An Exercise in System Representations

Kreutzer, Wolfgang; Åström, Karl Johan

1987

Document Version:
Publisher's PDF, also known as Version of record

CODEN: LUTFD2/(TFRT-7369)/1-077/(1987)

An Exercise in System Representations

Wolfgang Kreutzer
Karl Johan Åström

Department of Automatic Control
Lund Institute of Technology
September 1987

Department of Automatic Control
Lund Institute of Technology
P.O. Box 118
S-221 00 Lund Sweden

Document name
INTERNAL REPORT

Date of issue
September 1987

Document Number
CODEN:LUTFD2/(TFRT-7369)/1-077/(1987)

Author(s)
Wolfgang Kreutzer
Karl Johan Åström

Supervisor

Sponsoring organisation

Title and subtitle
An Exercise in Systems Representation

Abstract

This report deals with different ways for system representations. The goal is to explore simple but powerful ways of describing complex control systems hierarchically.

It is shown that an object-oriented approach where systems are objects with inputs, outputs, subsystems, connections and behaviour is a convenient approach.

A small prototype system to explore this notion is developed. The prototype is described in detail.

Key words

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title

ISBN

Language
English

Number of pages
77

Recipient's notes

Security classification

1. Basic Requirements of Systems Modelling & Analysis

1.1 Principles of Model Design & Implementation

The craft of model building has a long history in science and everyday life and is central to the way in which we understand the world around us. A human's model of her world depends on knowledge and reasoning strategies encoded in both the brain's hardware and symbol structures acquired during a lifetime of experiences; it guides the way we perceive, analyse and react to our environment. The act of model building explores alternative ways to represent this knowledge, while analysis and simulation are used in its application.
Rivett (1972), Klir (1972), Davis & Hersch (1981), Sloman (1978), Savage (1978), Piatelli-Palmarini (1980), Olsson (1980) and Pylyshyn (1984) offer a selection of interesting perspective for anyone who would like to explore the roots of the underlying concepts.

Figure 1.1 shows a much simplified picture of modelling as a mental activity. Four distinct phases are identified. The first of these is often referred to as **system identification**. It establishes relationships between an observer and some part of reality. A **system** is defined as a collection of objects, their relationships and behaviour relevant to a set of purposes. This definition stresses the fact that modelling should be a purpose-driven activity. There is never a best or even "correct" model. Its usefulness always depends on objectives and contexts of application.

The second phase is that of **system representation**, where symbolic images of objects, relationships and behaviour patterns are bound into structures as parts of larger mental frameworks of beliefs, background assumptions and theories owned by a problem solver. The completely disinterested observer is just a convenient fiction. As humans we can not avoid to be theory-driven. Our theories can be descriptive, used for explanation and prediction, or they can be normative, prescribing what ought to be done. Different kinds of theories will yield different kinds of models.

The third phase deals with **model design**. A **model** is an "appropriate" representation of structures and processes of a miniworld, instantiating some aspects of a theory. Humans continually explore models to reflect about situations, be it consciously ("Gedanken-experiments") or subconsciuosly. Models can quiclékly grow into very complicated structures. Constraint and control of their complexity forms the heart of any modelling methodology.
As a general rule ("Occam's razor") one should strive to keep models as simple as possible:

"It is vain to multiply entities without need".

Model complexity can range from a one-to-one image to a few symbols on the proverbial back of an envelope. One should always strive for the greater clarity of simple representations; although simplicity is itself an elusive concepts. It may best be described as a relation between what is to be represented, the medium of representation and an interpreting observer. We seem to have some natural tendency to strive for both a minimum set of concepts and a high degree of visualization. The first helps us to reason about the essence of complex systems while the second allows us to perceive systems as wholes.

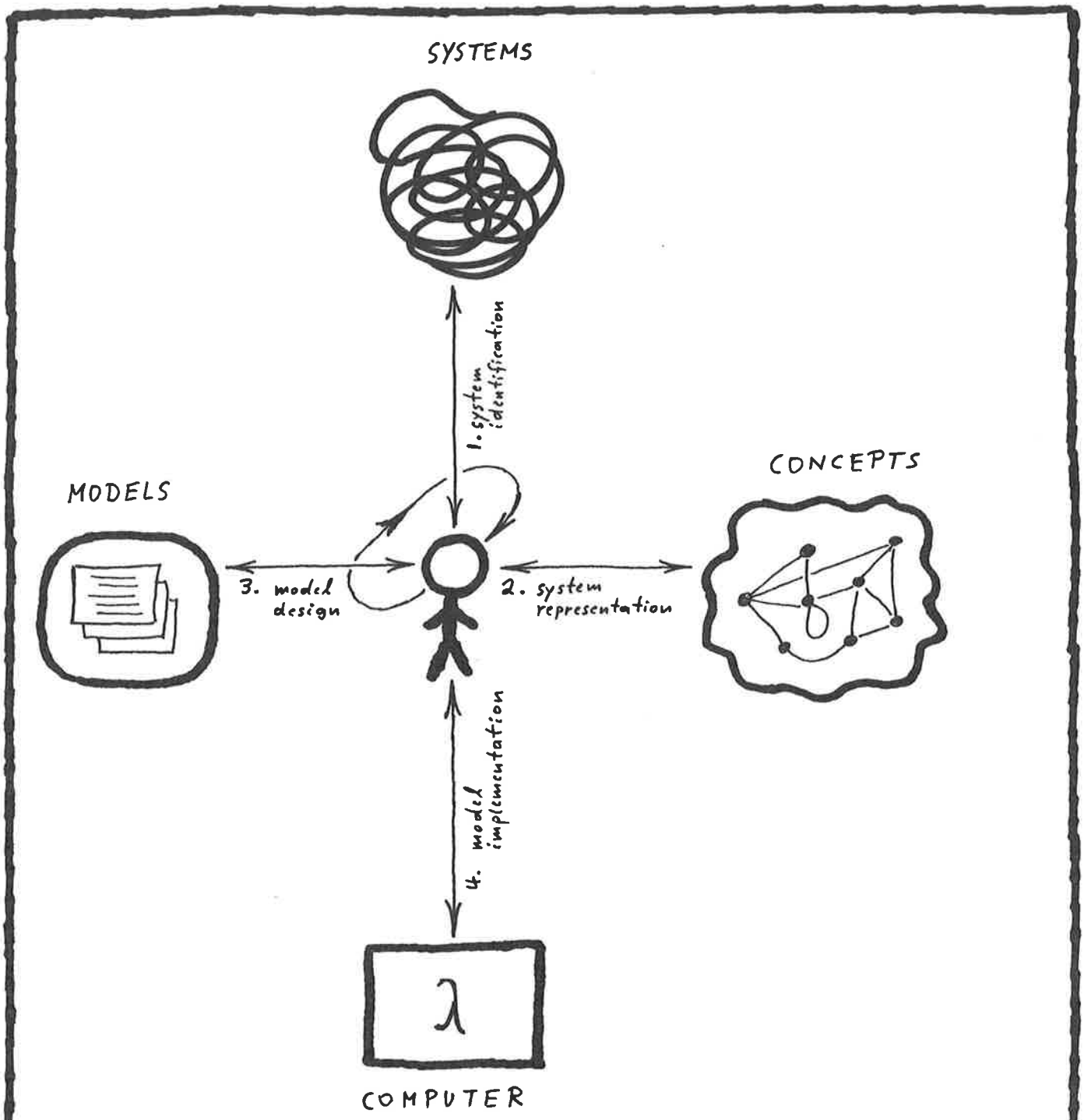


Figure 1.1: A simple Model of the Modelling Process.

Any useful model has to simplify and idealise. Abstraction and aggregation are important techniques to keep models of complex phenomena intellectually manageable. Great care must be taken to preserve all and only those of a system's characteristics which are essential. This again depends on a model's purpose.

Phase four of our model of the modelling process assumes that a computer is used. It may be referred to as the **model coding** stage. Her we seek a formal representation of symbol structures and their transformations into data structures and computational procedures provided by some programming language.

Models must be analysed to be useful; i.e. to further our understanding of some facet of a system, or predict the effects of alternative courses of action. Such analysis can be deductive, based on a theory about certain classes of structures, or it can be inductive, based on observation of a model in action. Both strategies assume that the model's **validity** can be established in some way. On a "rational" basis validity can be proven through formal correspodences between the model and the theory it was designed to instantiate; i.e. by showing how its behaviour can be logically derived from analysis of purely formal properties of its representation. A model's validity may also be empirically corroborated; i.e. via a set of correspondences between its own and the observed behaviour of the structure or process it is a model of. The ultimate test will, however, always be a pragmatic one: "does it serve its purpose ?". This may involve an improved understanding of some system, better decision making, or accurate prediction of future events. In all these cases there are quite different criteria according to which success will ultimately be judged.

Verification seeks correspondences between a model and its programmed representation. This can again be established empirically or on a purely rational basis. Empirical corroboration is achieved through extensive testing, while a proof of correctness according to formal specifications would be the rational alternative. The current state of the art in program proving, however, makes this a rather futile approach for any but the simplest of models. We have to settle for less than certainty, just as we can never be sure of the ultimate truth of our theories and specifications. These too can only be increasingly corroborated but never conclusively established. Corroboration of theories may be attempted through unsuccessful falsification or in some other way, depending on one's personal philosophical outlook.
Brown (1977) gives an instructive account of Falsificationism and competing philosophies of science, whose basec principles should be familiar to any practicing scientist.

Modelling can have two considerably different goals. Its more traditional application is in decision making. Here the emphasis is on generating numeric information; i.e. used for comparing, tuning or predicting a system's performance.
A different kind of model strives not so much for numeric solutions, but for improved understanding of complex systems. It supports a more exploratory, speculative style of model analysis. A quote from Hamming (1962) nicely summarizes the essence of this modelling style:

"The purpose of modelling is insight not Numbers".

Relatively little is known about a system. A well defined problem may not even exist. We are using modelling at a prototheoretical stage; as a vehicle for

thought-experiments. Its purpose lies in the acts of construction and exploration, and in our improved intuition about a system's behaviour, essential aspects, sensitivities, ... which we gain in the process. The model may never be used to genrate any numerical results at all.

This kind of modelling may well be the only feasible way to check the consistency and explore all consequences of complex theories. AI-miniworlds (see Winston (1984), Rich (1983)) are models in this sense. Many people believe that the potential of computer-aided modelling for this style of exploration of symbol structures will in future overshadow its importance in its traditional application as an aid to decision making.

1.2 Programming Languages for Systems Modelling & Analysis

There are intimate relationships between thought and language and language and methodology. This is well expressed by the following Wittgenstein (1961) quotation:

> "The limits of my language are the limits of my world".

The importance of appropriate conceptual and notational tools which guide and structure a modelling activity during all its phases can hardly be overstated. A well designed system description and programming language is not only a notation used to communicate with computer systems. It is also a cognitive device, which structures perception, representation and reasoning about classes of systems and problems.

Simplicity, expressiveness, flexibility and efficiency are at least partially conflicting goals between which suitable tradeoffs have to be made. In the long run nothing is gained from restricting a tool to too low a level. Improved machine efficiencies can be achieved by binding languages closer to machine structures. This can, however, lead to disproportional increases in development, debugging and maintenance efforts.

Final success of any tool depends critically on user acceptance and satisfaction. If a programming notation fails to provide effective and natural modelling styles, then separate methodogies, often ill-structured and inconsistent, have to be manually mapped into low-level abstractions.Unfortunately this may require highly error-prone transformations across large conceptual distances.

In stressing the priority of methodology over notation it has sometimes been argued that methodologies and implementation languages should be viewed as separate issues; that one should program into a language, not in it. This statement sweeps the real problem somewhat under the carpet. We still need a web of concepts to express, comprehend and use any methodology; and we surely need a methodology to guide us in any complicated task. Since language is possibly the most important tool of thought, concepts and methodologies can never be truly independent of formal or informal representation in some notation.

A good notation must suggest rather than hinder an appropriate representation of relevant ideas. It should be a catalyst which lets you "see" solutions rather than get in the way of what one is trying to express.

There is therefore a sore need for more research into relationships and tradeoffs between mind, concepts, representation and notation. Unfortunately neither mathematics nor psychology, the two disciplines one would most obviously turn to for help, offer much guidance. Although this topic is "difficult" and straddles traditional boundaries between disciplines, it should be at the core of cognitive psychology and computer science. Wittgenstein (1953 & 1956), Wickelgren (1974), Sloman (1978),

Hofstadter (1979), Olsson (1980) and Pylyshyn (1984) offer many stimulating ideas and insights; as does the literature on knowledge represenation for artificial intelligence programs.

Effective symbol systems must fit an individual's overall conceptual framework, offering familiar ideas and mnemonics. Multi-level **chunking**, where clusters of related concepts are aggregated and encoded by single symbols at higher semantic levels, can accomplish this taks. The importance of such encapsulation strategies for procedure and data abstraction has often not been fully recognised by programming language designers. Modelling methodology still lacks a satisfactory theoretical foundation. Some progress has been made in recent years, but it has generally not been incorporated in language designs yet.
Much can be learned from research into representation and structuring of domain specific knowledge, a filed which is explored by the artificial intelligence community.

Notational devices can be classified in a variety of ways. They can be formal or informal, one-dimensional or graphical. Informal tools like pseudocodes and some graphical symbol systems are especially useful during system identification and model design, whereas formal tools must be used for final representation and coding.
Advantages of graphical representations are that structural connectivity and symmetries are emphasized. Things that are conceptually related "appear" to be close. They also provide a rich syntax to visually define such concepts as "links", "flows", "directions", ... For most people these seems easier to grasp than their equivalent procedural encodings. The reason seems to lie in the inherent parallelism of the human visual system and the corresponding importance of visualization as a problem solving tool. Graphical notations' most frequently cited disadvantage lies in the fact that hierachies of levels must be used for complex structures (i.e. to fit them "on a single page"), but this may actually beneficial.

For many purposes classical mathematical concepts do not seem rich enough, whereas informal notations are too unstructured and ambiguous. Computer programming languages provide formal concepts to describe classes of objects and processes at fairly high levels. Translator programs can automate the process of mapping from "higher" to "lower" levels, a method which is far safer and less error-prone than manual tranformations by the model builder.

The "psychology of programming" is just beginning to establish itself as a respectable subject. Shneiderman (1980), Smith & Green (1980) and Sheil (1981 & 1983) are first steps in this direction. Comparing languages, including programming languages, for "goodness" or "badness" is a very difficult undertaking. Beauty is always in the eyes of the beholders. Quality is best thougth of as a relationship between a tool, a user and an application; with complex interactions between them. To paraphrase Winograd (1983): "To ask whether LISP is a better language than PASCAL is somewhat like asking whether a jellyfish is a better animal than a dinosaur. One should use both, although in their proper place. They have different strength and weaknesses and one is possibly better suited to certain tasks than the other; although even that will be difficult to establish empirically.
The author does not believe in the merit of any "universal" language, be it for programming or any other task. Even natural language is not equally well suited to everything. This is well demonstrated by specialized jargons for different areas of human discourse (scientists, engineers, doctors, lawyers, programmers (!), ...). Languages are cognitive tools which should be appropriate for the tasks to which they

are put and provide concepts familiar in the relevant "cultural context". Multi-purpose tools (i.e. "basic" mathematics) can work very well for a wide range of tasks, but some tasks will always require the higher power and precision of more specialized ones.
A number of textbooks have been written to compare programming languages; offering interesting insights into trade-offs between basic principles involved in language design, use and translation. Barron (1977), Tennent (1981) and MacLennan (1983) are some of the author's favourites.

Any recommendation on what style and tool to use is necessarily made on a somewhat subjective basis. This is how it should be, as long as the range of alternatives and some measure of their potential and limitations is known.
It is extremely difficult to convince people of the merits of switching to a different notation, regardless of its intellectual or practical superiority. Once user communities have entrenched themselves they usually remain very loyal to their chosen programming languages. Proponents (inmates) of a particular language culture hardly ever switch; although they will eventually retire.
The best which can be hoped for is to identify different programming paradigms and styles, define and exemplify their characteristics, and discuss their strengths and weaknesses. Some personal bias will always remain.

The author believes that object-oriented programming, as originated in **Simula** and more recently popularized by **Smalltalk** and so called "actor" languages, has the best potential for describing complex systems in a well structured and natural way. Its advantages may well become more widely appreciated in the future, since it becomes more and more evident that conceptual closure, context building and reasoning by analogy and default are central to the way we cope with the complexities of the world around us. It therefore seems only reasonable to demand that computer systems should support such problem solving styles, embedded in flexible, user-friendly and robust programming environments.
Some other interesting developments are clearly visible. With the dramatic drop in hardware prices the use of powerful personal workstations to host modelling tools has now become economically viable. The current trend points away from large central batch and time sharing systems, and towards interactive and integrated programming environments and graphically supported user interfaces hosted by single user machines which, via data communication networks, may draw on shared peripheral resources (i.e. mass storage, printers, ...) locally and on specialized information or processing power non-locally.
Well-designed interfaces for man / machine interaction should offer complete contexts for interactive model design and execution. Graphical displays of model structure, state and behaviour should be possible within such a framework, which should also embed a range of visually pleasing graphical symbol systems in a window-based, interaction-driven modelling environment.
A range of tools should be available to design and implement system models, as well as for any other complex programming task. Graphical symbol systems, pseudocodes, general and specialized programming languages all have their place in such a toolbox, alongside a rich repertoire of knowledge about their safe application.
Model implementation is just one of a range of activities we must perform during a systems analysis. These other ones include system identification, model specification and documentation, data analysis, model validation, solution or experimentation.
A toolbox for this domain should ideally be designed around a small set of simple, well-behaved building blocks and bound into a frame which is founded on some uniform and natural metaphor.

1.3 Some Fundamental Issues in Modelling Control Systems

In this study we wanted to consider the requirements for modelling and analysis of simple control system, using exploratory (Lisp) and object-oriented (Smalltalk) programming styles.

The basic relevant concepts are quite readily identified. We need to describe **systems**, which can be characterized by their inputs, states, outputs, and their behaviour. They may have quite a rich structure. It therefore seems sensible to permit partitioning of models into system / subsystem hierarchies. All component systems can be of different types and may be described in terms of state models, as input / output relations like impulse responses, or as transfer functions.

Many different **represenations of systems** are used in control theory. The ordinary differential equation model:

(**Eqn. 1.1**) $dx/dt = f(x, u, t)$; $y = g(x, u, t)$

where **x** is the state vector, **u** the input vector and **y** the output vector, is a common case. Often the fundamental form of the equation is not (1.1), where the derivative is solved explicitely, but rather

(**Eqn. 1.2**) **F** $(dx/dt, x, u, t) = 0$; **G** $(x, y, u, t) = 0$

Our discussion is restricted to systems of type (1.1). Partial differential equations and differential equations with time delay are also common. Again we restrict ourselves to differential equation models.

Linear systems, where the functions **f** and **g** take the form:

(**Eqn. 1.3**) $f = A(t)\,x + B(t)\,u$; $g = C(t)\,x + D(t)\,u$

are an important special case. For linear systems it is also possible to use different representations like input / output models of the form

(**Eqn. 1.4**) $$d^n y / dt^n + A_1 * d^{(n-1)}y / dt^{(n-1)} + \ldots + A_n y = B_1 * d^{(n-1)}y / dt + B_2 * d^{(n-2)}y / dt^{(n-2)} + \ldots + B_n u$$

which can also be represented by the matrix fraction

(**Eqn. 1.5**) $G(s) = A^{-1}(s)\,B(s)$

where **A** and **B** symbolize the polynomials

$$A(s) = s_n + A_1 s^{(n-1)} + \ldots + a_n; \quad B(s) = B_1 s^{(n-1)} + B_2 s^{(n-2)} + \ldots + B_n$$

The discrete time versions of (1.1), (1.2), (1.3), (1.4) and (1.5) are also needed.

It is a key issue to find suitable computer **representations**. The problem of multiple representations can be solved by data bases. In a typical example a system is represented by both a transfer function and a state equation. Small systems are not much of a problem, because transformation from one form to the other can be preformed "when needed". Such computations may, however, be extensive for large

systems. To obtain a reasonable degree of efficiency it then becomes necessary to store both descriptions.
It may also be desirable to have models of different complexity for the same physical object, as well as linearized models for different operating conditions. Since it is very difficult to visualize all possible combinations a priori, it is useful to have a flexible data base which permits modifications of the data structures without extensive reprogramming and recompilation.

System interconnection is another fundamental issue. The elementary types of connections are series, parrallel and feedback. For more complex systems it is desirable to have appropriate notations for hierarchical structures in which details of subsystems can be hidden, and that signals and variables at lower levels can be accessed in a well controlled fashion.

Apart from interconnection and decomposition there are many other relevant operations on system models, i. e. linearization, computing equilibrium values, system inversion and simulation. For linear systems it is also natural to transform coordinates, compute poles and zeros, determine observability and controllability, and perform Kalman decomposition. Some of these operations are conveniently done numerically. Others require formula manipulation.

More specifically, the following operations should be supported:

- Find all subsystems
- Find all interconnections
- Compute steady state
  - compute x, given u
  - compute the rest, given partial x, u
- Compute the steady state output map
- Linearize
- Compute poles and zeros
- Compute transfer functions
- Compute frequency curves
- Compact descriptions
  - eliminate all internal subsystems
- Compute sensitivity functions
- Compute well-conditioned linear representations
  - break to subsystems
  - scaling
  - reduce to scalar product form

Chapter 3 of this study will report on an experiment to provide the necessary primitives for these operations in a levelled and object-oriented design. Initially onl systemy decomposition and linearization were considered.

2. Three Paradigms of Programming: A Birds' Eye View.

2.1 "Structured" Programming: the classical approach.

Classical programming style sees program development as closely related to theorem proving. Both should proceed from axioms, in small, carefully considered steps whose correctness must always be assured. First a program's axioms should be defined by precise, consistent and unambiguous specifications and from there on proof and program should be developed "hand in hand".
This is the cornerstone of an intellectually appealing paradigm, sometimes referred to as "structured" programming, comfortably close to the customs of mathematics. Its main concerns are correctness and efficiency. Its methods and tools have taken many years to reach their current sophistication and many of the most famous computer scientists have been involved in its development or subscribe to it. It is well summarized and demonstrated by a number of influential books (Wulf et al. (1982) gives a particularly lucid account) and taught by practically all computer science departments around the world.
Its application has led to the development of strongly typed languages (**Pascal**, **Modula**, **Ada**, **Euclid**, **Edison**, ...). Type information can be used to check the validity of assignments and optimize storage allocation, which allows the compiler to guard and assist the programmer. New languages are even being designed with a view to avoid potentially dangerous (i.e. those which threaten a program's provability) or inherently inefficient features.

The basic principles of structured programming assume that you start from clear, unambiguous and immutable specifications. You can then design and bind a suitable module structure and their interfaces. The modules themselves will be implemented using the principle of "information hiding", so that no module can see more of the global structure than it needs to know. Modules can be programmed, compiled and verified separately, and the compiler will faithfully assist in error detection.

This methodology has been successfully applied to a wide range of problems and works well for many of them. With further improvements in the field of program verification it will eventually be able to give a reasonable guarantee of correctness.
There are, unfortunately, some inherent difficulties. Once data and module structures have been frozen (which must be done quite early) it is difficult (**very** messy) to make changes cutting across it. Instead of trying to patch the program it may in fact be wiser to just start from scratch again. The methodology requires stable specifications and demands a very early committment to the "correct " primitives and structures.
It is therefore unable to cope satisfactorily with large and complex systems which may be not at all well understood at the outset. Being able to change one's mind about basic assumptions without having to redo most of the work is essential here. While some acceptable intuition of correctness (although in a weaker sense than provability) is of course still required, machine efficiencies are secondary and may be relegated to an optimization phase once the program has stabilized.

"Exploratory programming", as practiced by the Artificial Intelligence (**AI**) community can cope with these problems. Interestingly their is a clash of cultures here. Many of the most influential people rooted in the classical style will argue that one should not try to do program what one does not yet understand. One should instead try to analyze and wait until the appropriate level of understanding is reached. If you adhere to this

principle, knowledge will safely (but only if all the basic assumptions were correct !) be accumulated in small and thoroughly analyzed building blocks. Some people may fear that at the current rate of progress we will unfortunately not be able to tackle many of the most interesting and urgent problems for a long time, if ever, in this way. Without the use of heuristics instead of formal deduction the human species would surely have not survived. The ideals of completeness and provability are clearly inappropriate in some contexts. It is also doubtful whether for most people the program development process really proceeds in such a clean and linear fashion as is exhibited in may textbooks. The heated arguments about the "context of justification" and "the context of discovery" within the philosophy of science are clearly also relevant here.

Finally one should bear in mind that the process of designing computer-based models ansd observing their behaviour is also an excellent way to learn. Often it is the only way to gain an improved understanding of complex phenomena.

2.2 "Exploratory" Programming: Lisp & Smalltalk

Classical programming has been the mainstream of computing for the last three decades. Attracting little attention outside the AI community, exploratory approaches to program development have quietly flourished for almost the same length of time. Lately there has been a sudden surge of interest in this programming style, mainly spawned by publicity generated through the expert system movement.

From the viewpoint of exploratory programming one tries to find the final program structure through a spiral process of approximation. Initially there are no well defined specifications, just some (usually overambitious) selection of problems one wants to understand and provide a solution for. Being able to change one's mind in midstream becomes essential.
In this paradigm we want to committ ourselves as late as possible. Above anything else we need flexibility and the ability to explore the space of a program's behaviour interactively. Good programming methodology, however, is still important, because it tells us how to decompose systems into primitives and how to combine these to form higher-level structures.

While we require flexibility and freedom from formal restriction we must somehow still be able to cope with the complexities we create. We therefore need to proceed via well chosen levels of abstraction, both for data and procedure objects. This strategy is well demonstrated by Abelson et al. (1985). We also need some means of chunking structures into concepts, to establish and inquire about contexts and the bindings of all its objects. Good debugging, tracing, bookkeeping, project management and documentation tools , which automatically handle much distracting detail, are a necessity. All these should be part of a uniform programming environment, with a consistent set of conventions and styles of interaction.
There must also be tools for analysis and compilation of modules to improve their performance characteristics, once a part of the program has stabilized.

Over the last few years so called "object oriented" design methodologies have further extended the general frame of exploratory programming. These techniques were pioneered by Simula and elaborated by Smalltalk. Nowadays there is an increasing number of Lisp-based systems offering these features in a slightly modified way (see figure 3.6). They are often referred to as **"flavours"** and are by now an integral part of

any serious expert system building tool (i.e. **LOOPS**, **KEE**, **Babylon** ...).
The reason for their growing popularity lies in their powerful encapsulation techniques and their facilities for "programming by analogy". The first feature enables us to cluster the definitions of a data type and all its relevant operators into one textual module with well defined interfaces. This can be used to greatly enhance a program's understandability and reliability. The second idea allows the specification of new program modules by stating the way in which they differ from already existing ones. All properties (data structures and procedures) of such a module's superclasses will automatically be inherited without having to repeat their specification. Since many objects, actions and situations are perceived by their similarities to ones already familiar to us and are therefore easily described by classification nets or hierarchies, this strategy produces a high degree of cognitive economy and a considerable simplification of complicated system descriptions.

This methodology and the associated programming tools allow us to tackle many complex problems, whose understanding would otherwise just be outside our reach. "Typeless" languages like **Lisp** and **Smalltalk** and the environments in which they are embedded can provide all of the features and tools we require to program in an exploratory style. The spiral process of building a sequence of appropriate models approximating a given reference system will yield an improved understanding and, eventually, a useful program.

2.3 "Declarative" Programming: Prolog

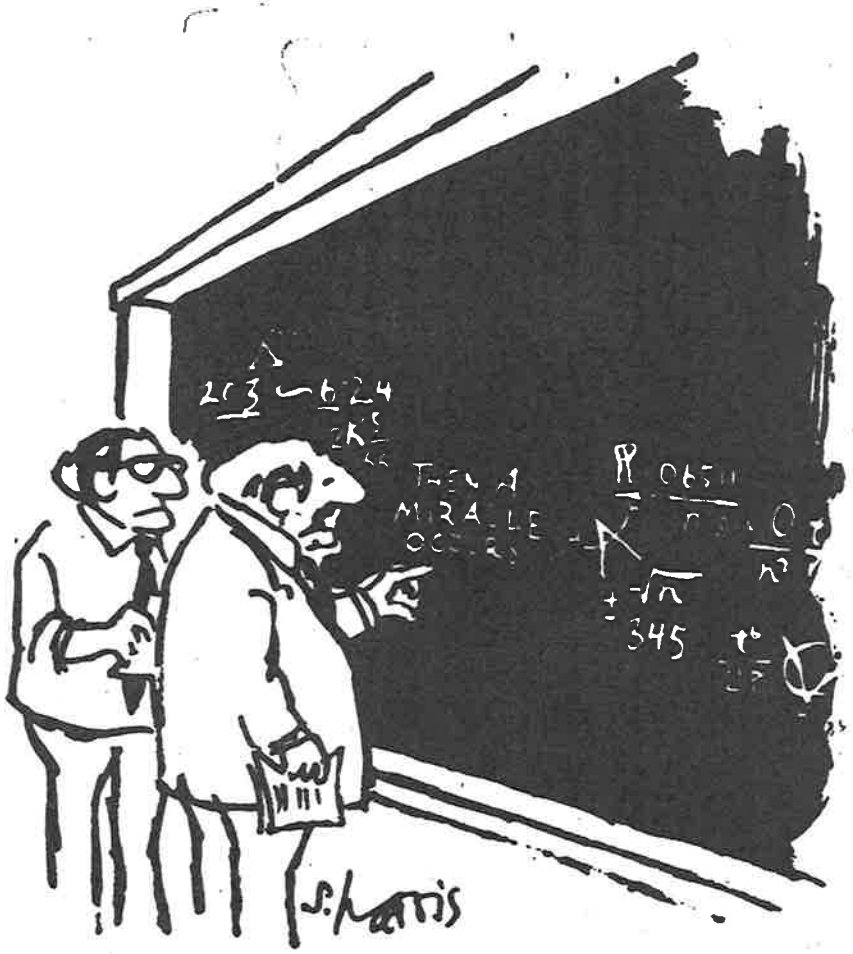
A third style of program development has recently emerged from research into artificial intelligence (natural language understanding in particular). It builds on classical logic as a foundation for system description.
Logic has a long and distinguished history as a scientific notation. Its properties have been extensively researched and its suitability for the analysis of (static) **structures** is indisputed. Programs, however, are commonly viewed as templates for **processes**, whose states change dynamically over time. While one will therefore normally write programs as recipes for computing solutions to a class of problems from a set of inputs, the logic programming movement tries to redefine programs as formal specifications. Here we need only state a class of solutions' formal properties, without prescribing how a candidate solution may be found.
Computation of course still has to take place, but here it is hidden behind the scene. To support this programming style we need a universal search program, a theorem prover, which explores the programs' state spaces in some systematic (and hopefully "intelligent") fashion. Although Robinson's resolution algorithm and advances in computer architectures have greatly increased the efficiencies of this process, speed of deduction (or rather the lack of it) is still the major stumbling block of this idea. Kowalski (1979) gives a good summary of the principles logic programming.

Uniform, **domain-independent deduction machines** are necessarily are the central idea of this paradigm. They are necessarily directionless; quite dramatic speed-ups can be achieved by incorporating domain-specific heuristics in a logic program, which guide the deduction process away from "non-promising" paths. All tools catering for this programming style provide features (clause ordering, cuts, ...) supporting such strategies. Their use, however, endangers the generality of a program (i.e. it may not always find a solution, even though it exists) and the provability of its conjectures. It invariably violates the spirit of declarative programming.

To tackle other than simple problems we must therefore hope for dramatic advances in machine efficiency, i.e. brought about by VLSI and massive parallelism.

Even then we are still faced with the problem of structuring large programs (assertions) so that they can be "understood" by humans, a problem which proves elusive and which has as yet not been solved within the framework of formal logic.
Since logic has no notion of "context" or any equivalent feature, there is no handle to introduce the idea of chunking. All assertions (clauses) are global and independant of one another. We can of course define higher level concepts from lower level ones (i.e. by simply writing the relevant compound clauses), but we can not group clauses into modules or in other any way restrict the scope of our names and definitions. This makes large logic programs unwieldy and structure must be imposed solely by textual layout and comments.

Although others have also been proposed and implemented, **Prolog** {see Clocksin & Mellish (1981)} is generally viewed as the prototypical language for logic programming. It has quickly been rising in popularity as a language for certain classes of AI applications (i.e. language understanding and deductive databases), particularly in Europe. During the last few years it has attracted particular notice through its role as a base language for Japans ambitious "Fifth Generation Computer" project. Its proponents have suggested it as a successor for Lisp. A large number of interpreters (written in Lisp, C, Fortran or Pascal) and compilers are available. Although there is usually a rich library of tools, no programming environments comparable to those of many Lisp implementations have appeared yet.
Prolog uses the Horn clause form of symbolic logic and expresses programs as rules and assertions about some miniworld. Reasoning about this miniworld is implemented by Prolog's deductive machinery; a unification theorem prover which tries to construct any desired solutions as a side effect of proving their existence.



"I think you should be more explicit here in step two."

3. An Experiment in Exploratory Programming.

3.1 Description of Examples

To investigate the merits and drawbacks of exploratory programming and object oriented design, a subset of the features identified as desirable for control systems modelling (chapter 1.3) were implemented in both Lisp and Smalltalk.
Two simple examples were used to test the resulting toolboxes. Although in what follows below we consider **linearization** and **system / subsystem decomposition** as our only applications, a careful selection of primitives and a levelled, modular design ensures that the toolboxes can easily be expanded to serve more ambitious purposes. Our immediate concerns were to explore the suitability of a set of programming tools and design methodologies for control system analysis. The toolbox programs can, however, also be used as stepping stones towards a full-fledged control systems modelling laboratory.

Example 1: This is a simple example taken from one of the exercises in Power & Simpson (1978; p. 41). It defines Goodwin's mathematical model for enzyme synthesis in cells. The system is a third-order, non-linear one, as described by the equations:

$$dx_1 / dt = a / (b + cx_3) - kx_1$$
$$dx_2 / dt = dx_1 - fx_2$$
$$dx_3 / dt = gX_2 - hx_3$$

In these equations $\mathbf{x_1}$ is the amount of messenger RNA in a cell, carrying blueprints from the genes within the cell nucleus to the ribosomes, where the enzyme is actually built up from amino acids; $\mathbf{x_2}$ is the amount of enzyme present; and $\mathbf{x_3}$ is the amount of repressor produced, which controls the rate of production of messenger RNA by repressing genes. All parameters are positive.
We will use this example to derive a linearized set of equations for small deviations from equilibrium.

Example 2: Figure 3.1 shows a block diagram of a fictional control system with hierarchical structure. At the top level there is a system **s1** with single input, the command signal **c**, and a single output, the process output **y**. The system s1 is composed from the following subsystems: a model, a feedforward block and a system s2. The **model** has two state variables, x1 and x2. Its state transition can be described by the differential equations:

$$dx_1 / dt = wx_2$$
$$dx_2 / dt = - wx_1 - 2wx_2 + wc$$

and the output map is described by

$$y = x_1$$

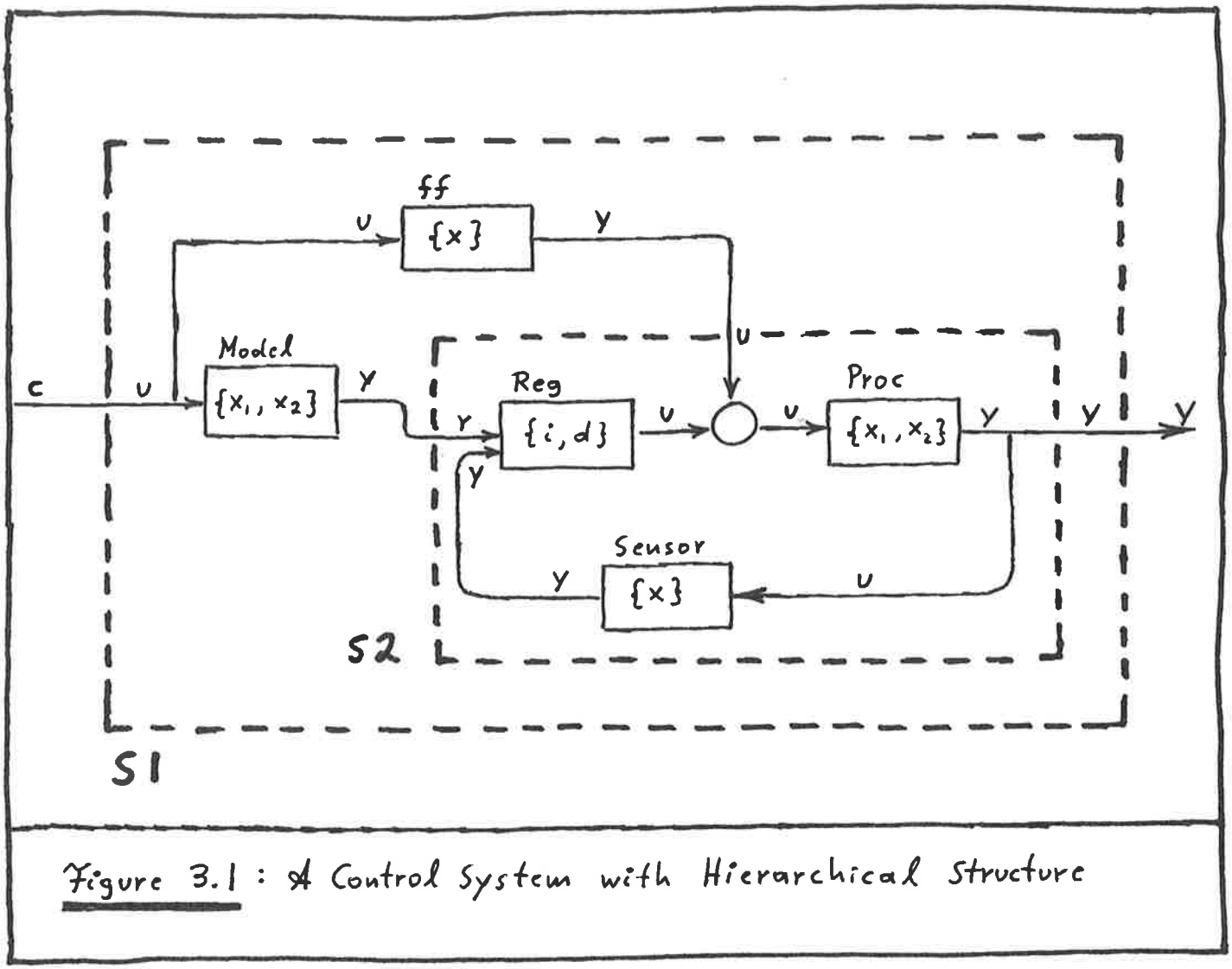


Figure 3.1 : A Control System with Hierarchical Structure

The **feedforward** system has one state variable. The state transition is characterized by the differential equation

$$dx / dt = - ax + (a - b) u$$

Its output map is

$$y = kx + u$$

The system **s2** is composed of three subsystems: regulator, process and sensor. The **regulator** is a PID regulator with two states **i** and **d**. Its state transitions are described by

$$dd / dt = N / T_d (y - d)$$
$$di / dt = 1 / T_i (r - y)$$

and the output map is

$$u = K * (r - y + i + N (d - y))$$

The **process** has two state variables: $\mathbf{x_1}$ and $\mathbf{x_2}$. State transitions are described by the differential equations:

$$dx_1 / dt = - a x_1^{(1/2)} + by$$
$$dx_2 / dt = a x_1^{(1/2)} - bx_2^{(1/2)}.$$

Its output map is given by

$$y = x_2$$

The **sensor** has only one state variable x. Its dynamics is defined by

$$dx / dt = 1 / (u - x)$$

The sensor has a nonlinear characteristic which appears in the output map

$$y = x / (1 + x^2).$$

We will use this example to demonstrate tools for inquiry and system decomposition.

3.2 LISP: Abstraction, Interpretation & Programming Environments

Lisp is the most widely used programming tool for exploratory systems modelling and has a long history as the main language of the AI community. During this time it has been continually refined and many useful utilities have been provided. Figure 3.2 shows the genealogy of the major dialects.

Lisp's interpretative and interactive mode of operation has encouraged its continuing modification. The Lisp culture is a **toolbuilding culture**. Its sparse syntax, while often cited as one of its most annoying features ("**L**ots of **I**rritating, **S**purious **P**arentheses") and its potential for self-reference (programs can write or modify other programs or even themselves) has also aided the quick and easy construction of new tools and languages for specialized applications.
In the classical paradigm building or modifying tools or language compilers is a difficult task, only tackled by highly trained specialists. Here the tool-users were always clients at the mercy of the tool-builders, who had rarely occasion to use their own tools extensively. In the Lisp culture implementation of tools and new languages was comparatively easy. Tool-builder and user were often one and the same person, which lead to a rapid cycle of modifications and refinements motivated by personal experience. Over the years a number of very powerful and user-friendly programming tools and environments were developed in this way, far superior to anything provided by the classical paradigm.
Good examples of the use of Lisp as base language for more specialized programming tools are many so called "AI languages" (i.e. **Planner**, **Conniver**, **CLisp**, **Fuzzy**, **Netl**, **FRL**, **KRL**, **OPS**, **KLONE**, **Act**, ...) which offer sets of specialized features (pattern matching, backtracking, production system interpreters, frames, property inheritance, fuzzy sets, coroutines, message passing, ...) within Lisp's general framework. Charniak et al. (1980) contains a good discussion.

Lisp is built on a simple foundation, with only a few primitive concepts. The strength of the language lies in the fact that these can be combined in very flexible ways, thereby building higher level structures which may grow to be very complicated.
Lisp was designed for symbol manipulation. Its primitive objects are therefore symbols (**atoms**) which may be used to denote concepts or their instances. Symbols are identified by names and may own properties. They can be viewed as "passive" (describing data objects) or "active" (describing procedure objects). Properties store states if they are passive, and behaviour if they are active.
Symbols can be combined to form structures. **Lists** are used as the basic structuring mechanism. Lisp's parentheses indicate conceptual chunking within lists, so that hierarchical (nested) structures can easily be built. The "meaning" of structures is derived by interpretation, there is no syntactic distinction between lists representing states and those representing actions (the same as there is no distinction between a data item and an operation in a v. Neumann-machine).
Every list which is not quoted is interpreted as a request for an action (**function call**). Its first element specifies the name of an operation applied to the objects denoted by the rest of the elements. Some suitable operators for building and dissecting lists, arithmetic and logical operations, modifying the interpretation sequence, defining new operators ... are provided by any Lisp system. While the CONS function is used to construct a list from two single elements, CAR and CDR ("return the first element of a list", "return the rest of the list") are probably the most notorious of the list dissection

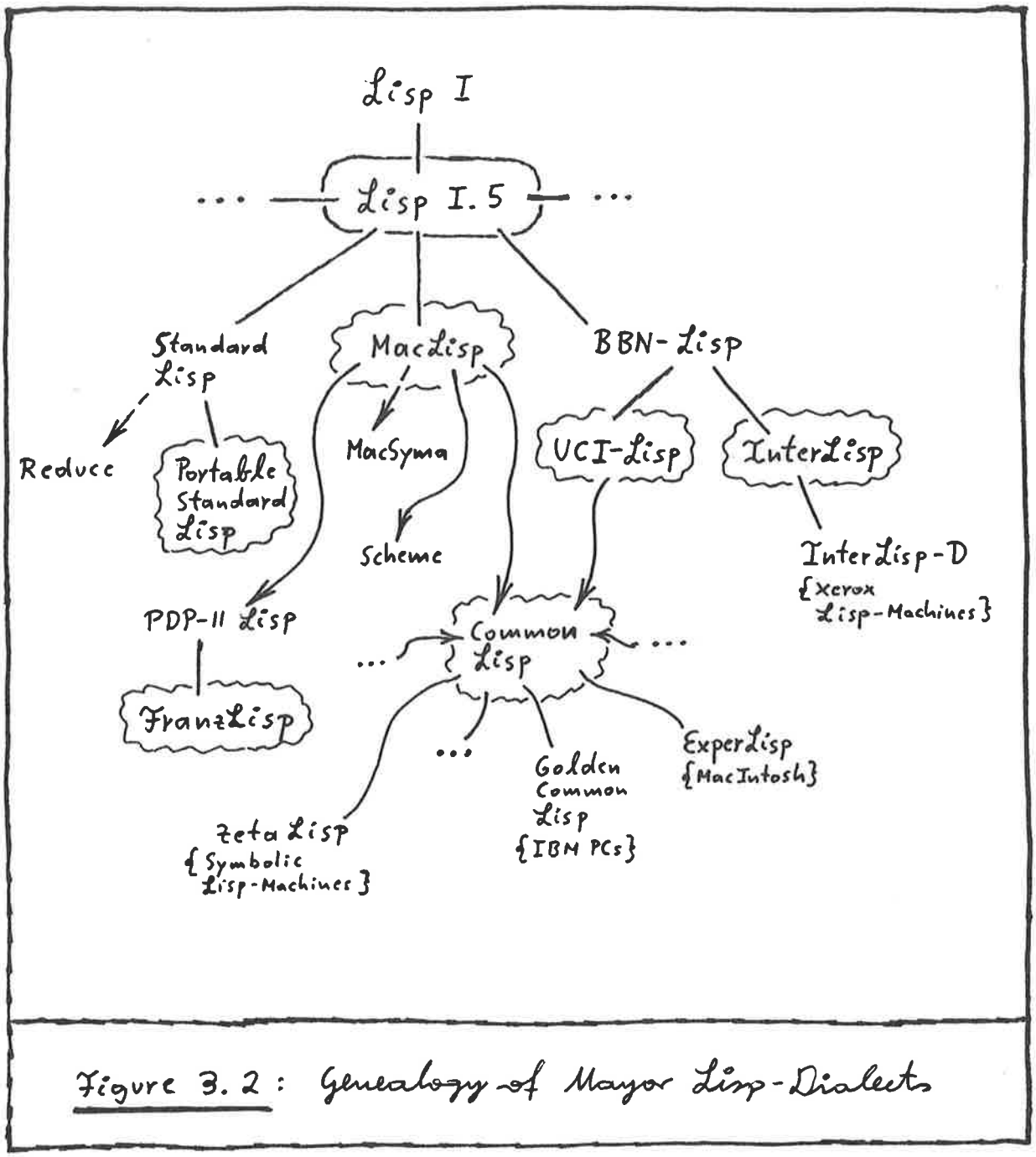


Figure 3.2: Genealogy of Major Lisp-Dialects

functions. The origin of their names reach back to the time of the IBM 7040, the first machine on which Lisp was implemented. CAR and CDR can be combined to form cascades: CADR ("return the first of the rest of this list - i.e. the second element), CDAR, CDDR, CADDR, ... which is a very useful shorthand notation for stepping through a list. Most modern Lisp systems offer a wide range of predefined functions (up to more than 500 for Interlisp). This set of basic functions can easily be extended by the user's own definitions.

To keep the syntax simple and maintain the language's easy extensibility all operations are written in **prefix notation** (so called "Cambridge Polish") and all lists are fully parenthesized (no rules of precedence to worry about). Identifiers can be of arbitrary length and may contain most special symbols, thereby offering ample scope for mnemonic names.

Many Lisps have no variable declarations and use dynamic **scoping**, where symbol bindings are searched for along the chain of function calls. This should be contrasted with classical programming, where declarations are usually mandatory and static scoping is used. Static scoping has advantages over dynamic scoping. Many modern Lisp dialects now support it, offering dynamic scoping as an option (i.e. SPECIAL variables). It is easier to understand, because it finds variable bindings according to the textual nesting of function modules, thereby localizing the search process for the human reader.
Lisp treats all symbols which are not parameters as global (so called "free" variables). Such variables must be used with caution and many Lisp programmers use special naming conventions for them (i.e. *...*). Local variables may be introduced through the LET function.

Execution of Lisp programs is triggered from the so called "**top level**", which operates in a "Read, Interpret, Write" cycle. Interpretation normally proceeds left to right, inside out, with a value (possibly nil) returned by each operation. There are functions for conditional selection of alternative execution paths and for specification of repetition. **Recursion** is Lisp's traditional main control structure. It is a very powerful instrument to define and process quite complex regular structures in a concise way. So called tail recursion (". . . and then do the same thing to the rest of this structure") is particularly common in list processing programs.
There are also a number of so called mapping functions, which repetetively apply a specified operation to a list of values (i.e. MAPCAR, ...).

All modern Lisp systems now offer a rich repertoire of **control structures**. Various forms of selection (i.e. IF) and iteration (i.e. DO, WHILE, DOLIST, . . .) are supported. Side effects (printing, binding values explicitely through SETQ) are also commonly used, although not part of "pure" Lisp (but purely applicable languages are unfortunately poorly applicable).

Finally a great number of other useful but often complicated features (various forms of macros, closures, exception handling, continuation passing, file handling ...) have been added to some dialects. Great care must be taken to use them in a disciplined fashion. Otherwise programs can quickly become tangled and opaque.

Good programming style is therefore extremely important. The temptation to "hack out" programs in an unsystematic fashion is possibly greater than in classical programming, where the necessity to adhere to type declarations enforced by the compiler at least

prohibits many "clever" programming tricks by undisciplined programmer. Lisp has no such restrictions; "everything goes".

Fortunately the concepts of symbols and symbol structures allows us to use data and procedure abstraction in a methodology of levelled design. Abelson et al. (1985) and Allan (1978) are excellent demonstrations of the power of this approach.

Apart from its excellent support of interactive programming, the main strength of Lisp lies in its flexibility and freedom from arbitrary restrictions.
This requires that the system has to defer binding structures to memory representations and machine code, but rather interprets and keeps structures "dynamic" at run time, a convenience we have to pay for in terms of memory and processing resources. Lisp systems have therefore a reputation of being "slow", burdened by garbage collection and chasing long chains of indirect references. While this was to a certain extend true in the past, it does often not hold anymore. New implementation and hardware architectures have removed much of this overhead. Even arithmetic, always cited as one of the language's weaker points has grown to become excellent in many modern Lisp systems.

Figure 3.3 shows an overview of the Lisp functions in our first toolbox. We have used Experlisp, a MacLisp / Common Lisp dialect which runs on the Apple MacIntosh and offers a pleasant interactive environment as well as a wide collection of predefined functions. It uses static scoping by default (SPECIAL variables are dynamically scoped). The primitive functions seem to be implemented in a surprisingly efficient manner.

The **toolbox** is organized according to the princples of levelled design which were advertised above. Great care was taken to select appropriate primitives, which were organized around a few data types. In anticipation of the Smalltalk implementation they were also arranged in a manner which makes their clustering into object classes as straightforward as possible.

Currently the supported **data types** are: **System** (implemented by using ExperLisp's "structure" package - simulating record definitions a la Pascal), **ConnectionTable** (implemented as an association list), **BehaviourList** (representing a description of system behaviour by differential equations - implemented as a list of lists of expressions), **Expression** (implemented as nested lists of operators and operands), **BehaviourTable** (representing a description of system behaviour by matrices - implemented as a list of coefficient matrices), and**CoefficientMatrix** (implemented as lists of lists of coefficients).
All **operations** are implemented as Lisp functions associated with these data types and grouped into a number of categories. **Constructor** functions build instances of a data type, while **destructor** functions are used to remove them. **Simplification** functions may additionally simplify an object's description while building a new instance of it. **Transformer** functions can change an objects representation. **Query** functions are predicates, which inquire about the state of an object. **Selector** functions must be used to access an objects components, while **display** functions will show a nicely formatted screen or printed representation.

Figure 3.4 shows an application of this toolbox, using the two examples described in chapter 3.1. The Lisp representation of these examples is contained in figure 3.5.

Figure 3.3: The **Lisp Toolbox** - An Overview

```
==========================================================
=                  First Level: "Primitives"             =
==========================================================
```

SYSTEM (implemented as a structure with the following slots:

Name	{a symbol}
Inputs	{a list of symbols}
States	{a list of symbols}
Outputs	{a list of symbols}
Subsystems	{a list of symbols}
Behaviour	{a BehaviourList})

constructor & destructor functions:

MakeSystem (aName anInputList aListOfStateVars anOutputList
aListOfSubsystems
aListOfStateTansitionFns aListOfOutputFns)
NewName (aSystem aName)
AddInputs (aSystem anInputList) **ClearInputs** (aSystem)
AddOutputs (aSystem anOutputList) **ClearOutputs** (aSystem)
AddStates (aSystem aStateList) **ClearStates** (aSystem)
AddSubsystems (aSystem aSubsystemList)
ClearSubsystems (aSystem)
NewBehaviour (aSystem aStateEqnList anOutputEqnList)
ClearBehaviour (aSystem)

query & selector functions:

Name?	(aSystem)	**Name**	(aSystem)
Inputs?	(aSystem)	**Inputs**	(aSystem)
Outputs?	(aSystem)	**Outputs**	(aSystem)
States?	(aSystem)	**States**	(aSystem)
Subsystems?	(aSystem)	**Subsystems**	(aSystem)
Connections?	(aSystem)	**Connections**	(aSystem)
Behaviour?	(aSystem)	**Behaviour**	(aSystem)

display functions:

ShowSystem (aSystem)

CONNECTIONTABLE (implemented as a list of binary connections
(lists of pairs))

constructor functions:

TableOfConnections (aSystem)
MakeBinaryConnection (aSource aDest)
MakeConnectionTable (anyNumberOfBinaryConnections)

query & selector functions:

Connections? (aConnectionTable)
FirstConnection (aConnectionTable)
RestOfConnections (aConnectionTable)

display functions:

ShowConnectionTable (aConnectionTable) {PrintPairs (aListOfPairs)}

BEHAVIOURLIST (implemented as a list of a list of
state equations (expressions) and a list of
output equations (expressions))

constructor functions:

MakeBehaviourList (aStateEqnList anOutputEqnList)
AddStateEqns (aBehaviourList aStateEqnList)
AddOutputEqns (aBehaviourList anOutputEqnList)

query & selector functions:

StateEqns? (aBehaviourList) **StateEqns** (aBehaviourList)
OutputEqns? (aBehaviourList) **OutputEqns** (aBehaviourList)
FirstEqn (anEqnList)
RestOfEqns (anEqnList)

display functions:

ShowBehaviourList (aBehaviourList)

EXPRESSION (implemented as a nested list of operators & operands
(prefix or infix).
Must always be fully parenthesized !)

constructor & simplification functions:

MakePrefixBinary (anOperator aFirstArg aSecondArg)
MakeInfixBinary (aFirstArg anOperator aSecondArg)
MakeUnary (aSign anArgument)

MakePrefixSum (aFirstArg aSecondArg)
MakePrefixDifference (aFirstArg aSecondArg)
MakePrefixProduct (aMultiplicand aMultiplier)
MalePrefixQuotient (aDividend aDivisor)
MakePrefixPower (aBase anExponent)

query & selector functions:

Constant? (anExpression) **UnaryOperator** (anExpression)
Variable? (anExpression) **UnaryArg** (anExpression)
SameVar? (firstExpression secondExpression)
Unary? (anExpression)

PrefixSum? (anExpression) **PrefixOperator** (anExpression)
PrefixDifference? (anExpression) **PrefixFirstArg** (anExpression)
PrefixProduct? (anExpression) **PrefixSecondArg** (anExpression)
PrefixQuotient? (anExpression)
PrefixPower? (anExpression) **InfixOperator** (anExpression)
InfixFirstArg (anExpression)
InfixSecondArg (anExpression)

transformer functions:

InfixToPrefix (anExpression) {Flat? (aList)}
PrefixToInfix (anExpression)
Derive (aPrefixExpression aVar) {DeriveAux (anInfixExpression aVar}

display functions:

ShowExpression (anExpression)

BEHAVIOURTABLE (implemented as a list of system-matrix,
input-distribution-matrix and
measurement-matrix
(all implemented as CoefficientMatrices))

constructor functions:

MakeBehaviourTable (aSystemMatrix anInputDistMatrix aMeasureMatrix)
AddMatrix (aBehaviourTable aCoefficientMatrix)

query & selector functions:

SystemMatrix? (aBehaviourTable)
SystemMatrix (aBehaviourTable)
InputDistMatrix? (aBehaviourTable)
InputDistMatrix (aBehaviourTable)
MeasurementMatrix? (aBehaviourTable)
MeasurementMatrix (aBehaviourTable)

display functions:

ShowBehaviourTable (aBehaviourTable)

COEFFICIENTMATRIX (implemented as a list of rows
(lists of coefficients (lists)))

constructor functions:

MakeCoefficientMatrix (anyNumberOfRowLists)
AddRow (aCoefficientMatrix aRowList)

MakeRow (anyNumberOfCoefficients)
AddCoefficient (aRowList aCoefficient)

query & selector functions

EmptyMatrix? (aCoefficientMatrix)

FirstRow (aCoefficientMatrix)
RestOfRows (aCoefficientMatrix)

FirstInRow (aRowList)
RestInRow (aRowList)

display functions:

ShowCoeffMatrix (aCoefficientMatrix)
ShowRow (aCoefficientList)

==========
= **Next Level: "Applications"** =
==========

; print or obtain information about subsystems
; and their hierarchical structure

ShowSubsystemHierarchy (aSystem aTabLevel)
{ShowSubsystemHierarchyAux (aSystemList someSpaces aTabLevel)
PrintSpaces (someSpaces) }

AllSubsystems (aSystem)
GetAllSubsystems (aListOfSystems) {RemoveDuplicatesFrom (aList)}

SubsystemHierarchy (aSystem)
GetSubsystemHierarchy (aListOfSystems)

; obtain information about systems' state variables

AllStates (aSystem)
GetAllStates (aListOfSystems)

; linearize a system's behaviour description
; (i.e. produce a linearized set of matrices (system, input-dist.,
; measurement) by differentiation of its state and output equations).

Linearize (aSystem)

Further Extensions (**not implemented**; will require additional primitive data types & functions)

; obtain "topological" information

ContainedIn (aSystem)
Cycle? (aSystem)
ConnectedTo (aSystem)

Path? (aSource aDest) **AnyPath** (aSource aDest)
AllPaths (aSource aDest)
ShortestPath (aSource aDest)

; trace "signal flows" through a system

Simulate (aSystem aTerminationCondition)
Monitor (aVariable)

; allow "recursive definitions"

Replicate (aSystem aTerminationCondition)

; Figure 3.4: ===== Trace of an Example Session =====

```
; (user inputs are in bold; the "structures" package, "systems" package
;  and "eamples" package has ben loaded and compiled !)
```

; **example 1:** Linearization & ...

(States? ES)
;t
(States ES)
;(x1 x2 x3)
(Behaviour ES)
;((((a / (b + (c * x3))) - (k * x1))((d * x1) - (f * x2))((g * x2) - (h * x3)))
nil)
(ShowBehaviourList (Behaviour ES))
;State-Eqns
;----------
;((a / (b + (c * x3))) - (k * x1))
;((d * x1) - (f * x2))
;((g * x2) - (h * x3))
;+++ no output equations defined +++
;nil
(ShowSystem ES)
;==========
; SYSTEM: ES
;==========
; Inputs : nil
; Outputs : nil
; States : (x1 x2 x3)
;
; *** Connections ***
;   ++++++++++
;
;
; *** Behaviour ***
;   +++++++++
;State-Eqns
;----------
;((a / (b + (c * x3))) - (k * x1))
;((d * x1) - (f * x2))
;((g * x2) - (h * x3))
;+++ no output equations defined +++
;nil

```
(PrefixToInfix '(+ (+ a (* b x))(* 3 (** x 3))))
;((a + (b * x)) + (3  * (x ** 3 )))
(InfixToPrefix '((a + (b * x)) + (3  * (x ** 3 ))))
;(+ (+ a (* b x))(* 3  (** x 3 )))
(Derive '((a + (b * x)) + (3  * (x ** 3 ))) 'x)
;(b + (3  * (3  * (x ** 2 ))))
(Linearize ES)
;((((- k) 0  (a * (- (((b + (c * x3)) ** -2 ) * c))))(d (- f) 0 )(0  g (- h)))(nil nil
nil) nil)
(ShowBehaviourTable (Linearize ES))
;System - Matrix
;--------------
;(- k) 0  (a * (- (((b + (c * x3)) ** -2 ) * c)))
;d (- f) 0
;0  g (- h)
;Input Distribution - Matrix
;--------------------------
;No coefficients in this row
;No coefficients in this row
;No coefficients in this row
;Measurement - Matrix
;-------------------
;nil
(ShowCoeffMatrix (SystemMatrix (Linearize ES)))
;(- k) 0  (a * (- (((b + (c * x3)) ** -2 ) * c)))
;d (- f) 0
;0  g (- h)
;nil
```

; example 2: System Decomposition & ...

(ShowSystem s1)

```
;=============================
; SYSTEM: S1
;=============================
; Inputs  : (c)
; Outputs : (y)
; States  : nil
;
; *** Subsystems ***
;    ++++++++++
;(s2 model ff)
;
; *** Connections ***
;    ++++++++++++
;(c S1)
;(S1 y)
;
;
; *** Behaviour ***
;    +++++++++
;+++ no state equations defined +++
;+++ no output equations defined +++
;nil
```

(TableOfConnections proc)

```
;((u proc)(proc y))
```

(ShowConnectionTable (TableOfConnections proc))

```
;(u proc)
;(proc y)
;
;nil
```

(StateEqns (Behaviour proc))

```
;(((- (a * (x1 ** (1  / 2 )))) + (b * u))((a * (x1 ** (1  / 2 ))) - (a * (x2 ** (1
/ 2 )))))
```

(OutputEqns (Behaviour proc))

```
;((x2))
```

```
(ShowBehaviourList (Behaviour proc))
;State-Eqns
;----------
;((- (a * (x1 ** (1  / 2 )))) + (b * u))
;((a * (x1 ** (1  / 2 ))) - (a * (x2 ** (1  / 2 ))))
;Output-Eqns
;-----------
;(x2)
;nil
(ShowBehaviourTable (Linearize proc))
;System - Matrix
;---------------
;(a * ((1  / 2 ) * (x1 ** ((1  / 2 ) - 1 )))) 0
;(a * ((1  / 2 ) * (x1 ** ((1  / 2 ) - 1 )))) (- (a * ((1  / 2 ) * (x2 ** ((1  / 2 )
- 1 )))))
;Input Distribution - Matrix
;--------------------------
;b
;0
;Measurement - Matrix
;-------------------
;0  1
;nil
(MeasurementMatrix (Linearize proc))
;((0  1 ))
(SystemMatrix (Linearize ff))
;((b))
(InputDistMatrix (Linearize sensor))
;(((t ** -1 )))

(AllStates 'model)
;((x1 x2) nil nil nil)
(GetAllStates '(reg proc sensor))
;((i d) nil ((x1 x2) nil ((x) nil nil)))
(AllSubsystems s1)
;(s2 model ff reg proc sensor)
(AllSubsystems s2)
;(reg proc sensor)
(GetAllSubsystems '(s1 s2))
;(s2 model ff reg proc sensor)
(GetSubsystemHlerarchy '(s1))
;(S1 (s2 model ff) s2 (reg proc sensor) reg nil proc nil sensor nil model nil
ff nil)
```

```
(ShowSubsystemHierarchy s1 7)
;++++++++++++++++++++++++++++++++++++++++
;  Subsyststems of: S1
;++++++++++++++++++++++++++++++++++++++++
;
;s2
;
;       reg
;
;       proc
;
;       sensor
;
;model
;
;ff
;
;nil
```

; Figure 3.5:

; ========= EXAMPLES of simple control systems ==========

; Example #1:

```
(SETQ ES (MakeSystem 'ES '() '(x1 x2 x3) '()  nil
                '( ((a / (b + (c * x3))) - (k * x1))
                   ((d * x1) - (f * x2))
                   ((g * x2) - (h * x3)) )
                '() ))
```

; Example #2:

```
(SETQ S1 (MakeSystem 's1  '(c)  nil '(y)
                '(s2 model ff)
                '() '() ))

(SETQ S2 (MakeSystem 's2  '(r u) nil '(y)
                '(reg proc sensor)
                '() '() ))

(SETQ Model  (MakeSystem 'model  '(c) '(x1 x2) '(y)   nil
     '( (w * x2) (((- (w * x1)) - (2 * (w * x2))) + (w * c)) )
     '( (x2) ) ))

(SETQ FF     (MakeSystem 'ff     '(u)  '(x)     '(y)  nil
     '( ((- (B * x)) + ((a - b) * u)) )
     '( (k * (x + u)) ) ))

(SETQ Reg    (MakeSystem 'reg    '(r y) '(i d)  '(u)  nil
     '( ((1 / Ti) * (r - y)) ((N /Td) * (d + y)) )
     '( (K * (((r - y) + i) + (N * (d - y)))) ) ))

(SETQ Proc   (MakeSystem 'proc   '(u)   '(x1 x2) '(y)  nil
     '( ((- (a * (x1 ** (1 / 2)))) + (b * u))
        ((a * (x1 ** (1 / 2))) - (a * (x2 ** (1 / 2)))) )
     '( (x2) ) ))

(SETQ Sensor (MakeSystem 'sensor '(u)   '(x)     '(y)  nil
     '( ((u - x) / T) )
     '( (x / (1 + (x * x))) ) ))
```

3.3 Smalltalk: Encapsulation, Message Passing & Property Inheritance

The Smalltalk-80 system, as described by Goldberg and Robson (1983), consists of two major components: an object-oriented programming language and a window & mouse-based programming environment. Similar to Lisp it is designed to support an exploratory programming style. Its excellent tools for graphical animations can be profitably employed by simulation models (i.e. Goldberg & Robson (1979)).
Figure 3.6 sketches its historical development and its relationships to other object oriented programming systems.

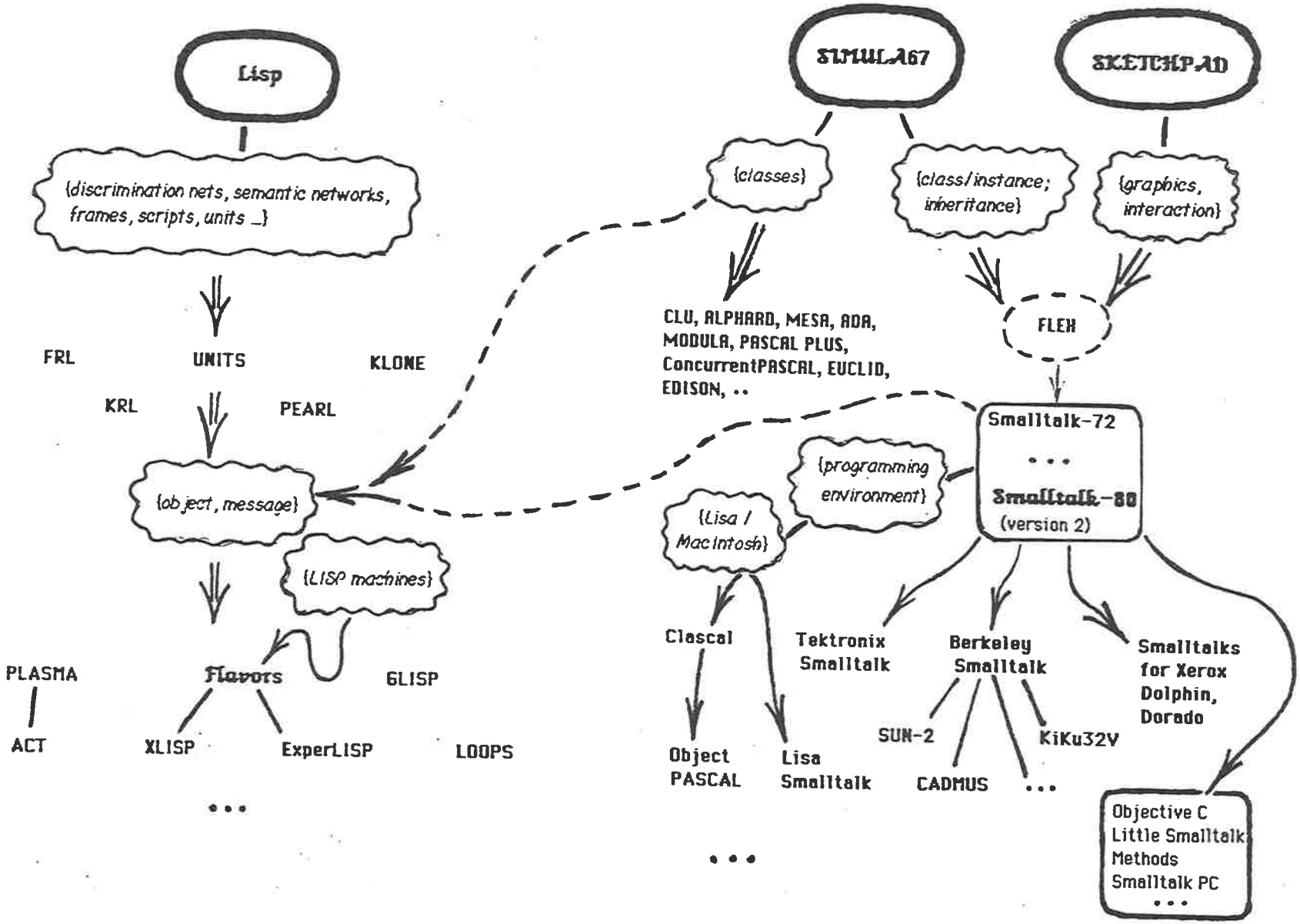
Smalltalk needs single user workstations with sufficient processing power and memory (at least one megabyte) to support its model of computation at a satisfactory level of performance. A memory-mapped graphics screen and a pointing device are required for the user interface.

In Smalltalk **every data structure is viewed as an object**, described by an appropriate class definition. Objects may have properties which are strictly private and may respond to any number of so called messages. Message passing between objects is the only way in which computations take place.

A Smalltalk program consists of class definitions. A **class** has a name and a superclass and may own any number of so called class and instance variables and class and instance methods. **Class variables & methods** are shared among all members of a class, while **instance variables & methods** are private to individual objects .
All **messages** an object will respond to form its so called **protocol**. Class definitions associate methods (implementations) with these messages. **Method definitions** may also be grouped into chapters (i.e. initialization methods, accessing methods, private methods), but this is transparent to the Smalltalk interpreter.

Please note that Smalltalk, like Lisp, is a typeless language. The types of objects should only be deducible from their names. This is not really a disadvantage, since in interpreted languages like Lisp, Prolog or Smalltalk the notion of "strong typing" does not buy anything for the user. Since programs in these languages are normally developed in a highly interactive fashion, type (and other) errors can easily be detected and corrected "on the fly"; powerful tools for doing this are part of the programming environment. This is different from compiled languages, where the increases in program reliability and efficiency achievable through strong typing are very valuable.

Smalltalk's syntax is somewhat unusual, but very consistent, flexible and expressive. New data structures (classes), operators (methods) and even control structures (methodss) may be defined very elegantly.
The language assumes that all global variables start with an upper case letter, whereas all locals start with lower case. Local variable declarations are enclosed in vertical bars and the left-pointing arrow is used as an assignment symbol. Square brackets delimit blocks, whose evaluation can be deferred. As in Lisp, all Smalltalk messages return some value, possibly the receiver itself. An up-arrow can be used to denote explicitely what should be returned.
Where necessary **Messages** can be separated by periods, or cascaded by semicolons (i.e. "Pen turn: 30; down; go: 10; up" defines a drawing sequence of 4



A Family Tree of Object-Oriented Programming Systems : Figure 3.6

cascaded messages). Messages can be classed as binary or keyword messages. Binary messages are written in infix notation. "2 + 1", for example is interpreted in the following way: message "+" with argument 1 is sent to object 2 (an instance of class "number"); numbers know "+" to denote addition, a process which is performed using their own value as the first and the argument as the second operand; the result (hopefully 3) is then returned. Keyword messages can either be unary (i.e. "Pen up"), or they may take any number of arguments. The keywords are normally used to describe the arguments' function, while the names of the arguments themselves indicate the expected types of their values (i.e. "Circle showWithRadius: aNumber FilledWith: aPattern at: aPoint").

"Self" and "super" are so called **pseudo-variables**. They allow references from the current object to itself and its superclass. **Self** is sometimes needed for triggering private messages, while **super** may be used to obtain the original of some method overridden by a new definition in the subclass.

The Smalltalk system consists of a large number of predefined classes, the so called "virtual image" and a virtual machine responsible for implementing basic primitives, memory management and command interpretation.
Smalltalk is a very "soft" system, there are very few unchangeable primitives. The user can browse and modify all the class definitions in the virtual image, which can both be blessing and curse. On the positive side one gains an almost extreme degree of flexibility to tailor one's environment to one's wishes. If you don't like a particular feature you can always change it ! On the other hand, of course, you have to know what you are doing, because it is quite easy to crash the system by unfortunate modifications (i.e. try to change the definition of "+" in class "numbers").
This is, however, not as bad as it may sound. Due to the object oriented nature of the programming environment changes usually are very localized and can be traced easily. One learns quickly to keep certain system classes alone, unless one knows exactly what one is doing.

The flexibility to modify almost all classes in the system is very effectively exploited by the typical style of good Smalltalk programmers. Writing a Smalltalk program should not consist in accumulation of large bodies of new code, but rather in skillful modification of existing definitions. There is a story about a programmer at Xerox PARC who over a long period of time used his workstation without any significant handicap to his productivity after his keyboard developed a malfunction.

Smalltalk's **user interface** is probably the most advanced integrated computing environment which has successfully been implemented.
Its use can be well described by employing the so called "desktop" metaphor.
This paradigm views the terminal screen as a large desk, on which documents and tools of different kinds are arranged; beside or overlapping each other, in different stages of execution. In Smalltalk each of these tools and documents is associated with a window or a symbol which can be expanded to a window. Such windows can overlap and cover all or parts of other windows. They can be opened, activated and closed. Each of them may represent a view into a process, which can be active, terminated or suspended. Smalltalk supports conceptual multiprocessing (coroutines) through time slicing. Suspended processes are reactivated as soon as a window with which they are associated is selected. The resident process can now be controlled through typed commands or menu selection (i.e. browsing class definitions, editing text, writing programs, executing programs, viewing an animation, drawing pictures, ...).

All processes can be temporarily suspended and reactivated at a later time. Program development is truly "exploratory" in the sense used above; i.e. different modules of a program can be quickly designed, tested, modified and integrated in an interleaved fashion .

In order to write a Smalltalk program a workspace and browser window should be visible. **Workspaces** allow the specification and evaluation of expressions which need not be permanently kept. **Browsers** can be used to show and manipulate (create, redefine, delete, pretty print, file, ..) messages of a specified method category within a specified class of a specified class category. Writing or modifying messages takes place within the lower pane of the browser window, using a syntax-driven editor. Menu commands are available for formatting and compilation of method definitions. Syntax and other simple errors are reported here. Spelling correction is also offered. More complex error conditions may still occur at evaluation time, where they can be traced through so called notifier and debugger windows.

Smalltalk's standard user interaction takes place through **menus** and mouse action. Normally a three-button mouse is used or simulated, which can select information ("red" button) or bring up two different kinds of menus, the contents of which may depend on the current context. One of these menus ("blue" button) lets you manipulate windows themselves (collapse, close, move, resize, ...), while the other one ("yellow" button) specifies operations on their contents (create browser, create workspace, quit, ... cut, paste, undo, format, compile, execute, print, ...).

The mouse is moved into a menu field and one of its buttons is pushed. Selected commands and windows are shown in reverse video (white on black). Active windows have "scroll bars", to indicate and modify the position of the current view within a selected document.

Inspectors are another useful class of processes. They will show a view of a selected objects with all its "property / value" bindings and permit expression evaluation within that context.

Apart from browsers, inspectors and workspaces Smalltalk offers a large number of powerful predefined error tracing, performance engineering, graphics editing and project management tools whose flavour can only be conveyed by using a Smalltalk system.

Our **Smalltalk toolbox** implements only a subset of the features provided in the Lisp implementation; i.e. only system / subsystem decomposition is curently supported. The main reason for this restriction lies in the nature of the experimental Apple MacIntosh implementation of Smalltalk used for this experiment, which offers very little space for user defined classes. This system is a subset of Apple's LISA Smalltalk. For our purposes it worked quite well, with only little apparent speed disadvantage over the LISA implementation. The fact that there is not much user space available should not be held against it. It is perfectly adequate for demonstrating and teaching basic principles. Since Smalltalk is an inherently memory-hungry system it is an remarkeable achievement to squeeze it into a 512K MacIntosh at all.

This implementation can, of course, not be used for building a full-fledged control system laboratory. For this a workstation of at least the power of the Tektronix 4404 would be required (or possibly a future "huge Mac" version of this software).

Figure 3.7 summarizes of the Smalltalk toolbox, while figure 3.8 shows a snapshot of its use.

Figure 3.7: **Summary of Smalltalk-toolbox**

Object class DEMO

• no class or instance variables

message protocol:

--- class methods for instantiation ---

new
workspace

--- instance methods for system creation ---

ff
model
proc
reg
sensor
s1
s2

Object class System

• no class variables
• instance variables: **input output state subsystems connections**

message protocol:

--- class methods for instantiation ---

new

--- instance methods for initialization ---

initialize

--- constructor & destructor methods ---

addConnectionFrom: aSource **to**: aDest
addInput: aSymbol **addState**: aSymbol
addOutput: aSymbol **addSubsystem**: aSymbol

removeAll **removeFirstSystem**

--- selector methods ---

getAllConnections **getAllSubsystems**
getInput **getState**
getOutput

--- query & selector methods ---

hasConnections **hasState**
hasInput **hasSubsystems**
hasOutput

getSystems **getToplevelStates**

--- internal methods ---

expand: aCollection

Object class SystemCollection

- no class variables
- instance variables: components

message protocol:

--- class methods for instantiation ---

new

--- instance methods for initialization ---

initialize

--- constructor & destructor methods ---

addSystem: aSystem

removeAll **removeFirstSystem**

--- Selector methods ---

getAllSystemHierarchy
getFirstSystem
getSystems
getTopLevelStates

Ch. 4

4. Conclusions: How to constrain Complexity.

Current trends in price / performance ratios of personal workstations make powerful interactive programming environments for control system modelling laboratories an increasingly attractive proposition. Many intellectually challenges which could not be adequately tackled in the past because of inadequate tools may now come within our reach.

Model implementation is just one among a range of activities we must perform during a modelling project. Others include system identification, model specification and documentation, data analysis, model validation and experimentation. Currently we use different notations and tools for all of these. It would be much more convenient to have a single unified framework {i.e. see Oren & Zeigler (1979)} to work in. Such a system should permit us to view different processes in different stages of execution (i.e. a graphical model, textual documentation, views into the source program's execution, various performance measures, ...). Each of these should reside in its own separate window. A mouse may be used to select and activate processes, with control options provided by pop-up menus. These may start an analysis or simulation, reset it to a previous state, edit models and graphic scenarios, trace objects' states, perform statistical analyses, ...

Building such an integrated modelling environment is a very ambitious undertaking. It will require invariably require a large amount of programming effort. In many ways it will be a complex experiment in itself, with many possible design options to be chosen from. Before we committ ourselves to particular strategies we will want to be sure of their effectiveness.
Experimentation with simple prototypes in terms of expressive power, efficiency and user acceptance becomes therefore essential and a range of appropriate tools for their rapid development is required.

Classical techniques have in the past proved too tedious and inflexible for this task. Exploratory tools embedded in Smalltalk-like programming environments may well prove a major breakthrough. We should be free to specify, design, implement and experiment with a prototype model; alternating between different activities at will. This is typical for much of human problem solving. Several tasks may be relevant to a particular project and they need to be interleaved at will; otherwise the creative process may be inhibited.

Programming environment supporting these ideas may well increase both productivity and quality of the modelling process by an order of magnitude.

This experiment has shown that **Lisp**, an exploratory programming style, and a methodology of design by levels of abstraction built upon carefully chosen primitives could easily cope with the modelling task and seems to be very well suited for rapid prototyping.

The author of this study has had extensive experience with "classical" programming languages (i.e. **Pascal**, **Dynamo**, **GPSS**, **Simscript**, **Simula**; see also Kreutzer (1986)) in modelling projects. Progress in building the Lisp-based toolbox was in many ways faster and much more enjoyable. It probably also resulted in a "better" (more "structured") product. Much of this can be credited to to the interactive nature of Lisp and the associated freedom to change one's mind in mid-stream. In classical programming styles one often shies away from make any fundamental changes once a significant part of the coding has been done, because one dreads the necessity to start

from scratch again. This tends to lead to ad hoc patches which diminish the consistency and harmony of the end product.
Debugging was also much easier, even though the ExperLisp implementation on the MacIntosh provided only rudimentary features when compared with many other modern Lisp systems. Just the possibility to copy and evaluate expressions interactively was invaluable. This is of course much more tedious in the classical "edit / compile / link / execute" cycle. Matching parentheses, a recurrent nuisance in some Lisp systems, turned out to be no problem at all, since the editor blinked matching pairs automatically.
Part of the credit should also go to the mouse-based editing on the MacIntosh. In this environment it was very easy to make repetetive modifications in the middle of some function, evaluate it and watch the results in a different window.

The only misgivings about the ExperLisp systems concern its lack of debugging aids and object oriented programming features as well as the fact that some of the more esoteric functions did "not quite" work as advertised. As always the remedy is promised for the next release; "real soon now".

The **Smalltalk** environment is much more sophisticated than the ExperLisp one. Since only a small number of features were implemented, due to reasons explained above, only tentative conclusions can be drawn at this stage. It would certainly be worthwhile to extend the experiment to a genuine Smalltalk workstation.

What can be said is that the Smalltalk programming metaphor supports object oriented design well, by enforcing structuring and the definition of small functions (methods). The programming environment is extremely pleasant to use and supports thisprogramming style through the browser and the syntax directed editing facility. It was very easy to quickly define a large number of methods, particularly when, as is quite common, they did not differ all that much from one another. Property inheritance is also very helpful here.

Smalltalks syntax was easily assimilated and felt "right" after some acclimatization. It turned out to be highly orthogonal and lent itself to program in a very "self documented" fashion.
The system contains a multitude of useful predefined data structures (classes) and operations (methods). Not all were implemented in this version and only very few were required for the toolbox.
Syntax checks seem to be thorough and the spelling correction and error tracing features are seem useful if fully exploited. This wasnot the case here; in this implementation you could actually crash the system (out of memory) with too complicated error traces.
The inspection facility turned out to be particularly valuable if an error condition occurred. The speed of the system was just adequate for the simple tasks which were implemented, although one could easily get ahead in typing. This reflects the fact that a "minimal" performance Smalltalk system was used.
As for the Lisp system, the possibility to leave a task (window), perform a different action in another window (i.e. testing, error tracing, or making a modification of another method of the same class or one of its sub- or superclasses) and return to the original task (window) turned out to be essential for this style of programming.

Object oriented programming itself feels very natural, once one grows accustomed to think in terms of "programming as simulation of a real-world system". It proceedes rather like building a model of some task domain, identifying the relevant objects, classifying these according to similarities and differences, and defining a language of interaction among them. The relevant objects can then be mapped into class / subclass definitions, according to their similarities and the properties they may inherit from or provide to other classes. The language of interaction defines all the messages a class of objects should understand, which can then be implemented as Smalltalk methods.

Apart from these observations some theoretical reflections on object orientation and its effects on a models complexity may be of value.

The notion of programming as model building leads to a close correspondence between objects, relationships, actions and processes of some relevant aspect of reality and its formal representation. The resulting encapsulation of state descriptions and procedures into object structures yields highly modular programs. This methodology works best for separable systems and if their "proper" decomposition yields classes of systems with strongly interacting components. Instances of these classes in turn should be only loosely coupled to other instances of their own or other classes. The fact that all references to an object's state occur through methods encapsulated within the object itself leads to robust programs. Objects may defend themselves against invalid inputs (messages).
In this sense there is a correspondence to the idea of "abstract data types" and other abstraction techniques provided by some modern classical programming languages (**Edison**, **Modula**, **Ada**, **CLU**, **Alphard**, **Mesa**, ...). This is not surprising, since all these developments can ultimately be traced back to **Simula**'s class concept. The basic objectives are quite different, however.

Object oriented programming is dominated by the notion of a close correspondencs between programs and models of some task domain. Object descriptions are encapsulated into textually closed modules because they describe logically related aspects of a model. Their message protocols define new levels of abstractions at which one can talk about a system in categories which should be natural for the specified class of applications. Instances of these objects are created by computational processes, which are essentially simulations of the candidate system.
New levels of abstraction are of course also created by using languages catering solely for data encapsulation. The emphasis, however, is on program reliability here, following the principle of "information hiding". Modules should posses no more knowledge about their environment than is absolutely necessary for their task. The ability to defend against invalid access and incorrect information is the important aspect, not the modelling idea. Languages supporting this kind of object encapsulation often bind object instances already at compile time, for reasons of efficiency. Packages and modules in Ada or Modula can not be dynamically created by some computational process. Their primary purpose lies in improving program reliability, not model structuring.
The idea of programming by analogy through differential descriptions and property inheritance is therefore not relevant here. However, it occupies a central role in object oriented programming frameworks. Here it reduces the effort of model description by classing new objects as special cases of already known ones, stating only their differences explicitely.

Both programming techniques ensure that only local changes (to a class' or method's implementation) are necessary if changes in a system's performance are required. Changes may, of course, be somewhat more extensive if the system's functionality must be changed.
This locality improves both understandability and ease of maintenance of programs, and it encourages independent development and testing of modules.

Names can be local within the context of a class. General naming conventions are unnecessary apart from a few globally visible structures (like class names, ...). This seemingly trivial property actually turns out to be extremely convenient . Even for the small Lisp toolbox you will have noted that names tend to be very long. One reason for this is that function descriptions must always be concatenated with the name of the data structures they manipulate, so that we arrive at names like: ShowSystem, ShowConnectionTable, ShowExpression, ShowCoefficientmatrix, ShowBehaviourList, ShowBehaviourTable. In large programs this quickly leads to "name pollution" and the strong temptation to make do with unambiguous and ad hoc, but short names, sacrificing structure and understandability in the process. It is of course much nicer to be able to "overload" the "Show"identifier and let its meaning be determined by context (i.e. the type of object "Show" is sent to). This can be easily done in an object oriented language where each class is independent from the rest of the program and provides its own implementation for methods.

Conceptual parallelism (coroutines) can easily be implemented in the object oriented paradigm. Smalltalk, for instance, provides the necessary primitives (predefined classes). Genuine parallelism, on the other hand, is not directly suppored. After sending a message, objects wait for an answer before proceeding on their life cycles. Before the in principle promising properties of object oriented systems can be exploited for genuine parallelism (multi-processing), new concepts for process coordination and interrupt handling must be implemented. Only a few experimental languages (i.e. **ETHER**, ...) have experimented with such ideas.

The main disadvantage of object oriented software architectures lies in its high resource requirements. A multitude of transient objects have to be administered and long chains of indirect references must be processed. This leads to "inefficient" performance compared to classical systems. This problem may soon disappear through performance improvements and price reductions for powerful personal workstations. Another solution lies in the design of special purpose architectures catering for the specific requirements of this programming style. A few experimental projects of this kind are known (i.e. Unger (Berkeley), Suzuki (Japan), ...). Although not an easy task for which immediate results should be expected, it should be much less difficult to make progress here than on the problems faced by logic programming.

Figure 4.1 again summarizes the main characteristics of object oriented programming styles.

Advantages	Disadvantages
1. **Direct mapping** of micro-**world** models **onto** programmed **representations**. • *programs are easier to understand, communicate and justify.*	1. **"relational" knowledge** (i.e. quantification, negation) **must be distributed** throughout the system.
2. Object encapsulation yields **highly modular structures.** • *abstraction; an object's behaviour can depend only on its message protocol, not on its internal representation.* • *modularity; objects can be designed and tested independently.* • *modularity; programs are easier to maintain.* • *robustness; access to objects' state can only occur thorugh their methods.*	2. **Many transient,** dynamically created **objects**: memory-intensive.
3. **Cognitive Economy** can be achieved **through differential descriptions and inheritance networks.**	3. Many **long chains of references and multi-level interpretation:** processor-intensive.
4. **Overloading** of operator names can **easily** be **achieved**.	
5. Offers an **excellent framework for** conceptual **(coroutines) and good potential for** real-time concurrency **(tasks)**.	

Figure 4.1 : **Object-Oriented Programming: Advantages/Disadvantages.**

In closing it should be observed that the advantages of object oriented programming hold for Lisp-based flavour systems as well as for Smalltalk. There the processing overhead is less dramatic, because of the usually much less sophisticated user interface.
One aspect which was not explored at all by this case study relates to graphics. Both ExperLisp and Smalltalk support rich libraries of graphics functions, which can also handle windows and mouse interactions. It would be interesting to explore the programming effort required to implement at least some simple graphics interface for a control system laboratory.

Both Lisp and Smalltalk have been shown to be very attractive candidates as implementation languages for rapid prototyping of models of control system. **Lisp** is more widely known and used and has lower resource requirements, while **Smalltalk** provides both a better support for object oriented programming and a much more sophisticated programming environment. It is, however, plagued by the expensive hardware needed to fully exploit its potential.
Whether hardware prices will drop sufficiently to make this an irrelevant consideration remains to be seen. The quality of the MacIntosh implementation is certainly remarkeable in view of the resource restrictions it has to cope with.
An extended version of ExperLisp with improved debugging support and additional object oriented programming features may well be a reasonable compromise in the meantime.

More research is required in a number of areas which have not be explored for reasons of time and resource restrictions.

It would be instructive to analyze the suitability and restrictions of **Prolog**. The ease (or lack thereof) of programming graphical interfaces in Lisp and Smalltalk could also be analysed.
More complete implementations of toolboxes would have been desirable in both cases in order to achieve some reliable intuition about their relative efficiencies.

All this would provide valuable addtional insights. It seems, however, unlikely that the main conclusions drawn from this study would be significantly affected by them.

Appendices:

The code for the **Lisp** and **Smalltalk** toolboxes is shown in the following two appendices.

The programs have been written with a view on readability. They are thoroughly formatted, commented and any attempt has been made to use "meaningful" identifiers.

A few **coding conventions** were used throughout. All **global identifiers** start with an upper-case letter, whereas all **locals** always start with lower-case. **Parameters** are prefixed with the phrase "a" or "an" or "some" (i.e. aList, someCoefficients). In Lisp all **predicate functions** end with a question (i.e. EmptyList?) and all **functions which will produce side-effects** (apart from printing) end with an exclamation mark (i.e. SetThisValue!). This convention was adopted from Scheme. It could, unfortunately, not be used in Smalltalk, because its syntax only allows letters and numbers as parts of identifiers.

For the Smalltalk toolbox it should be noted that the **left-pointing arrow** is printed as a "-" here. This is due to a restricted character set in the used font and was not changed for lack of time.

```
; ***********************************************************
; *                  Level Beta (Basic Concepts)            *
; ***********************************************************

; ********** Implementation of "SYSTEM" (as a structure)

(DEFSTRUCT System
(defines the structure of sytem descriptions)
    Name      Inputs    States   Outputs
    Subsystems  Behaviour)

; ********** constructor and destructor fns

(DEFUN MakeSystem
(creates & returns a new system object)
   (aName  anInputList  aStateList  anOutputList
    aSubsystemList aStateEqnList anOutputEqnList)
  (LET (x)
     (SETQ x (make-System))
     (SETF (System-name         x) aName)
     (SETF (System-inputs       x) anInputList)
     (SETF (System-outputs      x) anOutputList)
     (SETF (System-states       x) aStateList)
     (SETF (System-subsystems   x) aSubsystemList)
     (SETF (System-behaviour    x)
        (MakeBehaviourList aStateEqnList
                    anOutputEqnList )) x ))

(DEFUN NewName (aSystem aName)
(returns a changed (new name) system instance)
  (LET (x)
     (SETQ x aSystem)
     (SETF (System-name x) aName) x ))

(DEFUN AddInputs (aSystem anInputList)
(returns a changed (additional inputs) system instance)
; WARNING: does not check for duplicates !
  (LET (x)
    (SETQ x aSystem)
    (SETF (System-Inputs x) (APPEND (Inputs aSystem)
                                    anInputList) ) x ))

(DEFUN ClearInputs (aSystem)
(returns the system without inputs (set to NIL))
  (LET (x)
     (SETQ x aSystem)
     (SETF (System-Inputs x) NIL) x ))
```

```
(DEFUN AddOutputs (aSystem anOutputList)
(returns a changed (additional outputs) system instance)
; WARNING: DOes not check for duplicates !
  (LET (x)
      (SETQ x aSystem)
      (SETF (System-outputs x) (APPEND (Outputs aSystem)
                                       anOutputList)) x ))

(DEFUN ClearOutputs (aSystem)
(returns the system without outputs (set to NIL))
  (LET (x)
      (SETQ x aSystem)
      (SETF (System-Outputs x) NIL) x ))

(DEFUN AddStates (aSystem aStateList)
(returns a changed (additional states) system instance)
; WARNING: Does not check for duplicates !
  (LET (x)
      (SETQ x aSystem)
      (SETF (System-States x) (APPEND (States aSystem)
                                      aStateList)) x ))

(DEFUN ClearStates (aSystem)
(returns the system without states (set to NIL))
  (LET (x)
      (SETQ x aSystem)
      (SETF (System-States x) NIL) x ))

(DEFUN AddSubsystems (aSystem aSubsystemList)
(returns a changed (additional subsystems) system instance)
; WARNING: Does not check for duplicates !
  (LET (x)
      (SETQ x aSystem)
      (SETF (System-Subsystems x) (APPEND (Subsystems aSystem)
                                          aSubsystemList)) x ))

(DEFUN ClearSubsystems (aSystem)
(returns the system without subsystems (set to NIL))
  (LET (x)
      (SETQ x aSystem)
      (SETF (System-Subsystems x) NIL) x ))

(DEFUN NewBehaviour (aSystem aStateEqnList anOutputEqnList)
(returns a changed (new behaviour) system instance)
  (LET (x)
      (SETQ x aSystem)
      (SETF (System-Behaviour x)
          (MakeBehaviourList aStateEqnList anOutputEqnList) )
 x ))
```

```
(DEFUN ClearBehaviour (aSystem)
/returns the system without any template for behaviour
(set to NIL))
  (LET (x)
     (SETQ x aSystem)
     (SETF (System-Behaviour x) (MakeBehaviourList NIL NIL)) x ))

; ********** query & selector fns

(DEFUN Name?        (aSystem) (IF (Name        aSystem) T NIL))
(DEFUN Inputs?      (aSystem) (IF (Inputs      aSystem) T NIL))
(DEFUN Outputs?     (aSystem) (IF (Outputs     aSystem) T NIL))
(DEFUN States?      (aSystem) (IF (States      aSystem) T NIL))
(DEFUN Subsystems?  (aSystem) (IF (Subsystems  aSystem) T NIL))
(DEFUN Connections? (aSystem) (IF (Connections aSystem) T NIL))
(DEFUN Behaviour?   (aSystem) (IF (Behaviour   aSystem) T NIL))

(DEFUN Name (aSystem)
 (IF (EQUAL aSystem NIL)  NIL
     (System-name (EVAL aSystem))))
(DEFUN Inputs (aSystem)
 (IF (EQUAL aSystem NIL)  NIL
     (System-inputs (EVAL aSystem))))
(DEFUN Outputs (aSystem)
 (IF (EQUAL aSystem NIL)  NIL
     (System-outputs (EVAL aSystem))))
(DEFUN States (aSystem)
 (IF (EQUAL aSystem NIL)  NIL
     (System-states (EVAL aSystem))))
(DEFUN Subsystems (aSystem)
 (IF (EQUAL aSystem NIL)  NIL
     (System-subsystems (EVAL aSystem))))
(DEFUN Connections (aSystem)
 (IF (EQUAL aSystem NIL)  NIL
     (RemoveDuplicatesFrom
      (APPEND (Inputs aSystem) (Outputs aSystem))) )
(DEFUN Behaviour  (aSystem)
 (IF (EQUAL aSystem NIL) NIL
     (System-behaviour  (EVAL aSystem))))
```

```
; ********** display-fns

(DEFUN ShowSystem (aSystem)
 (PRINT "==============================")
 (PRINC "  SYSTEM: ")
 (PRINT        (Name aSystem))
 (PRINT "==============================")
 (PRINC " Inputs  : ")  (PRINT (Inputs  aSystem))
 (PRINC " Outputs : ")  (PRINT (Outputs aSystem))
 (PRINC " States  : ")  (PRINT (States  aSystem)) (TERPRI)
 (COND ((Subsystems? aSystem)
      (PRINT " *** Subsystems *** ")
      (PRINT "      ++++++++++      ")
      (PRINT (Subsystems  aSystem))    (TERPRI)) )
 (COND ((Connections? aSystem)
      (PRINT " *** Connections *** ")
      (PRINT "      ++++++++++      ")
      (ShowConnectionTable (TableOfConnections aSystem))
      (TERPRI)) )
 (COND ((Behaviour? aSystem)
      (PRINT " *** Behaviour *** ")
      (PRINT "      +++++++++      ")
      (ShowBehaviourList (Behaviour aSystem)) ))
 NIL )

; Implementation of "identifier" is (directly) via Lisp symbols.
; Implementation of "ValueList" (inputs, outputs, states) and
; "SystemCollectionList" (set of systems) is (directly) via Lisp
; list-structures.

; Implementation of "BinaryConnection" is via pairs (lists of two
; atoms). The "CONNECTIONTABLE" data type can therefore be
; represented as an association-table (list of such pairs).

; ********** constructor-fns

(DEFUN TableOfConnections (aSystem)
 (LET (x)
  (SPECIAL x)
  (SETQ x aSystem)
   (APPEND
    (MAPCAR (LAMBDA (source)
                (MakeBinaryConnection source (Name x)))
              (Inputs x ) )
    (MAPCAR (LAMBDA (dest  )
                (MakeBinaryConnection (Name x)  dest))
              (Outputs x) )) ))

(DEFUN MakeBinaryConnection (aSource aDest) (LIST aSource aDest))
```

```
(DEFUN MakeConnectionTable (anAssoc &rest moreAssocs)
  (APPEND (LIST anAssoc) moreAssocs) )

; ********** query & selector fns

(DEFUN Connections? (aConnectionTable) (NOT (NULL aConnectionTable)))

(DEFUN FirstConnection    (aConnectionTable)  (CAR aConnectionTable))
(DEFUN RestOfConnections (aConnectionTable)  (CDR aConnectionTable))

(DEFUN Source (aBinaryConnection)  (CAR  aBinaryConnection))
(DEFUN Dest    (aBinaryConnection)  (CADR aBinaryConnection))

***** display-fns

(DEFUN ShowConnectionTable (aConnectionTable)
 (PrintPairs aConnectionTable) (TERPRI) )

(DEFUN PrintPairs (aListOfPairs)
; auxiliary for "ShowConnectionTable"
 (COND ((NULL aListOfPairs) NIL)
       (T (PRINT     (CAR aListOfPairs))
          (PrintPairs (CDR aListOfPairs)) ) ))

; Implementation of "BEHAVIOURLIST" is through a nested list of
; expressions:(List of list of
;                 state-eqns: (list of eqns across all n state
;                         vars. (poss. involving inputs))
;           &     list of
;                 output-eqns:(list of eqns across all m output
;                         vars. (poss. involving states)))

; ********** constructor-fns

(DEFUN MakeBehaviourList (aStateEqnList anOutputEqnList)
  (LIST aStateEqnList anOutputEqnList))

(DEFUN AddStateEqns (aBehaviourList aStateEqnList)
 (LET (x)
   (SETQ x (APPEND (StateEqns aBehaviourList) aStateEqnList))
  x))
(DEFUN AddOutputEqns (aBehaviourList anOutputEqnList)
 (LET (x)
   (SETQ x (APPEND (OutputEqns aBehaviourList) anOutputEqnList))
  x))
```

```
; ********** query & selector fns

(DEFUN StateEqns? (aBehaviourList)
  (IF (StateEqns aBehaviourList) T NIL))
(DEFUN OutputEqns? (aBehaviourList)
  (IF (OutputEqns aBehaviourlist) T NIL))

(DEFUN StateEqns  (aBehaviourList) (CAR aBehaviourList) )
(DEFUN OutputEqns (aBehaviourList) (CADR aBehaviourList) )

(DEFUN FirstEqn    (anEqnList) (CAR anEqnList))
(DEFUN RestOfEqns (anEqnList) (CDR anEqnList))

; ********** display-fns

 (DEFUN ShowBehaviourList (aBehaviourList)
  (COND ((StateEqns? aBehaviourList)
         (PRINT "State-Eqns")
         (PRINT "----------")
         (MAPCAR PRINT (StateEqns aBehaviourList)))
        (T (PRINT "+++ no state equations defined +++")) )
  (COND ((OutputEqns? aBehaviourList)
         (PRINT "Output-Eqns")
         (PRINT "-----------")
         (MAPCAR PRINT (OutputEqns aBehaviourlist)))
        (T (PRINT "+++ no output equations defined +++")) ) )

; Behaviourlists contain equations represented as expressions.
; Implementation of an "EXPRESSION" is through nested lists in
; (fully parenthesized) infix notation (prefix notation is used
; internally and can easily be offered at the user level).

; ********** constructor & simplification fns

(DEFUN MakePrefixBinary (anOperator aFirstArg aSecondArg)
  (LIST anOperator aFirstArg aSecondArg) )
(DEFUN MakeInfixBinary  (aFirstArg anOperator aSecondArg)
  (LIST aFirstArg anOperator aSecondArg) )
(DEFUN MakeUnary (aSign aSymbol)
 (COND ((OR (EQUAL '+ aSign) (EQUAL '- aSign))
        (LIST aSign aSymbol))
       (T (PRINT "INVALID sign in unary expression") NIL ) ) )
```

```
(DEFUN MakePrefixSum (arg1 arg2)
 (COND ((AND (NUMBERP arg1) (NUMBERP arg2)) (+ arg1 arg2))
       ((NUMBERP arg1)
        (IF (ZEROP arg1) arg2
            (MakePrefixBinary '+ arg1 arg2)) )
       ((NUMBERP arg2)
        (IF (ZEROP arg2) arg1
            (MakePrefixBinary '+ arg1 arg2)) )
       (T (MakePrefixBinary '+ arg1 arg2)) ))

(DEFUN MakePrefixDifference (arg1 arg2)
 (COND ((AND (NUMBERP arg1) (NUMBERP arg2)) (- arg1 arg2))
       ((NUMBERP arg1)
        (COND ((ZEROP arg1)  (MakeUnary '- arg2))
              (T (MakePrefixBinary '- arg1 arg2))) )
       ((NUMBERP arg2)
        (IF (ZEROP arg2) arg1
            (MakePrefixBinary '- arg1 arg2)) )
       (T (MakePrefixBinary '- arg1 arg2)) ))

(DEFUN MakePrefixProduct (Multiplicand Multiplier)
 (COND ((AND (NUMBERP multiplicand) (NUMBERP multiplier))
        (* multiplicand multiplier))
       ((NUMBERP multiplicand)
        (COND ((ZEROP multiplicand)   0)
              ((= multiplicand  1 ) multiplier)
              ((= multiplicand -1 ) (MakeUnary '- multiplier))
              (T  (MakePrefixBinary '* multiplicand multiplier))) )
      ((NUMBERP multiplier)
       (COND ((ZEROP multiplier  )   0)
             ((= multiplier  1 )  multiplicand)
             ((= multiplier -1 ) (MakeUnary '- multiplicand))
             (T  (MakePrefixBinary '* multiplicand multiplier))) )
      (T  (MakePrefixBinary '* multiplicand multiplier)) ))

(DEFUN MakePrefixQuotient (Dividend Divisor)
 (COND ((AND (NUMBERP dividend) (NUMBERP divisor))  (/ arg1 arg2))
       ((NUMBERP dividend)
        (COND ((ZEROP dividend)   0)
              (T  (MakePrefixBinary '/ dividend divisor))) )
      ((NUMBERP divisor)
       (COND ((ZEROP divisor)
              (PRINT "Attempt to divide by zero ! Ignored."))
             ((= divisor  1 )  dividend)
             (T  (MakePrefixBinary '/ dividend divisor)) ) )
      (T  (MakePrefixBinary '/ dividend divisor)) ))
```

```
(DEFUN MakePrefixPower (aBase anExponent)
  (COND ((AND (NUMBERP aBase) (NUMBERP anExponent))
         (^ aBase anExponent))
        ((NUMBERP aBase)
         (COND ((ZEROP aBase) 0)
               ((= aBase 1)  1)
               (T  (MakePrefixBinary '** aBase anExponent))) )
        ((NUMBERP anExponent)
         (COND ((ZEROP anExponent) 1)
               ((= anExponent  1 ) aBase)
               (T  (MakePrefixBinary '** aBase anExponent))) )
        (T  (MakePrefixBinary '** aBase anExponent)) ))

; ********** query & selector fns

(DEFUN Constant? (anExpression)
  (IF (ATOM anExpression) (NUMBERP anExpression)) )
(DEFUN Variable? (anExpression)
  (IF (ATOM anExpression) (SYMBOLP anExpression)) )
(DEFUN SameVar? (firstExpression secondExpression)
  (IF (AND (Variable? firstExpression) (Variable? secondExpression))
      (EQUAL firstExpression secondExpression)) )

(DEFUN Unary? (anExpression)
  (NULL (PrefixSecondArg anExpression)) )

(DEFUN PrefixSum? (anExpression)
  (IF (NOT (ATOM anExpression))
      (EQUAL (PrefixOperator anExpression) '+)) )
(DEFUN PrefixDifference? (anExpression)
  (IF (NOT (ATOM anExpression))
      (EQUAL (PrefixOperator anExpression) '-)) )
(DEFUN PrefixProduct? (anExpression)
  (IF (NOT (ATOM anExpression))
      (EQUAL (PrefixOperator anExpression) '*)) )
(DEFUN PrefixQuotient? (anExpression)
  (IF (NOT (ATOM anExpression))
      (EQUAL (PrefixOperator anExpression) '/)) )
(DEFUN PrefixPower? (anExpression)
  (IF (NOT (ATOM anExpression))
      (EQUAL (PrefixOperator anExpression) '**)) )

{----- NOTE: the corresponding "Infix..." fns have not been
                implemented !}

(DEFUN UnaryOperator  (anExpression) (CAR  anExpression) )
(DEFUN UnaryArg       (anExpression) (CADR anExpression) )
```

```
; ----- for PREFIX expressions -----
(DEFUN PrefixOperator  (anExpression) (CAR  anExpression) )
(DEFUN PrefixFirstArg  (anExpression) (CADR  anExpression) )
(DEFUN PrefixSecondArg (anExpression) (CADDR anExpression) )

; ----- for INFIX expressions -----
(DEFUN InfixOperator  (anExpression) (CADR  anExpression) )
(DEFUN InfixFirstArg  (anExpression) (CAR  anExpression) )
(DEFUN InfixSecondArg (anExpression) (CADDR anExpression) )

; ********** Transformer-fns

(DEFUN InfixToPrefix (aNestedInfixExpression)
 (COND ((NULL aNestedInfixExpression) NIL)
       ((ATOM aNestedInfixExpression) aNestedInfixExpression)
       ((Unary? aNestedInfixExpression)
        (MakeUnary (UnaryOperator aNestedInfixExpression)
                   (InfixToPrefix
                     (UnaryArg aNestedInfixExpression)) ))
       ((Flat? aNestedInfixExpression)
        (MakePrefixBinary (InfixOperator  aNestedInfixExpression)
                          (InfixFirstArg  aNestedInfixExpression)
                          (InfixSecondArg aNestedInfixExpression)))
     ( T (MakePrefixBinary
           (InfixOperator aNestedInfixExpression)
           (InfixToPrefix (InfixFirstArg aNestedInfixExpression))
           (InfixToPrefix (InfixSecondArg aNestedInfixExpression))
     )) ))

(DEFUN PrefixToInfix (aNestedPrefixExpression)
  (COND ((NULL aNestedPrefixExpression) NIL)
        ((ATOM aNestedPrefixExpression) aNestedPrefixExpression)
        ((Unary? aNestedPrefixExpression)
          (MakeUnary (UnaryOperator aNestedPrefixExpression)
                     (PrefixToInfix
                       (UnaryArg aNestedPrefixExpression))))
       ((Flat? aNestedPrefixExpression)
         (MakeInfixBinary (PrefixFirstArg  aNestedPrefixExpression)
                          (PrefixOperator  aNestedPrefixExpression)
                          (PrefixSecondArg aNestedPrefixExpression)))
       (T (MakeInfixBinary
            (PrefixToInfix (PrefixFirstArg aNestedPrefixExpression))
                          (PrefixOperator aNestedPrefixExpression)
                          (PrefixToInfix
                            (PrefixSecondArg aNestedPrefixExpression))
       )) ))
```

```
(DEFUN Flat? (aList)
; auxiliary fn for "InfixToPrefix" and "PrefixToInfix".
; this fn returns true only if a list is not nested
 (LET ((result T))
  (COND ((ATOM aList) t)
        ((NULL aList) result)
        (T (SETQ result (ATOM (CAR aList)))
           (IF result (Flat? (CDR aList)) )) ) ))

(DEFUN Derive (anExpression aVar)
/differentiates a behavioural expression with respect to a given
variable. The expression must be in fully nested infix notation
and an expression with the same structure is returned.
Only the following BINARY operators are recognized:
+, -, *, /, **./
 (PrefixToInfix (DeriveAux (InfixToPrefix anExpression) aVar)) )

(DEFUN DeriveAux (anExpression aVar)
; does all the hard work for "Derive". This fn uses prefix
; representation for equations.
 (COND ; expression is a constant or variable
       ; : 1 if same as der. var, 0 otherwise
      ((Constant? anExpression) 0)
      ((Variable? anExpression)
       (IF (SameVar? anExpression aVar) 1 0) )
       ; expression is a sum or difference
       ; : add or subtract their differentials
      ((PrefixSum? anExpression)
       (MakePrefixSum
          (DeriveAux (PrefixFirstArg  anExpression) aVar)
          (DeriveAux (PrefixSecondArg anExpression) aVar)) )
      ((PrefixDifference? anExpression)
       (MakePrefixDifference
          (DeriveAux (PrefixFirstArg  anExpression) aVar)
          (DeriveAux (PrefixSecondArg anExpression) aVar)) )
       ; expression is a Product
       ;:multiply with their partners' differentials and add
      ((PrefixProduct? anExpression)
       (MakePrefixSum
         (MakePrefixProduct (PrefixFirstArg anExpression)
                    (DeriveAux (PrefixSecondArg
                               anExpression)  aVar))
         (MakePrefixProduct (PrefixSecondArg anExpression)
                    (DeriveAux (PrefixFirstArg
                               anExpression)  aVar)) ))
```

```
      ; :differentiate product of first and
      ;  (1 over second arg).
      ((PrefixQuotient? anExpression)
       (DeriveAux
        (MakePrefixProduct
          (PrefixFirstArg anExpression)
          (MakePrefixPower
            (PrefixSecondArg anExpression) -1)) aVar))
      ; expression is a Power (exponentiation)
      ; : differentiate product of exponent and (one less
      ;   than original exponentiation).
      ((PrefixPower? anExpression)
        (MakePrefixProduct
          (PrefixSecondArg anExpression)
          (MakePrefixProduct
            (MakePrefixPower
              (PrefixFirstArg anExpression)
              (MakePrefixDifference
                (PrefixSecondArg anExpression) 1))
            (DeriveAux (PrefixFirstArg anExpression) aVar)
          ))) ))

; ********** display-fns

(DEFUN ShowExpression (anExpression) (PRINT anExpression))

; a system's behaviour can alternatively be described in matrix
; notation. Such "BEHAVIOURTABLE" objects are represented as lists
; with 3 elements: 1) System matrix
;                     (n*n; across all n states),
;                  2) Input Distribution matrix
;                     (n*m; across all n states & all m inputs),
;                  3) Measurement matrix
;                     (1*n; across all 1 outputs & all n states).

; ********** constructor-fns

(DEFUN MakeBehaviourTable (aSystMat anInputDistMat aMeasureMat)
  (LIST aSystMat anInputDistMat aMeasureMat))

(DEFUN AddMatrix (aBehaviourTable aCoefficientMatrix)
  (APPEND aBehaviourTable (LIST aCoefficientMatrix)))
```

```
; ********** query & selector fns

(DEFUN SystemMatrix?  (aBehaviourTable)
  (NOT (NULL (SystemMatrix aBehaviourTable))) )
(DEFUN InputDistMatrix?  (aBehaviourTable)
  (NOT (NULL (InputDistMatrix aBehaviourTable))) )
(DEFUN MeasurementMatrix? (aBehaviourTable)
  (NOT (NULL (MeasurementMatrix aBehaviourTable))) )

(DEFUN SystemMatrix       (aBehaviourTable) (CAR  aBehaviourTable))
(DEFUN InputDistMatrix    (aBehaviourTable) (CADR  aBehaviourtable))
(DEFUN MeasurementMatrix (aBehaviourTable) (CADDR aBehaviourTable))

; ********** display-fns

(DEFUN ShowBehaviourTable (aBehaviourTable)
  (PRINT "System - Matrix")
  (PRINT "---------------")
  (ShowCoeffMatrix (SystemMatrix aBehaviourTable))
  (PRINT "Input Distribution - Matrix")
  (PRINT "--------------------------")
  (ShowCoeffMatrix (InputDistMatrix aBehaviourTable))
  (PRINT "Measurement - Matrix")
  (PRINT "--------------------")
  (ShowCoeffMatrix (MeasurementMatrix aBehaviourTable)) )

; each of the three components of a BehaviourTable is stored as a
; two-dimensional "COEFFICIENTMATRIX" organized by rows (lists).
; One expression (list) or 0 per coefficient.

; ********** constructor-fns

(DEFUN MakeCoefficientMatrix (&REST someRowLists)
  someRowLists)
(DEFUN AddRow (aCoefficientMatrix aRowList)
  (APPEND aCoefficientMatrix (LIST aRowList)))

(DEFUN MakeRow (&REST someCoefficients)
  someCoefficients)
(DEFUN AddCoefficient (aRowList aCoefficient)
  (APPEND aRowList (LIST aCoefficient)))
```

```
; ********** query & selector-fns

(DEFUN EmptyMatrix? (aCoefficientMatrix) (NULL aCoefficientMatrix))

(DEFUN FirstRow     (aCoefficientmatrix) (CAR aCoefficientMatrix))
(DEFUN RestOfRows (aCoefficientmatrix) (CDR aCoefficientMatrix))
(DEFUN FirstInRow  (aRowList) (CAR aRowList))
(DEFUN RestInRow   (aRowList) (CDR aRowList))

; ********** display-fns

(DEFUN ShowCoeffMatrix (aCoefficientMatrix)
 (DOLIST (row aCoefficientMatrix) (ShowRow row)) )

(DEFUN ShowRow (aCoefficientList)
 (IF (NULL aCoefficientList) (PRINC "No coefficients in this row"))
 (DOLIST (coeff aCoefficientList)
   (COND ((ATOM coeff) (PRINC coeff) )
         ((EQUAL (LENGTH coeff) 1) (PRINC (CAR coeff)))
         (T (PRINC coeff)) )
   (PRINC " ") )
   (TERPRI) )
```

```
; ***************************************************
; *                Level Gamma (Applications)                *
; ***************************************************

(DEFUN ShowSubsystemHierarchy (aSystem aTabLevel)
 (PRINT "++++++++++++++++++++++++++++++++++++++" )
 (PRINC "  Subsyststems of: ") (PRINT (Name aSystem))
 (PRINT "++++++++++++++++++++++++++++++++++++++") (TERPRI)
 (COND ((NOT (Subsystems? aSystem)) NIL)
     (T  (ShowSubsystemHierarchyAux (Subsystems aSystem)
         0
         aTabLevel)) ))

(DEFUN ShowSubsystemHierarchyAux
                                  (aSystemList someSpaces aTabLevel)
 (COND ((NULL aSystemList) NIL)
       (T (PrintSpaces someSpaces)
         (PRINT (CAR aSystemList)) (TERPRI)
         (ShowSubsystemHierarchyAux (Subsystems (CAR aSystemList))
                                    (+ someSpaces aTabLevel)
                                    aTabLevel)
        (ShowSubsystemHierarchyAux (CDR aSystemList)
                                   someSpaces
                                   aTabLevel )) ))

(DEFUN PrintSpaces (someSpaces)
; auxiliary for "ShowSubsystemHierarchyAux"
  (DOTIMES (index someSpaces) (PRINC " ")) )

(DEFUN GetAllSubsystems (aSystemCollection)
(This function generates a list of the subsystems connected to
all the systems in a collection (depth-first search))
(IF (NOT (NULL aSystemCollection))
     (RemoveDuplicatesFrom (APPEND
       ;get the first system's subsystems
        (Subsystems (CAR aSystemCollection))
       ;get these system's subsystems
        (GetAllSubsystems (Subsystems (CAR aSystemCollection)) )
       ;process the rest of the list
        (GetAllSubsystems (CDR aSystemCollection)) )) ))

(DEFUN AllSubsystems (aSystem)
; of a single system !
 (GetAllSubsystems (LIST aSystem)) )
```



```
(DEFUN RemoveDuplicatesFrom (aList)
```

```
 (IF (NOT (NULL aList))
     (CONS (CAR aList)
      (RemoveDuplicatesFrom (REMOVE (CAR aList) (CDR aList))) ) )

(DEFUN GetSubsystemHierarchy (aSystemCollection)
/For each system in the collection this function associates
clusters of subsystems with their names (uses depth-first
search)/
(IF
   (NULL aSystemCollection) NIL
    (APPEND
       ;get the first system's name and a list of its subsystems
        (LIST (CAR       aSystemCollection)
             (Subsystems (CAR aSystemCollection)) )
       ;now get each of the subsystem's subsystems
        (GetSubsystemHierarchy
          (Subsystems (CAR aSystemCollection)))
       ;process the tails of the lists on the 'way up'
        (GetSubsystemHierarchy (CDR aSystemCollection)) ) ))

(DEFUN SubsystemHierarchy (aSystem)
; for a single system /
 (GetSubsystemHierarchy (LIST aSystem)))

(DEFUN GetAllStates (aSystemCollection)
/this fn produces a list of states for all systems contained in a
collection AND all their subsystems ... (depth-first)/
  (IF
    (EQUAL aSystemCollection NIL) NIL
     (LIST
       ;get the first system's states
       (States (CAR aSystemCollection))
       ;recurse down the states of all its subsystems
       (States      (CAR (Subsystems (CDR aSystemCollection))))
       (GetAllStates  (CAR (Subsystems (CDR aSystemCollection))) )
       ;now recurse down the tails of the collections
       (GetAllStates (CDR aSystemCollection)) ) ))

(DEFUN AllStates (aSystem)
; of a single system /
 (GetAllStates (LIST aSystem)))




(DEFUN Linearize (aSystem)
/this fn linearizes a system's behaviour definition by partial
differentiation across all equations, state, input and output
variables. It returns a BehaviourTable-list defining three
matrices (as described above)/
(Let ((x) (table) (matrix) (row))
```

```
(SETQ table '())
(SETQ matrix '())
(DOLIST (eqn (StateEqns (Behaviour x)))
  (SETQ row '())
  (DOLIST (var (States  x))
    (SETQ row (AddCoefficient row (Derive eqn var))))
  (SETQ matrix (AddRow matrix row)) )
(setq table (AddMatrix table matrix)) (setq matrix '())
(DOLIST (eqn (StateEqns (Behaviour x)))
  (SETQ row '())
  (DOLIST (var (Inputs  x))
    (SETQ row (AddCoefficient row (Derive eqn var))))
  (SETQ matrix (AddRow matrix row)) )
(setq table (AddMatrix table matrix)) (setq matrix '())
(DOLIST (eqn (OutputEqns (Behaviour x)))
  (SETQ row '())
  (DOLIST (var (States  x))
    (SETQ row (AddCoefficient row (Derive eqn var))))
  (SETQ matrix (AddRow matrix row)) )
(setq table (AddMatrix table matrix))
table) )
```

```
Object subclass: #Demo

instanceVariableNames: 's1 s2 model ff reg proc sensor '

classVariableNames: ''
poolDictionaries: ''
category: 'Syst-Description'

instance methods For: 'syst-creation'

ff
"creates a system"
ff - System new.
ff initialize;  addInput: #u; addOutput: #y; addState: #x.
^ff

model
"creates a system"
model - System new.
model initialize;  addInput: #c; addOutput: #y; addState: #(x1 x2).
^model

proc
"creates a system"
proc - System new.
proc initialize;  addInput: #u; addOutput: #y; addState: #(h1 h2).
^proc

reg
"creates a system"
reg - System new.
reg initialize;  addInput: #(r y); addOutput: #u; addState: #(i d).
^reg

sensor
"creates a system"
sensor - System new.
sensor initialize; addInput: #u; addOutput: #y; addState: #(x1 x2).
^sensor

s1
"a simple system"
s1 - System new.
s1 initialize;  addInput: #c; addOutput: #y; addSubsystem: s2;
```

```
addSubsystem: model;
addSubsystem: ff;
addConnectionFrom: #uModel to: #c;
addConnectionFrom: #uFF    to: #c;
addConnectionFrom: #rS2    to: #yModel;
addConnectionFrom: #y      to: #yS2.
^s1
```

```
s2
"creates a system"
s2 - System new.
s2 initialize;  addInput: #(r u);  addOutput: #y;
addSubsystem: reg;
addSubsystem: proc;
addSubsystem: sensor;
addConnectionFrom: #rReg      to:#r;
addConnectionFrom: #yReg      to:#ySensor
addConnectionFrom: #uProc    to:#uReg;
addConnectionFrom: #uProc    to:#uFF;
addConnectionFrom: #ySensor to:#y.
^s2

class VariableNames: "

 class Methods For: 'As yet unclassified'

new
"create a new demonstration"
^super new

workspace
"setup a test scenario"
| x  y  model  ff  reg  proc  sensor  s1  s2 |
y - Demo new.
model - y model.
ff - y ff.
reg - y reg.
proc - y proc.
sensor - y sensor.
s1 - y s1.
s2 - y s2.
x - SystemCollection new.
x initialize.
x  addSystem: s1; addSystem: s2.

Object subclass: #System

instanceVariableNames: 'input output state subsystems connections'

classVariableNames: "
poolDictionaries: "
category: 'Syst-Description'
```

instance methods For: 'initialization'

initialize
"initialize all data structures of a system"
| |
input - Set new.
output - Set new.
state - Set new.
subsystems - Set new.
connections - Dictionary new

instance methods For: '**constructors**'

addConnectionFrom: aSource **to**: aDest
"include a path between systems"
| |
connections at: aSource put: aDest

addInput: aSymbol
"define an input to a system"
| |
input addAll: aSymbol

addOutput: aSymbol
"define an output from a system"
| |
output addAll: aSymbol

addState: aSymbol
"define a state of a system"
| |
state addAll: aSymbol

addSubsystem: aSystem
"include a subsystem"
| |
subsystems add: aSystem

instance methods For: '**selectors**'

getAllConnections
"retrieve a system's direct connections"
| |
^ connections

getAllSubsystems
"retrieve a system's direct subsystems"
| |
^ subsystems

getInput
"retrieve a system's input"
| |
^ input

```
getOutput
"retrieve a system's output"
| |
^ output

 getState
"retrieve a system's state"
| |
^ state
```

instance methods For: 'queries'

hasConnections
"check"
| |
^ connections isEmpty not

hasInput
"check"
| |
^ input isEmpty not

hasOutput
"check"
| |
^ output isEmpty not

hasState
"check"
| |
^ state isEmpty not

hasSubsystems
"check"
| |
^ subsystems isEmpty not

instance methods For: 'internal'

expand: aCollection
"recursively searches a collection and returns ALL its (direct and indirect) elements. It assumes that the structure is homogeneous."
| |
(aCollection isEmpty) ifTrue: [^self]
ifFalse: [^aCollection collect:
[:each| each expand: (each getAllSubsystems)]]

class methods For: 'instantiate'

new
"creates a new system"
| |
^ super new

```
Object subclass: #SystemCollection

instanceVariableNames: 'components '
classVariableNames: ''
poolDictionaries: ''
category: 'Syst-Description'

instance methods For: 'initialization'

initialize
"initialize web"
| |
components - OrderedCollection new

instance methods For: 'constructors'

addSystem: aSystem
"add the systems to the web"
| |
components addLast: aSystem

instance methods For: 'selectors'

getAllSystemsHierarchy
"retrieve ALL the systems in the web (follows subsystem-links!!)"
| |
^ (components collect: [:each| each expand: (each getAllSubsystems)] )

getFirstSystem
"retrieve the first system in the web"
| |
^ components first

getSystems
"retrieve all 'top-level' systems in the web"
| |
^ components

getToplevelStates
"retrieve states for all 'top-level' systems in the web"
| |
^ (components collect: [:each| (each hasState)
                                    ifTrue: [each getState] ]  )
```

instance methods For: 'destructors'

removeAll
"remove all the systems from the web"
| |
[components isEmpty] whileFalse: [components removeFirstSystem]

removeFirstSystem
"remove the first system from the web and return it"
| |
^ components removeFirst

class methods For: 'instantiate'

new
"creates a new web of systems"
| |
^ super new

Bibliography

Abelson, H./ Sussman, J./ Sussman, J. 1985
Structure and Interpretation of Computer Programs
Cambridge (Mass.) 1985

Allen, J. 1978
Anatomy of LISP
New York 1978

Barron, D.W. 1977
An Introduction to the Study of Programming Languages
Cambridge 1977

Brown, H.I. 1977
Perception, Theory and Comittment. The New Philosophy of Science
Chicago & London 1977

Charniak, E./ Riesbeck, C./ Mcdermott, D. 1980
Artificial Intelligence Programming
Hillsdale 1980

Clocksin, W.F./ Mellish, C.S. 1981
Programming in Prolog
New York / Berlin 1981

Davis, P.J./ Hersch, R. 1981
The Mathematical Experience
Brighton 1981

Goldberg, A./ Robson, D. 1979
A Metaphor for User Interface Design
Proceedings 12th Hawaii International Conference on System Sciences
Vol.6, No.1(1979); 148-157

Goldberg, A./ Robson, D. 1983
Smalltalk-80. The Language and Its Implementation
Reading/ Menlo Park ... 1983

Hamming, R.W. 1962
Numerical Methods for Scientists and Engineers
New York 1962

Hofstadter, D.R. 1979
Goedel, Escher, Bach: The Eternal Golden Braid
New York 1979

Klir, G.J. (ed.) 1972
Trends in General Systems Theory
New York/ Cincinatti ... 1972

Kowalski, J. 1979
Programming in Logic
Amsterdam/ New York ... 1979

Kreutzer, W. 1986
Simulation Programming Styles & Languages
Reading/ Menlo Park ... 1986

MacLennan, B.J. 1983
Principles of Programming Languages: Design, Evaluation & Implementation
New York 1983

Olsson, G. 1980
Birds in Egg - driB ni sggE
London 1980

Oren, T.I./ Zeigler, B.P. 1979
Concepts for Advanced Simulation Methodologies
SIMULATION Vol.30, No.3(March 1979); 70 - 82

Piatelli-Palmarini, M. (ed.) 1980
Language and Learning. The Debate between Jean Piaget and Noam Chomsky
Cambridge (Mass.) 1980

Power, H.M./ Simpson, R.J. 1978
Introduction to Dynamics and Control
London/ New York ... 1978

Pylyshyn, Z. 1984
Computation and Cognition
Cambridge (Mass.) 1984

Rich, E. 1983
Artificial Intelligence
London/ New York ... 1983

Rivett, P. 1972
Principles of Model Building
London/ New York ... 1972

Savage, C.W. (ed.) 1978
Perception and Cognition Issues in the Foundations of Psychology
Minnesota Studies in the Philosophy of Science Vol. IX; Minneapolis 1978

Sheil, B.A. (ed.) 1981
The Psychological Study of Programming. Special Issue
ACM Computing Surveys Vol.13(1983)

Sheil, B.A. 1983
Power Tools for Programmers
Datamation February 1983; 131 - 144

Shneiderman, B. 1980
Software Psychology
Cambridge (Mass.) 1980

Sloman, A. 1978
The Computer Revolution in Philosophy
Hassocks 1978

Smith, H.T./ Green, T.R.G. (eds.) 1980
Human Interaction with Computers
London/ New York ... 1980

Tennent, R.D. 1981
Principles of Programming Languages
New York/ London ... 1981

Wickelgren, W.A. 1974
How to Solve Problems
San Franzisco 1974

Winograd, T. 1983
Programming Issues for the 1980s (Panel Discussion)
at: SIGPLAN '83. Symposium on Programming Language Issues in Software Systems
reprinted in: SIGPLAN Notices Vol.19, No.8(Aug. 1984); 51 - 61

Winston, P.H. 1984
Artificial Intelligence; 2nd ed.
Reading/ Menlo Park ... 1984

Wittgenstein, L. 1953
Philosophical Investigations
Oxford 1953

Wittgenstein, L. 1956
Remarks on the Foundations of Mathematics
Oxford 1956

Wittgenstein, L. 1961
Tractatus Logico-Philosophicus
London 1961

Wulf/ Shaw/ Hilfinger/ Flon 1982
Fundamental Structures of Computer Science
Reading/ Menlo Park ... 1982

Zeigler, B.P. 1976
Theory of Modelling and Simulation
London/ New York ... 1976

Zeigler, B.P./ Elzas, M.S./ Klir, G.J./ Oren, T.I. (eds.) 1979
Methodology in Systems Modelling and Simulation
Amsterdam/ New York ... 1979

Zeigler, B.P. 1984
Multifacetted Modelling and Discrete Event Simulation
New York 1984