SOFTWARE CONCEPTS FOR AN ATMOSPHERIC OPTICS PROGRAMMING ENVIRONMENT

V.B. Novosel'tsev and V.T. Kalaida

Institute of Atmospheric Optics,
Siberian Branch, USSR Academy of Sciences, Tomsk
Received November 15, 1988

In this paper, we analyze modern software design methodology and the subject matter of atmospheric optics, and propose the implementation of a software environment conductive to problem-solving in that discipline. We analyze the requirements placed on such a system by both the user and the developer, and suggest the most suitable architecture to meet those requirements.

Atmospheric optics is a scientific discipline characterized by large amounts of data to be processed in solving relevant problems[1]. In addition, there are other peculiarities, including a critical need for complicated computer networks which are used, for example, to process data taken by a grid of outposts, providing for global reconstruction of the fields of a number of optical parameters of the atmosphere. The problem is further complicated by the need to carry out data-processing and analysis in real time.

The above considerations appear to impose very stringent requirements on the software for atmospheric optics problems. The software design concepts proposed reflect the state of the art in computer science, and take advantage of experience gained in the production of special-purpose systems geared to problem-solving in the field of atmospheric optics[2,3]. We use the term "problem-solving system" (PSS) here to describe the software in question.

The PSS comprises an integrated hardware/software environment. In recent years, it has often been pointed out[4–6] that the most time-consuming aspect of such projects is software development. To a certain extent, this is unfortunately due to the fact that the programming of large systems with a great many interfaces is more of an art than a science.

Programs as marketable products must take two points of view into account: that of a potential user, on the one hand, and that of the developer on the other.

User requirements may be divided into two categories. The first includes those of a general nature, and pertain to all modern software systems, while the second includes features specific to PSS.

The first category thus includes (a) user-friendliness; (b) primarily interactive operation; (c) fast system response. The special requirements specific to a PSS- include (a) efficient, powerful and convenient data-management tools; (b) formal description capability for the subject area (SA) chosen, i.e., conceptual and algorithmic knowledge representation; (c) a deductive inference capability and an expert system (ES) built around knowledge base (SAKB) and finally, (d) problem-oriented interfaces between the various PSS- subsystems and the user that provide a wide range of service functions. The latter enable an untrained user to solve problems within the scope of his subject area by means of SSS.

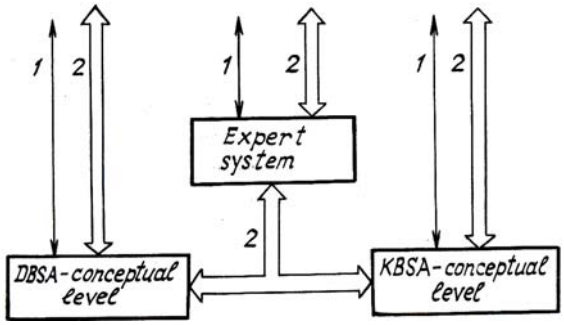

FIG. 1. PSS architecture (user's view-point): user interfaces, ⇔ data streams.

Thus, a user's idea of the functional PSS architecture can in general be represented as in Fig. 1. The user can formulate and solve his problem solely by ES whose functions comprise both an examination (expertise) of its own and proper utilization and storage of data and knowledge. Additionally, there also exists a possibility of direct use and modification of the data and SA knowledge bases. Moreover, a mixed PSSS strategy can be chosen, which implies that the user is free to take the initiative in the man-machine dialog with the ES and introduce the necessary changes in the data (knowledge base, ES inference rules). User interfaces are to meet particular requirements. First, the actual number of interfaces must be minimized. Second, the minimal interface set should be consistent. Third, the interface drivers should be adaptable to the specific subject area, i.e., problem-oriented. Finally, apart from their subject-area adaptability, the interfaces should provide the user with a certain amount of freedom to operate with more complicated entities, such a relationships between ideas, in a language as close to the appropriate professional jargon as possible.

Should we compare the user's ideas of a PSSS with the visible (upper) part of an iceberg, the system functional architecture would then correspond to the iceberg as a whole. This figurative juxtaposition seems to reflect a very important fact: a software system implementation should not be concerned solely with the development of the technical (underwater) aspect of the problem. In other words, the user's ideas as to how the system is to be manipulated should not be ignored. The system is to be well-balanced to ensure that no user's requirements are overlooked and no detail related to the implementation itself come to light. A functional PSS- architecture from a developer's viewpoint, shown in Figure 2, is . a revised version of that given in Figure 1.

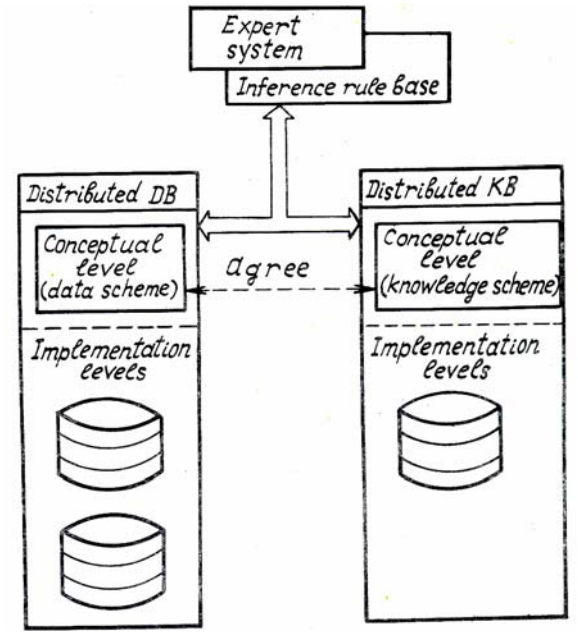

FIG. 2. Revised PSS architecture

Following Lavrov[7], we consider three levels of subject-area corresponding:

1. The conceptual level, where the abstract objects of the subject area corresponding to the natural terms of the relevant theory are specified and certain logical relations between the objects are stated;

2. The algorithmic or procedural level including computational procedures that are implementations of some of the relations from the above level" indicating the feasibility of obtaining output values from input quantities;

3. The actual or subject level, made up of the experimental evidence or observable that represent values for some abstract objects of the subject area that are stored in the database.

The first kind of information (Level 1) may be decomposed into two new sublevels distinguished by the relations involved and the degree of complexity inherent in the notions linked by the relations. This sublevel is often referred to as a computational model of the original subject area (see Ref. 7). The relations from the second sublevel are as a rule much more complicated. They can be defined using heuristic considerations, and are mainly built over nontrivial structures of the subject-area model elements. In our PSSS project, the knowledge of the second sublevel is supported by ES, while the information of the first sublevel forms a subject-area knowledge base describing the computational model.

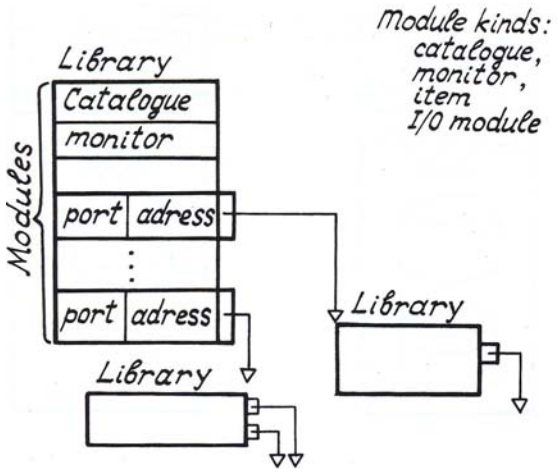

FIG.3. Library system

PSS is organized as an expandable system of monitors that support both internal and interfaces, and implement all other system functions. A number of libraries are also involved in the system. One of the major PSSS principles is its hierarchical configuration. A library serves as a fundamental unit of the system (see Figure 3). PSS includes standard libraries supported by the system library monitor. However, units organized in a different way can be employed too, each having a supporting monitor of its own. A library catalogue contains conventionally located auxiliary information and a pointer to the corresponding monitor loader.

Modules are elements of some library, and in particular, the catalog of a library and the control monitor are modules. Even an external library can be included in a given library as a module. In a given library, a module may even be another library, or more precisely, a pointer to another library and to a memory segment allocated for data transfer (port). It is to be noted that a similar structure (an address port) is incorporated in each library activated by the one of interest. The former will be termed the primary library and the latter will be named secondary. The basic operation cycle at this level is a session. The secondary library activation opens up the session by executing a number of startup procedures, such as, e.g., a primary library port initialization, secondary library duplication, and a monitor linkage. When the session is over, as needed, the primary library monitor can convert a copy of the secondary library into an actual element, introducing a new reference entry in the library catalogue. The proposed structure is readily implemented. Moreover, it matches the functional PSS architecture of Fig. 2 fairly well, and simplifies

project development and support. However, our approach is by no means optimal, its main disadvantages being the abundance of references and strong restrictions on the internal library interfaces.

Another PSS component is, in fact, a set of conventional expert systems of a hierarchical (treelike) structure resulting from the decomposition of a subject area into nearly independent fields. Such an organization of the PSS. expert component exhibits a number of favorable properties. These are summarized as follows:

1. Improvement in execution time

Any expert system is based upon a set of inference rules. A subject field comparable to atmospheric optics area requires a priori a large rule base. Since the expert evaluation efficiency is governed by the second or higher power of the total number of rules, the decomposition of the initial rule base provides a definite improvement in expert-system resource. This is .confirmed by the inequality

$$\left(\sum_i n_i\right)^m > \sum_i n_i^m, \quad m>1$$

2. Simplicity of system modification

It Is important to provide for the maintenance and support phase of the software life cycle. This stage implies that the software has been divorced from the development groups and is available to modification. A structured overall system design (particularly the expert part) facilitates introduction of possible changes in the software after the project has been completed. In our case, tree nodes can be modified, deleted or replaced subject to the condition that standard interface requirements be met.

3. Simplicity of subject area modification

Expert systems reflect current knowledge about the subject area of interest. In effect, this knowledge is not to be treated as something absolute and final, because new information may be acquired and/or the relevant theory may be altered. The proposed hierarchical structure of our expert system, or to be more precise the inherent modularity of such a design, provides a simple means for introducing new conceptual knowledge about a subject area in accordance with the changing views of an expert user in the relevant discipline.

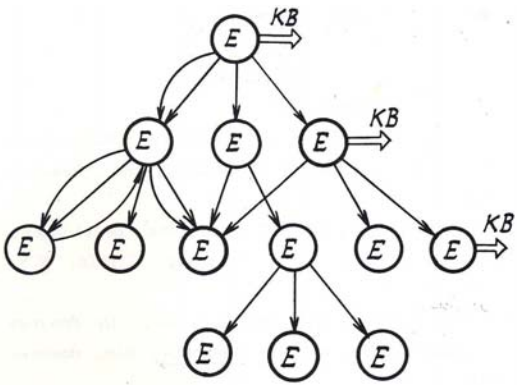

FIG.4. Expert system operation

User interaction with the system consists in making a number of queries of experts at different levels (hereinafter, by "experts" we mean specific components of the system as a whole that are located at the nodes of the three in Fig. 4). The user can abort the current expert communication and return to a higher level, where an expert will have a broader, than detailed information or, following instructions of the current expert, address a lower-level expert for a solution to a particular problem stated at the previous step. Such an expert can also clarify an approach to a solution of the user's problem. Expert system applications appear to be similar to the performance and verification process[8]. However, it should be borne in mind that our approach relies on experts making use of independent traditional ES's with their own rule bases and specific inference procedures. There may be connected branches in the ES tree. This is the case where the system has to deal with different problems (having different Initial states] and, in doing so, to solve similar subproblems. The lowest-level experts are tightly coupled to the computational model and database for the subject area. In the course of this interaction, one builds a program to solve the user's problem, as well as implementing a program execution monitor for the local-area network.

A constituent part of the PSSS architecture is its knowledge base (KB); the latter provides a description of the subject-area model. While implementing specification tools for the SAM definition, we have tried to provide for a well-balanced use of procedural and non-procedural mechanisms. Pure PROLOG[9] or Descartes[10] are typical languages supporting the non-procedural programming style. When both tools were actually used it proved necessary to supplement them with structures provides by more traditional programming languages. In the terminology of Tyugy[11], languages used to describe computational models are designated "conceptual programming languages". A translation of a description of a computational model, combined with an implementation of the elements of that model, constitutes a knowledge base that may be used to automatically obtain a solution to a problem formulated within the CM framework. The problem-solving program is designed to a set of functional specifications. If the CM furnishes a comprehensive characterization of the SA under investigation, advantage can be taken of non-procedural specifications, i.e., it is the parameters to be computed which are specified rather than the operational algorithm. The proposed language for the SAM description combines certain concepts underlying both Descartes[10] and Utopist[11,12].

Informally, a CM according to Tyugu[13] (referred to as a standard) is a set of concept names in some applied theory, known as elements, and the computational relationships that connect them. While both Descartes and Utopist are capable of describing models structurally, they essentially do no more than reduce the amount of testual description involved. The

point is that before the analysis/synthesis procedures are executed, the curtailed definition is recast in a standard CM form (the unfolding process). Undoubtedly, of great interest is the extension of the CM-theory to include the application of the synthesis procedures to the structural CM description as well. It can be illustrated by the following considerations:

1. If a CM description makes use of recursive structures, the unfolding process may be incomplete.

2. Provided recursions are forbidden, the size of the unfolded model may be exponentially dependent on the initial CM definition.

3. Unfolding will further complicate matters when a synthesized program is to be analyzed in terms of a given CM description because it will give rise to a great number of auxiliary elements, making the debugging of the CM definition rather difficult.

The foregoing disadvantages have been pointed out by many authors (see e.g. , Ref. 14). As an alternative, some subset of the predicate calculus is generally employed to avoid. Our approach is based on the structural computational model formalism introduced elsewhere . The resulting CM is a finite collection of relationship schemes of the form $M = \{T_1,...,T_m\}$, where T_i (i = l,..., m) defines a nontrivial SA object structure. For a heterogeneous object, a list of its component names (scheme elements) and a set of computability sentences (CS) of the form F: $A \rightarrow x$ over the elements are to be fixed. CS imply that applying the program term F to the values of A-elements will yield the values of x-elements. So far the relationship scheme has proved to be similar to the standard CM. The resemblance will vanish as soon as another way to define element types appears, associating scheme elements with certain relationship schemes of the same model. This mechanism leads to a CM that adequately reflects the hierarchy of the original SA concepts. Note that recursive object definitions are used in this formalism.

The information received from the CM description in combination with the problem statement represents a specification utilized by the PSSS program generator for target program synthesis. The programs thus built involve both branches and recursion. The synthesis procedures are sufficiently effective to be used interactively. Up to this point, the project in question has had no particular database management system. Fortunately, the functional architecture (see Figure 3) does allow atmospheric optics problems to be solved by means of the available databases. Nevertheless, this by no means suggests that the development of special PSSS databases would be an unwarranted luxury.

REFERENCES

1. V.E. Zuev, Opt. Atm. **1**, 5 (1988).
2. O.K. Voitsekhovskaya, V.E. Zuev, and VI.G. Tyuterev, Opt. Atm. **1**, 3 (1988).
3. V.S. Komarov, A.A. Mitsel', S.A. Mikhailov, Yu.N. Ponomarev, V.P. Rudenko and K.M. Firsov, Opt. Atm. **1**, 84 (1988).
4. F.P. Brooks, The Mythical Man Month, Addison-Wesley Publishing Company, Inc., Reading, Mass. (1975).
5. J.M. Fox, Software and its Development, Prentice-Hall, Inc., Englewood Cliffs, N. Y. (1982).
6. V.N. Agafonov, ed., *Requirements and Specifications in Program Development,* (Mir, Moscow, 1984).
7. S.S. Lavrov, *Knowledge Representation and Application in Automated Systems* Mikroprotsessornye sredstva i sistemy,, **14** (1986).
8. J.L. Olty and M.J. Coombs, Expert Systems. Concepts and Examples (Published by NCC Publications, 1984).
9. A. Colmerauer, H. Kanoui, R. Pasero and P. Roussel, *Un system de Communication Homme-Machine en Francais, Research report. Groupe Intelligence Artificiel* (Universite Aix Marseille **11**, 1973).
10. I.O. Babayev, F.A. Novikov and T.I. Petrushina, *Descartes-Input Language of SPORA-System Applied Informatics* (Nauka, Moscow, 1984).
11. E.H. Tyugu, *Conceptual Programming* (Nauka, Moscow, 1984).
12. M.I. Kakhro, A.P. Kalja and E.H. Tyugu ES EVM (PRIZ) Hardware Programming System (Finansy i statistlka, Moskow, 1981).
13. E.H. Tyugu, *Solving Problems by Computational Models,* Zhur. Vychislitel'noi matematiki i matematicheskoi fiziki **10**, 716 (1970).
14. V.S. Neiman, Program Synthesis for Descriptions of Recursive Relation in: (Sintez programm, Ustinov, 1985).
15. V.B. Novosel'tsev, Structural Computational Models, Formal Basis, in: (Sintez program, Ustinov, 1985).