised and described
- the manner in which the agents in the system act in relation to their view of the system must be expressed and represented
- the way in which the concurrent action of agents is constrained by environmental factors (such as the speed of response, action and communication, and other relevant physical characteristics of the entities in the system) must be taken into account.

On present evidence, DST models will prove to be an effective way to represent indivisible actions. LSD specifications can be used to describe how an agent acts in isolation. An important characteristic of our modelling method is that one and the same action on the part of an agent can be associated with distinct indivisible state-changes in the system, depending upon context. We believe this to be a significant feature when composing models; for example, it enables us to specify that an agent can pull a lever prior to specifying what mechanism is thereby engaged.

In the use of LSD, it is tempting to introduce idealised forms of communication between agents that enforce synchronisation of actions as if this were directly imposed by the sensory input and protocols observed by the agents. This kind of modelling may be useful as a mathematical description of intended behaviour, but does not provide a satisfactory basis for prescribing programmable components, since it makes inappropriate engineering assumptions. If we avoid such abuse, LSD specifications are not generally executable in a meaningful way; they only capture that part of the synchronisation between actions that is based upon how agents respond to changes in their immediate environment. Without knowledge of how fast agents respond, how fast values are communicated, there will generally be many inappropriate ways in which agent actions can be synchronised [4,7].

Simulation and the abstract definitive machine

In their present state of development, our methods can be used only as the basis of simulations of system behaviour subject to environmental assumptions that are introduced in an explicit manner. In effect, the LSD specification of a system can be interpreted operationally provided that we take account of knowledge of each agent's particular characteristics. For instance, the interval of time between a door bell ringing and the door being opened - if indeed it is opened at all, is determined by quite different considerations from the interval between a diver jumping off a board and entering the water. Similarly, though two agents may be privileged to press the door bell, it may very well be physically impossible for them to do so at one and the same time.

Our simulations make use of a new machine model - the abstract definitive machine (ADM). In the ADM, the computational state is represented by means of a set of definitions. An ADM program is abstractly described by a system of entities, each of which comprises a set of definitions and actions. Each action consists of a guarded sequence of redefinitions and instructions to instantiate or delete instantiated entities. In the execution of a machine program, the computational state, as represented by the instantiated definitions,

is repeatedly modified through the parallel execution of actions whose guards evaluate to true in the current state.

Simulation of an LSD specification in the ADM involves the translation of LSD agent protocols into ADM entities incorporating parameters to reflect execution delays and the generally non-deterministic response of agents. In our present prototypes, these parameters are specified probabilistically in accordance with the expected characteristics of agents and their environment.

Simulation in the ADM computational model has many unusual characteristics [5,7]. Because the representation of state is readily interpreted in application-oriented terms, it is relatively easy for the user (the concurrent systems designer) to intervene in execution as appropriate, effectively exercising the privileges of an omnipotent and omniscient super-agent. Certain kinds of interference, such as that arising in the seesaw example mentioned above, can be detected within the ADM and referred to the designer for arbitration. In general, we attach importance to modelling systems faithfully to the point of admitting conflicts that can arise in practice; only in this way can the relationship between the behaviour of the system and the activity of the agents be fully understood.

An illustrative example

Our approach will be illustrated by means of a simple example. An electronic cat-flap is a device that is designed to keep out unauthorised cats and to give the owner control over when authorised cats are able to come in and go out. It takes the form of a cat-door large enough for a single cat to pass through that can operate in two modes, as an electronic device or manually, at the discretion of the owner. When the flap is electronically active the cat door opens to the inside only in response to a ferrite key attached to the cat's collar. The owner has additional control in the form of a locking device that can be used in conjunction with either the electronic or the manual mode of operation; this can be set to prevent the cat-door from opening inwards, outwards, or in either direction.

An LSD specification for the electronic cat-flap is given in Figure 1. The protocols for the man and cat agents reflect the different ways in which they can interact with the flap. The man is able to switch the flap between electronic and manual mode of operation and to alter the position of the 4-way lock. In entering or leaving the house, the cat expresses its intentions through putting pressure on the cat-flap. The cat flap responds according to its state. When it is not obstructed, the cat-flap returns to a closed position autonomously.

In identifying these possible system behaviours from the LSD specification, each agent is conceived as repeatedly monitoring the guards in its protocol before committing itself to performing the sequence of actions associated with one of its true guards. The choice of which action to perform is non-deterministic: the cat's intention is subject to change at any moment for example. For an intelligent operational interpretation, informal knowledge about the way in which agents operate is required. The cat-flap may return to its closed position either under the influence of gravity or through the action of spring; in either event, this response should occur immediately whenever the flap is open and unobstructed, and should be consistently fast in execution. In contrast, there will be few occasions on which the man chooses to exercise the privilege to change the status of the 4-way lock. It is also worth noting how the privileges of the man agent may capture rules for intelligent operation of the cat-flap (cf the concerns raised by Pylyshyn in [14]).

Other issues concern the speed with which oracles are updated or consulted. There are scenarios consistent with the privileges of the agents that lead to inappropriate system states. The 4-way lock might just be set whilst the flap is open. It is feasible (presumably

even in practical use) for two cats to come through the cat flap head to tail as if they were one cat. The parameters adopted for simulation of the specification in the ADM are chosen to reflect reasonable assumptions about the relative speed at which agents operate and signals are received. In the specification, the synchronisation between sensory input to the electronic cat-flap and the state of the electronic lock is deemed to be exact, and is represented by a derivative.

3. Concluding remarks

We have described and illustrated the fundamental concepts underlying a new style of programming that has been under development for several years. The motivating ideas behind this work reflect the strong influence of intensional programming: our approach represents an alternative way to address the problem of using variables to describe state-changes without resorting to a totally undisciplined use of assignment. At the present time, our methods are best suited for informal and experimental use, but we hope that this can be remedied through extension of the fundamental concepts, more precisely specified mathematical models and techniques, and the development of better practical software prototypes.

We believe that our research indicates the importance of developing a programming paradigm that takes full account of the exceedingly rich cognitive input required in the construction of embedded systems. This imposes a discipline upon the programmer that has at least as much in common with experimental science and engineering as it has with mathematics. If this is the correct perception of the programming task, it suggests rather different priorities from those that have so far been the dominant influence on the theory of programming language design.

References

- [1] Backus J *Can programming be liberated from the von Neumann style?* Turing Award Lecture 1977, CACM 21, 8 (August) 1978, 613-641
- [2] 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
- [3] 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
- [4] 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
- [5] Beynon W M, Slade M D, Yung YW *Parallel computation in definitive models*, CONPAR 88, BCS Workshop Series CUP 1989, 359-366
- [6] Beynon W M, *Parallelism in a definitive programming framework*, Proc Parallel Computing 89, Leiden Sept 1989, to appear
- [7] Beynon W M, Norris M T, Orr R A, Slade M D *Definitive specification of concurrent systems*, UKIT'90, Southampton March 1990, to appear
- [8] Beynon W M *Evaluating definitive principles for interactive graphics*, New Advances in Computer Graphics, Springer-Verlag 1989, 291-303
- [9] Beynon W M, Russ S B *Variables in Mathematics and Computer Science*, RR#141, Dept of Computer Science, University of Warwick, 1989
- [10] Bird R, Wadler P *Introduction to Functional Programming*, Prentice-Hall 1989
- [11] Davis A M *A comparison of techniques for the specification of external system behaviour*, CACM(31) 1988, 1098-1115
- [12] Hoare, C A R *Communicating Sequential Processes*, Prentice-Hall 1985

- [13] Johnson W L *Deriving Specifications from Requirements*, Proc 10th Int Conf on Software Engineering, Singapore, 428-438, 1988
- [14] Pylyshyn Z W *Computation and Cognition: Toward a Foundation for Cognitive Science*, MIT Press 1984
- [15] Revesz, G E *Lambda-calculus, Combinators and Functional Programming*, Cambridge Tracts in Theoretical Computer Science, Vol 4
- [16] Russell B *ABC of Relativity*, 3rd Edition, Allen and Unwin 1969
- [17] Sartout W, Balzer R *On the inevitable intertwining of specification and implementation* CACM, 25 (7) (July), 438-440
- [18] Wadge W W, Ashcroft E A *Lucid, the dataflow programming language*, Academic Press, 1985


```

agent flap() {
state
    (int)    #lflap = 5           // the length of the flap
    (int)    #angle = 0          // the inclination of the flap
    (bool)   #switch = true      // whether the electronic lock is operating
    (int)    #fourWayLock = 0    // fourWayLock =
                                // 0 - the cat can pass through from either direction
                                // 1 - the cat can go out only
                                // 2 - the cat can go in only
                                // 3 - the cat cannot pass through from either direction
    (int)    #Radius = 10       // range of the detector of the electronic lock

oracle
    (bool)   #pushOut
    (bool)   #pushIn
    (int)    pos

derivate
    (bool) #elecLock = abs(pos) > Radius ^ switch
    (bool) #canPushOut = angle != 0 \ / !elecLock ^ (fourWayLock == 0 \ / fourWayLock == 1)
    (bool) #canPushIn = angle != 0 \ / !elecLock ^ (fourWayLock == 0 \ / fourWayLock == 2)

protocol
    pushOut ^ canPushOut -> angle = |angle| + 1
    pushIn ^ canPushIn -> angle = |angle| - 1
    !pushOut ^ !pushIn ^ angle > 0 -> angle = |angle| - 1
    !pushOut ^ !pushIn ^ angle < 0 -> angle = |angle| + 1
}

// the 4-way-lock is rotated only when the flap is closed
agent man() {
state-oracle
    (bool)   switch
    (int)    fourWayLock

oracle
    (int)    angle

protocol
    switch == false -> switch = true
    switch == true -> switch = false
    angle == 0 ^ fourWayLock != 0 -> fourWayLock = 0
    angle == 0 ^ fourWayLock != 1 -> fourWayLock = 1
    angle == 0 ^ fourWayLock != 2 -> fourWayLock = 2
    angle == 0 ^ fourWayLock != 3 -> fourWayLock = 3
}

agent cat() {
state
    (int)    #height = 2        // height of the cat
    (int)    #pos               // pos > 0 - outside the house; < 0 inside the house
    (int)    #intention         // intention =
                                // 1 - is going out
                                // 0 - is staying put
                                // -1 - is coming in

oracle
    (int)    lflap, angle

derivate
    (bool) #obstructOut = lflap * tan(angle) > pos > (lflap - height) * tan(angle) - 1
    (bool) #obstructIn = lflap * tan(angle) < pos < (lflap - height) * tan(angle) + 1
    (bool) pushOut = intention > 0 ^ obstructOut
    (bool) pushIn = intention < 0 ^ obstructIn

protocol
    intention > 0 ^ !obstructOut -> pos = |pos| + 1
    intention < 0 ^ !obstructIn -> pos = |pos| - 1
    true -> intention = 1
    true -> intention = 0
    true -> intention = -1
}

```

Figure 1: A LSD specification of a cat flap