DEUTSCHES ELEKTRONEN-SYNCHROTRON

DESY 94-048
March 1994

Further Development of the ZEUS EXPERT SYSTEM: Computer Science Foundations of Design

M. Flasiński

Deutsches Elektronen-Synchrotron DESY, Hamburg

ISSN 0418-9833

NOTKESTRASSE 85 - 22603 HAMBURG

To be sure that your preprints are promptly included in the
HIGH ENERGY PHYSICS INDEX,
send them to (if possible by air mail):

DESY **Bibliothek** **Notkestraße 85** **22603 Hamburg** **Germany**	**DESY-IfH** **Bibliothek** **Platanenallee 6** **15738 Zeuthen** **Germany**

DESY 94-048
March 1994

ISSN 0418-9833

Further Development of the ZEUS EXPERT SYSTEM: Computer Science Foundations of Design

Mariusz Flasiński *
DESY

March 21, 1994

Abstract

The prototype version of the ZEUS Expert System, ZEXP, was diagnosing selected aspects of DAQ System during ZEUS running in 1993. In November 1993 ZEUS decided to extend its scope in order to cover all crucial aspects of operating the ZEUS detector (Run Control, Slow Control, Data Acquisition performance and Data Quality Monitoring). The paper summarizes fundamental assumptions concerning the design of the final version of the ZEUS Expert System, ZEX. Although the theoretical background material relates primarily to ZEX, its elements can be used for constructing other expert systems for HEP experiments.

1 Introduction

In July 1992, ZEUS decided to build an Eventbuilder (EVB) Expert System. As the first step towards a full expert system it was decided to enhance the EVB monitoring system sending messages to Run Control. Based on the positive experience with this system the design of a full blown expert system for the operation of ZEUS was started. Software systems based on *Artificial Intelligence (AI)* techniques were studied for their suitability in high energy physics experiments. At the same time, the reasearch focusing on a development of AI models that could meet hard real-time constraints imposed by such experiments was carried out.

In April 1993, the ZEUS Expert System Prototype, ZEXP [BFH92, BFHO93a], was embedded in the ZEUS data acquisition (DAQ) system. ZEXP diagnosed pre-selected aspects of the experiment during the 1993 ZEUS running. The system delivered reliable diagnostics of the DAQ system performance in the EVB environment by exhaustive monitoring of the EVB interfaces to DAQ computers of: detector components and trigger systems[BFHO93b]. In order to receive a feedback from the users (e.g. shift crews) on the usefulness of AI-based software tools covering a wide spectrum of experiment running activities, some additional services had to be offered, see Fig. 1b. For example, ZEXP made checks on the status of the readout system (affecting data quality). The checking was limited to the links between the EVB and the components and between the EVB and the Third Level Trigger (TLT) farm. Other aspects of running the experiment like, e.g. monitoring the RC Run Sequencing status and the rates of the Global

*On leave from the Department of Computer Science, Jagiellonian University, Cracow, Poland.

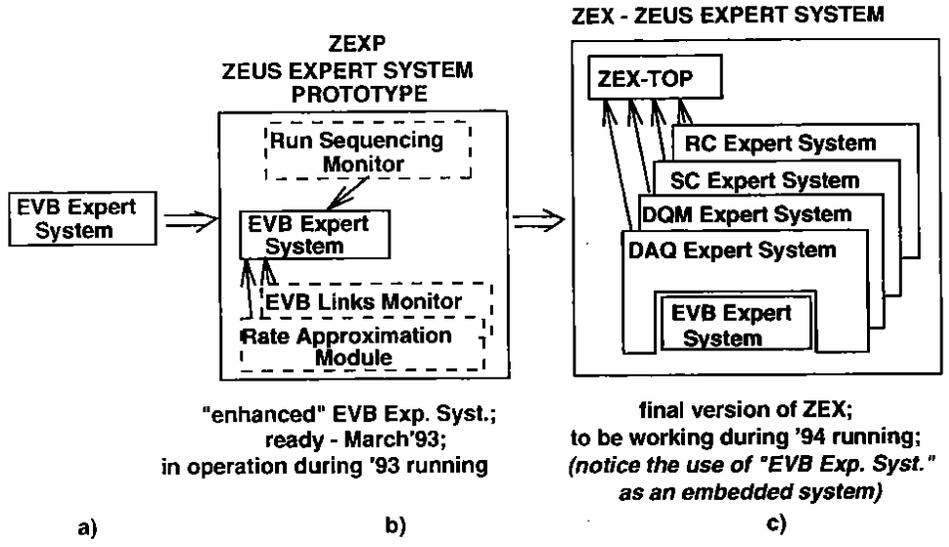

Figure 1: The development of the ZEUS Expert System.

First and Second Level Trigger systems (GFLT, GSLT) were deduced approximately from information available in the EVB [BFHO93b]. ZEXP fulfilled its "prototypical" role helping potential users to discover new additional features of a system useful for running the experiment [Dosetal94].

The problem of applicability of *AI reasoning techniques* in the context of hard real-time constraints was also verified positively [BFHO93a]. The good experience with combined syntactic [Fla88, Fla89] and semantic [FlaLew91] schemes for recognizing distributed system status and its resource capabilities [FlaKot89, FlaKot92] observed previously, indicated that such a mixed scheme might be promising also in this case. Some additional research work had to be done, especially in the *syntactic* pattern recognition area [Fla94]. (The use of well-known *discriminant* pattern recognition schemes for our purpose was discussed in [BFH92].)

In November 1993, the development and run-time platforms of a Talarian *RTworks* [Tal92] expert system shell became available to extend the expert system scope (in order to cover all the crucial aspects of ZEUS running: Run Control, Slow Control, Data Acquisition performance and Data Quality Monitoring [BFHO93c]). The original code, Finite State Automata-based EVB Expert System, was extracted from the prototype version and embedded in the final system, see Fig. 1c. The remainder of the prototype ("prototype simulators" mentioned above) is being replaced by ZEX RC-, SC-, DAQ-, and DQM-subsystems implemented with *RTworks* shell.

The much broader scope of ZEX and the use of a new AI-based development platform (*RTworks* supports rule-based programming) required methodological studies in order to layout the design for ZEX.

The present paper summarizes the computer science background of the ZEUS Expert System design. A general methodology for designing real-time expert/control systems for HEP experiments is described. Finally, the paper can serve as an introduction and a guide to those areas of computer science, which are used for constructing such systems. The basic notions concerning rule-based expert systems are presented in Appendix A, whereas fundamental ideas of theory of formal languages and automata are introduced in Appendix C. The considerations concerning the system design are illustrated with diagrams using the Booch Object-Oriented Design [Boo91] notation. Readers, who are not familiar with the OO approach are strongly adviced to read first Appendix B.

2 ZEX Architecture - Artificial Intelligence Aspect

2.1 ZEX as an analytic expert system

Clancey in [Cla85] introduced a classification of expert systems according to *generic operations* performed by an expert system with respect to a real-world (mechanical, electrical, computerized, etc.) system. He distinguishes between *synthesizing expert systems* supporting construction of real-world systems and *analytic expert systems* interpreting real-world systems behaviour. There are four generic operations of an analytic expert system:

- *monitor* status of a real-world system (detect its discrepant behaviour),
- *diagnose* faulty components of a monitored system (explain its discrepant behaviour),
- *predict* consequences of a specific bahaviour of a monitored system,
- *control* a monitored system, i.e. determine actions forcing a desired behaviour of a monitored system.

The design of the ZEUS Expert System (ZEX) focuses on delivering these generic operations.

2.2 Blackboard approach

From the computer science point of view, ZEUS can be considered as a complex distributed computer system that can be decomposed according to various criteria, e.g. a component-oriented criterion (BCAL, CTD, etc) or a purpose-oriented criterion (Slow Control subsystems, Run Control susbsystems, etc). Consequently, both its (static) structure and its behaviour create a multidimensional hierarchy. Since the ZEUS Expert System should cover as wide scope of the ZEUS activity as possible [BFHO93a], ZEX inherits this complexity. Because ZEX is a (hard) *real-time expert system*, the choice of an efficient artificial intelligence-based technique is a crucial decision for preparing the system.

The so-called *Blackboard approach* was introduced by Hayes-Roth [Ermetal80], and developed to be one of the most powerful techniques of design of expert systems in the eighties [EngMor88]. In this approach a system is partitioned into (see Fig. 3):

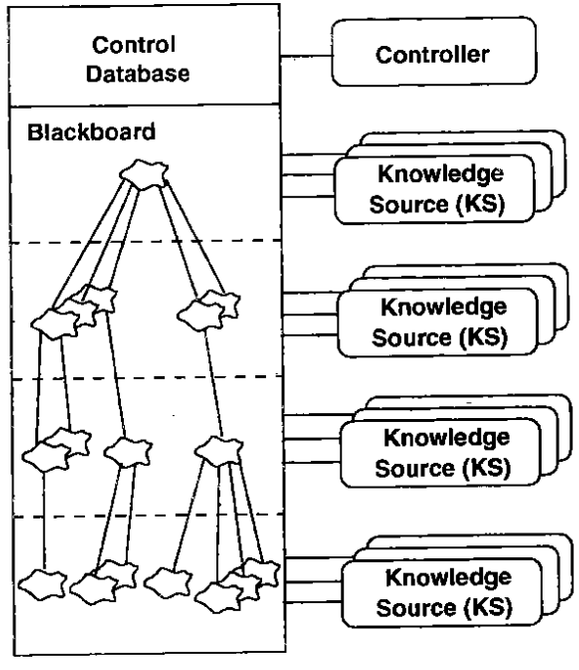

Figure 2: A Blackboard Architecture scheme.

- a global hierarchical data structure of a *solution space* (see Appendix A), called *the Blackboard*,
- independent, hierarchically organized *Knowledge Sources (KS)*, which contain *problem-solving knowledge*, and are running under *the Controller* (which is a special KS containing control knowledge, i.e. knowledge allowing the system to: determine the focus of attention while reasoning, schedule KSs acceess to the Blackboard, etc).

This approach is very efficient for applications, in which the problem space not only is large, but also factorable along some number of dimensions. Then, the objects of the Blackboard are hierarchically organized into information levels corresponding to levels of data processing. (The approach is analogical to the syntactic/structural approach to pattern recognition, in which recognized complex phenomena are decomposed into hierarchies of simpler and simpler structures, which are easier to be analyzed [Fu82].) Since, this approach has proven to be space- and time-efficient in comparison with alternate approaches [EngMor88], the design of the ZEX architecture is based on the Blackboard approach.

3 ZEX Design - Object-Oriented Methodology

3.1 Object-Oriented logical design

The Object-Oriented (OO) approach emerged in the early seventities at the same time in the fields of: software engineering and Artificial Intelligence (AI). (In AI it was connected with researches into representation and processing of knowledge [Min75].) One can easily find elements of the OO aproach in the definition of the blackboard system introduced above. In fact, both a global data structure (Blackboard) and a procedural part ("active knowledge" in the form of KSs) of a blackboard system are formulated at different levels of *abstraction*, creating a multidimensional *hierarchy* (for the OO meaning of this terms, see Appendix B). "Active knowledge" itself is *modularized* and *encapsulated* into Knowledge Sources. Therefore, the OO approach seems to be especially suitable for design of blackboard systems.

Let us consider the top-level OO decomposition of ZEX. In Fig. 3 we see three main parts of ZEX:

- ZEX Data Monitor, which acquires, filters, preprocesses and converts monitored-data transmitted by Monitoring Subsystems of ZEUS before it is sent to ZEX Inference Engine,
- ZEX Inference Engine analyzes monitored-data in real-time, then delivers: ZEUS-behaviour-interpretation in the form of a *(status, diagnosis, prediction)*-triple and proposed-control-action to a shift crew,
- ZEX User Interface is used to provide *point-and-click* colour graphics views displaying histograms, ZEX messages, situation diagrams, etc.

The object decomposition at this level is shown in Fig. 4 with an *object diagram* (see Appendix B). Data monitor objects (ZEX interfaces to ZEUS monitoring subsystems, like SC, DQM, etc.) are activated by the knowledge sources object to transmit monitored data to ZEX. Each knowledge source, which is active sends two kinds of messages to the blackboard object:

- provide-solution-space-results - to get partial solutions and/or input data, which are necessary for reasoning,
- record-solution-space-results - to contribute to the blackboard with reasoned results.

The important results of the system reasoning are sent to user-interface objects, which correspond to ZEX views.

The OO approach turned out to be very beneficial for the logical design of blackboard-based ZEUS Expert System[1].

[1]Nota bene, the two best-known blackboard systems, BB1 (Stanford) [JohHay86] and BLOB (air defense, the British Ministry of Defence) [EngMor88], were developed with the help of OO design.

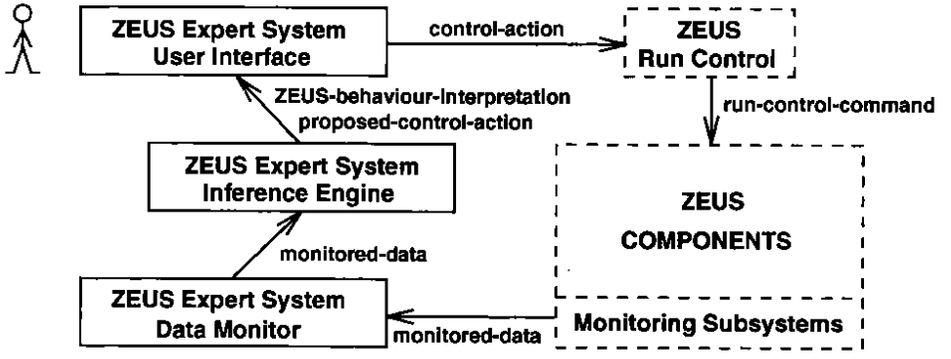

Figure 3: ZEUS Expert System Block Diagram.

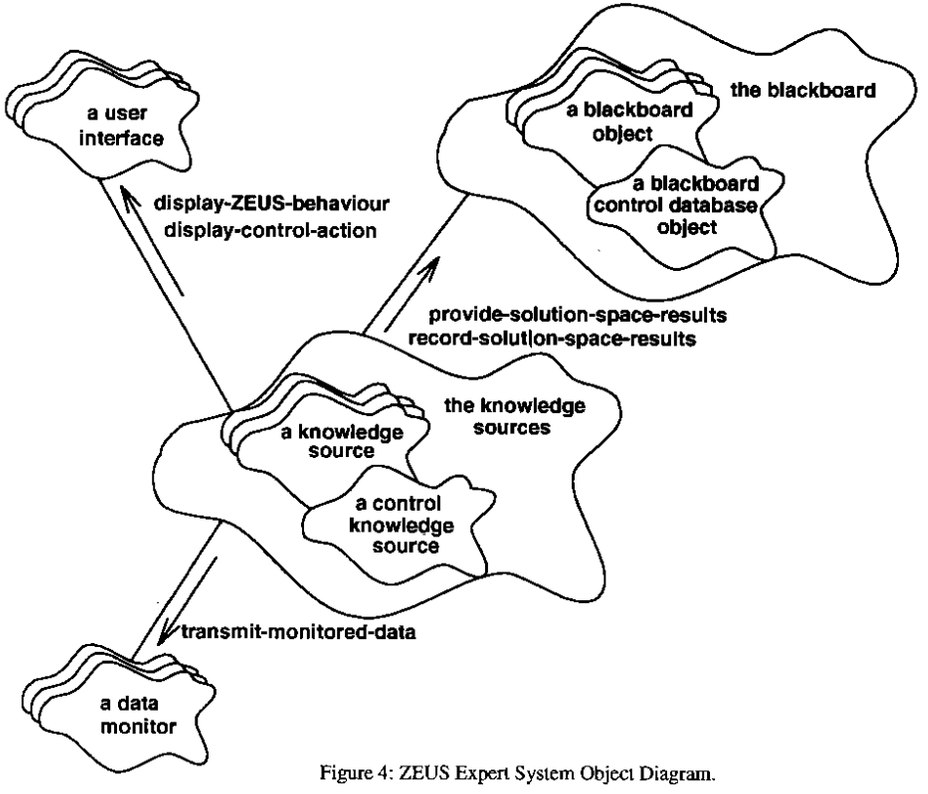

Figure 4: ZEUS Expert System Object Diagram.

3.2 Frame-based design of ZEX data structures

The abstract representation of the *expert system knowledge* is a combination of *data structures* and *interpretation procedures* (see Appendix A). In ZEX, data structures are stored in the blackboard (see Fig. 5). Thus, on the one hand, all of them are treated just as objects of the class **Blackboard Object**. On the other hand, they form a hierarchical structure of classes (system $\longrightarrow$ subsystems $\longrightarrow$ components), which is shown in Fig. 5.

Objects belonging to the same class should be characterizable by the (more or less) same structure used for their description. Of course, there may be some exceptions. In general, however, for each class we should be able to define a template of a structure keeping descriptive object information. This structure template, called a *frame* [Min75], is used as a fundamental building block for representing **blackboard objects** at the implementation level of ZEX.

The OO approach allowed us to systematize design assumptions for existing systems, which interface with ZEX. For example, the use of the *hierarchy principle* enabled us to define a new object structure for Slow Control *monitored data*. Now, they are viewed as a combination of:

- **Readout Data** corresponding to the detector components under control (e.g. a readout of a CTD high voltage status), and
- **Readout Medium Data** corresponding to the slow control computers monitoring the detector components (e.g. a readout of a status of crates monitoring a CTD high voltage status), see Fig. 6.

Although both aspects, obviously, relate to the same SC components (in our example: CTD-HV), such a structural decomposition of monitored data (into levels and meta-levels) makes the reasoning in ZEX more effective, especially in cases of malfunctioning of **Readout Media** (SC-related channels, crates, and systems).
Then, if Slow Control **Readout Data** are concerned, the further decomposition is made, this time according to the *abstraction principle* (see Appendix B.2). Thus, we have:

- **Physics Phenomenon - Related Readout Data** being typically of the continuous nature (e.g. beam-related information), and
- **Detector Component - Related Readout Data** being typically of the discrete nature (e.g. a number and a topology of dead (bad) channels of the calorimeter).

The real-time expert systems should be able to reason over data treated as a function of time. For this reason, the frames keeping **Monitored Data** should allow one to store data in ring buffers. ZEX frames support two kinds of variables: **Snapshot - Measured Variables** and **Time Series - Measured Variables**, as shown in Fig. 7. **Time Series - Measured Variables** are kept in **Ring Buffer** structures, the records of which are time-stamped.

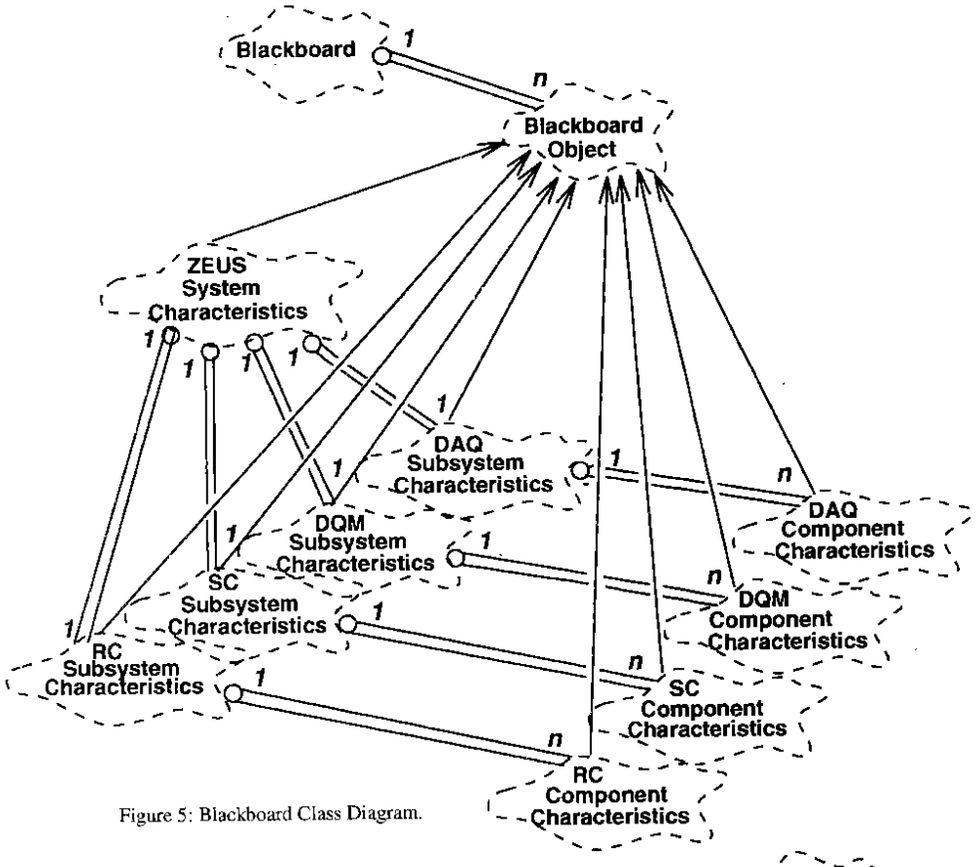

Figure 5: Blackboard Class Diagram.

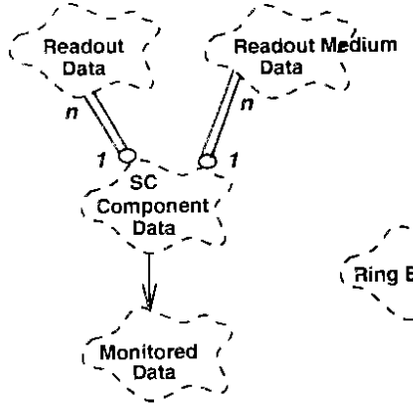

Figure 6: SC Component Data Class Diagram.

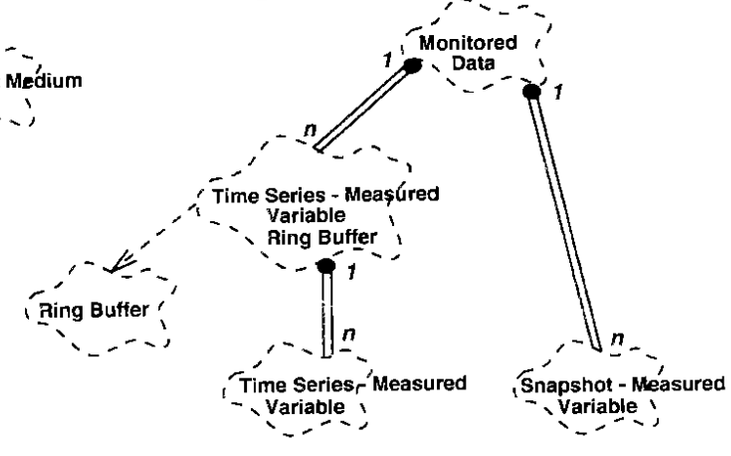

Figure 7: Monitored Data Class Diagram.

4 ZEX Interpretive Procedures - Formal Grammar Model

The *interpretive procedures* of both ZEXP and ZEX are based on a *production system (formal grammar)* model (see appendices: A and C). In ZEXP, *finite state automata, FSA* were used as pattern recognizers, whereas in ZEX, both *production systems (rule-based systems)* and *FSA*-based pattern recognizers are used as Knowledge Sources (see Fig. 8). Now, we will discuss shortly these two models.

4.1 Rule-based interpretive procedures

A production system [Broetal86] (see Fig. 9) consists of:

- a working memory containing input data and local variables keeping e.g. intermediate results, and
- a set of rules of the form:

$$\textbf{if} < antecedent > \textbf{then} < consequent >.$$

In the field of expert systems an $< antecedent >$-$< consequent >$ pair is called a $< condition >$-$< action >$ pair because expert system rules are usually used to encode associations between patterns of data from the working memory of the system and actions that the system should perform as consequence.

Expert system production rules are a formalism developed on the basis of the *theory of formal languages and automata* (see Appendix C). In fact, a rule set serves for manipulating symbols, and its theoretical model are simply *formal grammars*. The rule can be expressed as a *grammar production*:

$$< antecedent > \longrightarrow < consequent >,$$

where $< antecedent >$ and $< consequent >$ correspond to left- and right- sides (respectively) of the production.

In "strong" rule-based systems, we can:

- operate on multidimensional structures (e.g. trees, graphs) rather than only linear structures (e.g. lists),
- define the contexts, in which some rules can be activated and others cannot (the conditions of a rule activation),
- use special functions/external actions, which are called/invoked, if a rule is applied.

In terms of formal linguistics, we talk (respectively) about [FlaKot92, Fla94]:

- *structures* of left- and right- sides of a production instead of words over symbols,

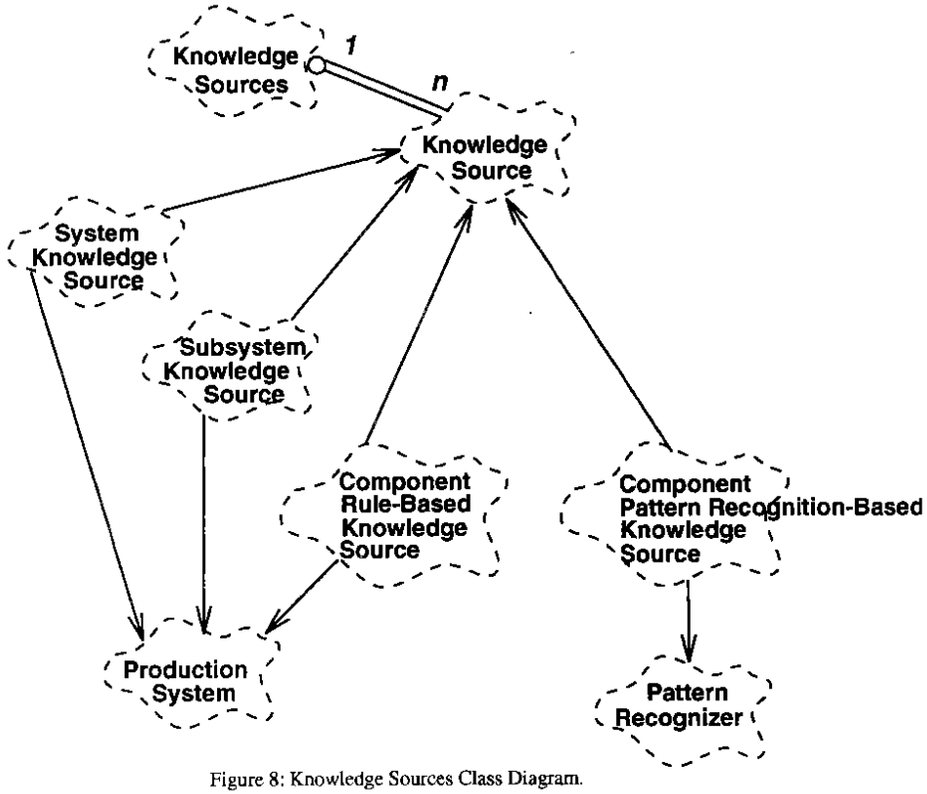

Figure 8: Knowledge Sources Class Diagram.

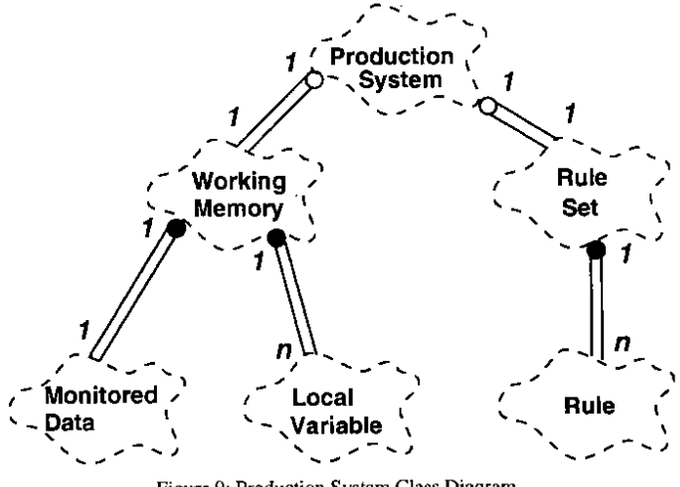

Figure 9: Production System Class Diagram..

- *a predicate of the production applicability*, and
- *a set of functions* associated with a production.

For such "strong" rule-based systems, a *programmed grammar* [FlaKot92, Fla94] is a formal model. It is defined in the following way.

Definition 1

A *programmed structure grammar* is a quadruple

$$G = (\Sigma, \Delta, P, S)$$

where Σ is a finite, nonempty set of nonterminal (auxiliary) symbols, called a *nonterminal alphabet*,
Δ is a finite nonempty set of terminal symbols, called a *terminal alphabet*,
P is a finite set of productions of the form (L, R, π, F), in which
L is the structure of the left-hand side of the production,
R is the structure of the right-hand side of the production,
$\pi : A_L \longrightarrow \{TRUE, FALSE\}$ is the predicate of the production applicability, A_L is the set of attributes of the structure L,
F is the set of functions associated with the production,
S is the initial structure, called the *axiom.*

Programmed structure grammars are a formal model for rule-based Knowledge Sources of ZEX.

4.2 Pattern recognition-based interpretive procedures

When using rule-based systems, we need only to define rules. The task of their interpretation is fulfilled by a rule interpreter that decides when to apply which rules. However, sometimes such a general (so not optimum) tool is not recommended because of its small time efficiency.

In the presence of hard real-time constraints, a syntactic pattern recognizer can be used at lower levels of a hierarchy of interpretive procedures [Fla94]. It consists of (see Fig. 10):

- a *signal processor* filtering monitored data (it smoothes data treated as sampled signals),
- a *cluster classifier*, which classifies, at discrete moments, an observed phenomenon (via observed data) to one of the pre-defined classes (clusters) and codes the class with a symbol,
- an *automaton* (see Appendix C), which reads a string of these symbols (treated as a word of some language) and recognizes the state of the phenomenon in time-series (it also informs the user about the time-series recognition with each change of the state of the phenomenon).

A combination of a signal processor and a cluster classifier is used in real-time expert systems, and it was discussed in [BFH92]. The automata are used in ZEX as a kind of specialized (so optimum) rule interpreters. For the same reasons as those mentioned in Section 4.1, the definition of an automaton has to be "enhanced" (cf. Appendix C), by:

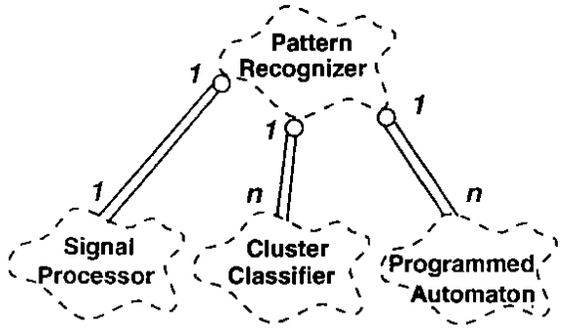

Figure 10: Pattern Recognizer Class Diagram

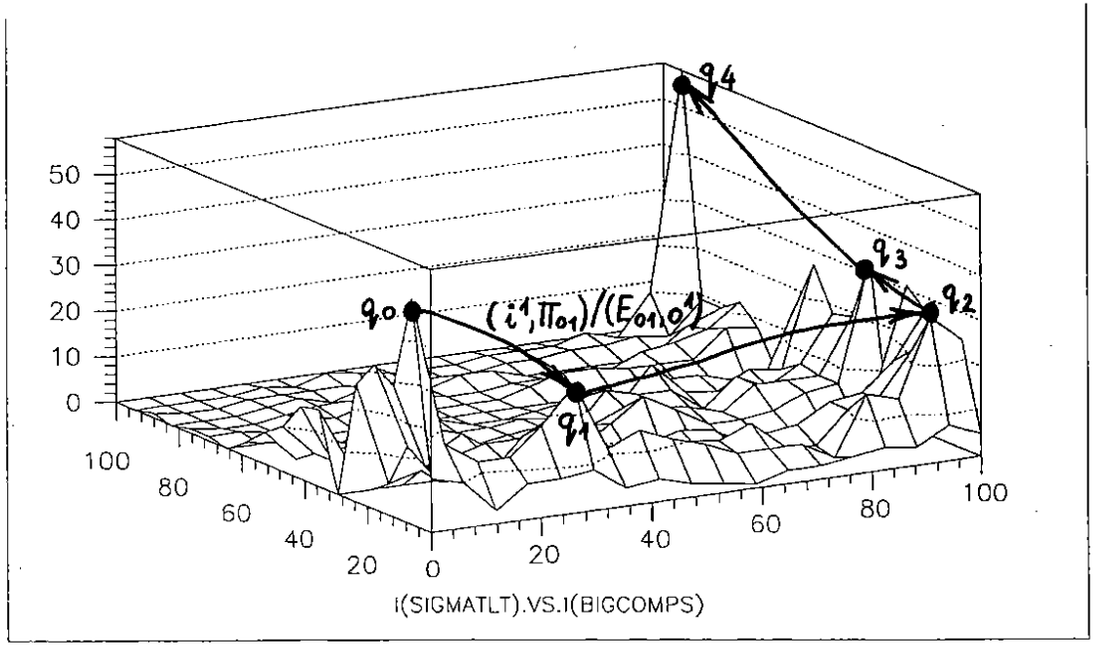

Figure 11: Example of transitions in a programmed automaton.

- allowing it to operate on complex structures,
- making transitions dependent on contexts, and
- defining special functions/actions.

Definition 2
A *programmed automaton* is a six-tuple

$$A = (Q, I, O, M, \delta, q_0)$$

where Q is a finite set of states,
$I = I_{\Delta_1}$ is a finite set of input structures over the terminal alphabet Δ_1,
$O = O_{\Delta_2}$ is a finite set of output structures over the terminal alphabet Δ_2,
M is a finite set of memory objects,
$\delta : Q \times I^n \times \Pi \longrightarrow Q \times E \times O$ is the transition function, in which
$\Pi : A_I \cup A_M \longrightarrow \{TRUE, FALSE\}$ is the predicate of the transition permission, A_I, A_M are sets of attributes of the input structure I and memory objects M (respectively),
$E : A_O \cup A_M \longrightarrow V$ is a set of evaluation functions, A_O is the set of the output structure O, and V is the set of admissible values,
$q_0 \in Q$ is the initial state.

An example of the work of one of the ZEX automata is shown schematically in Fig. 11. The automaton recognizes the status of EVB interface buffers. (The figure shows clusters in a feature space defined with a total buffers' load of big components, $I(BIGCOMPS)$, and total buffers' load of TLT interface transputers, $I(SIGMATLT)$.) The distinguished states, corresponding to feature space cluster centers, are defined in the following way:

- q_0 - normal buffers' status,
- q_1 - TLT buffers are being filled,
- q_2 - TLT buffers are full,
- q_3 - TLT buffers are full and big components' buffers are being filled,
- q_4 - the system is blocked.

A transition from q_k to q_m is defined with an item of the form:

$$(i^a, \Pi_{km})/(E_{km}, o^b),$$

where i^a is an input symbol pointing to a possibility of activating a transition,
Π_{km} defines additional conditions that have to be fulfilled to activate a transition,
E_{km} defines functions (actions) associated with a transition, and
o^b defines an output symbol (message) issued by an automaton after a transition[2].

[2]There exists an analogy between programmed automata and Markov chains, if we assume that the predicate of the transition permission Π gives the probability of a transition instead of a *TRUE/FALSE* value (fuzzy logic).

5 Control of ZEX Reasoning - Graph Grammar Model

As we have mentioned in Section 2.2, Knowledge Sources, of the blackboard system run under a supervision of the *Controller*. The main task of the Controller is to decide which Knowledge Sources (set of rules, or pattern recognizers) should be activated at a given moment, and it determines the order of the Knowledge Sources activation. In ZEX, the Controller plays two roles:

1. It keeps and updates the structure, called the *Situation Graph*, describing a line of reasoning, which can be (post mortem) used for e.g. giving an explanation of a diagnosed system (i.e. ZEUS) failure.
2. It keeps and updates the structure, called the *Knowledge Sources Activation Structure*, which allows ZEX to determine the focus of attention during on-line monitoring of a diagnosed system (ZEUS) working.

Ad 1. An example of a Situation Graph is shown in Fig. 12. Its nodes (vertices) represent System (ZEUS), Subsystems (Data Quality Monitoring - DQM, Slow Control - SC, Run Control - RC, Data Acquisition System - DAQ) and aspects of these Subsystems (BEAM-BGR (beam background), CTD-HV (high voltage power supply of the Central Tracking Detector), GFLT (the Global First Level Trigger), etc.). Its edges describe the current *cause-result* relationships between these aspects. Graphs like the one shown in Fig. 12, were used for the control of allocation processes in distributed computer systems [FlaKot89, FlaKot92]. They differ from standard digraphs (directed graphs) in:

- associating attributes to nodes and edges in order to "enhance" their descriptive power,
- imposing certain restrictions to their structure in order to make them parsable/analyzable efficiently [Fla88, Fla89, Fla93].

Let us introduce their formal definition.

Definition 3
An attributed indexed edge-unambiguous graph, IE graph, over Δ and Γ is a seven-tuple

$$H = (V, E, \Delta, \Gamma, \phi, \alpha, \beta)$$

where V is a finite, non-empty set of nodes that indices have been ascribed to in an unambiguous way,
Δ is a finite, non-empty set of node labels,
Γ is a finite, non-empty set of edge labels,
E is a set of edges of the form (v, λ, w), where $v, w \in V$, $\lambda \in \Gamma$, such that index of v is less than index of w,

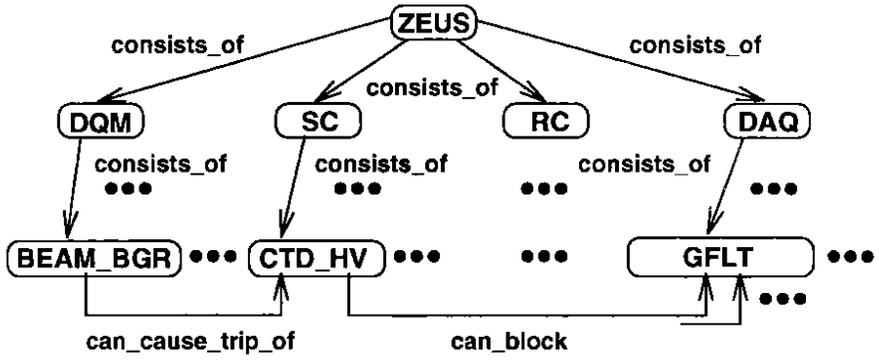

Figure 12: The initial Situation Graph.

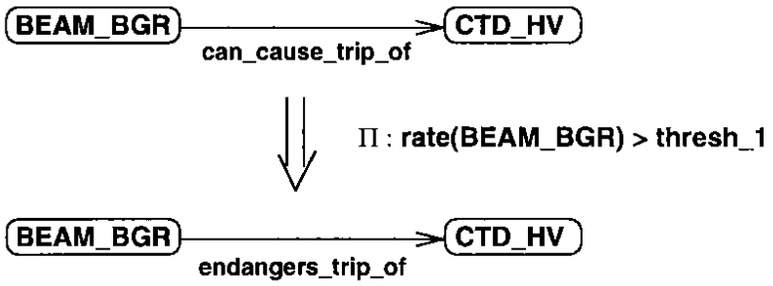

Figure 13: The graph grammar production applied, if a high background noticed.

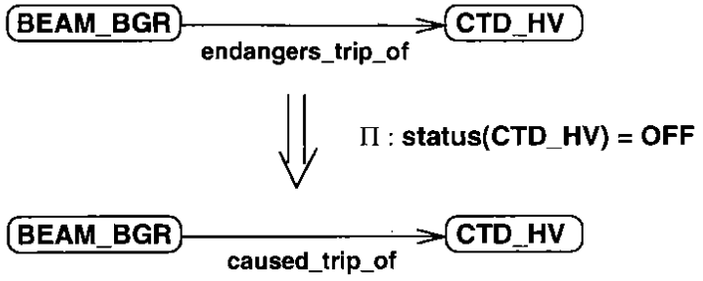

Figure 14: The graph grammar production applied, if the CTD high voltage tripped.

$\phi : V \longrightarrow \Delta$ is a node-labelling function,
α is a finite set of functions attributing nodes,
β is a finite set of functions attributing edges.

In order to update information contained in the Situation Graph, we define rules that will modify the graph according to specific conditions. For example, a rule activated in a presence of a high background is shown in Fig. 13. The predicate of applicability of the rule Π defines the condition for a rule activation. Three rules shown in Figs 13-15 describe one of the possible consequences of a high beam background. Now, if the following sequence of events is monitored by ZEX:

- a beam background rate is higher than **thresh-1**, see Π in Fig. 13,
- then CTD-HV trips (a CTD HV "trip" is a CTD HV power supply failure), see Π in Fig. 14, and
- finally a GFLT rate dies,

then the rules shown in Figs 13-15 are applied subsequently to the Situation Graph. If, for example, the run is aborted, ZEX keeps track of what was going on before the failure. This information can be used for reasoning.

The set of rules acting on graph structures is defined formally as an attributed programmed graph grammar [Fla88, FlaKot89, FlaKot92] belonging to the edNLC family [Janetal82].

Definition 4

An attributed programmed edNLC graph grammar is a quadruple

$$G = (\Delta, \Gamma, P, Z)$$

where Δ is a finite, non-empty set of node labels,
Γ is a finite, non-empty set of edge labels,
P is a finite set of productions of the form (L, R, π, C, F), in which
L and R are left- and right- side digraphs having labelled nodes and edges,
π is the predicate of the production applicability,
$C : \Gamma \times \{in, out\} \longrightarrow 2^{\Delta \times \Delta \times \Gamma \times \{in,out\}}$ is the so-called embedding transformation (it is a formula describing a way of embedding the graph R in a transformed graph [Fla88]),
F is the set of functions associated with the production,
Z is the initial graph of the same type as L, called the *axiom*.

Ad 2. The same formalism and technique can be used for focusing the ZEX attention during on-line monitoring. Knowledge Source Activation Structure shown in Fig. 16 keeps the information allowing ZEX to activate certain Knowledge Sources according to an "active" status of other Knowledge Sources. The possible activation links (graph edges) are fixed in the structure. If an edge goes from KS-m to KS-n, then it means that whenever KS-m is active also KS-n should be activated (see Fig. 17).

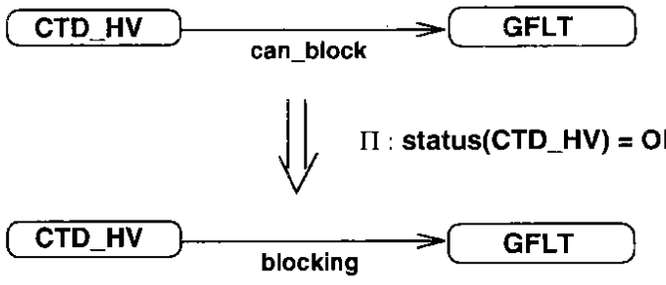

Figure 15: The graph grammar production applied, if the CTD high voltage tripped.

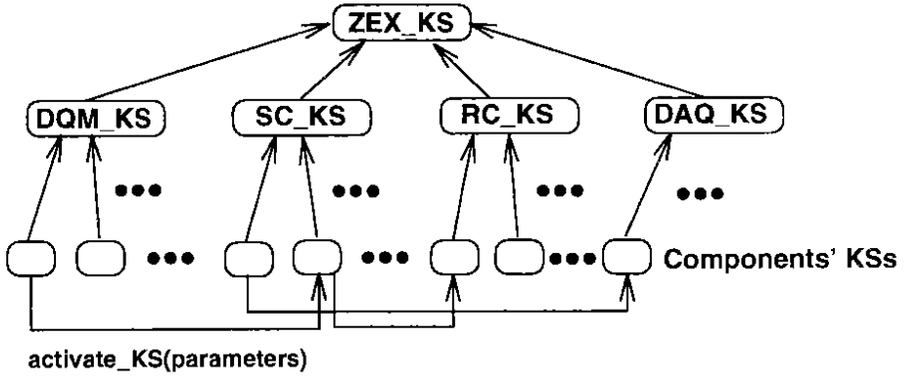

Figure 16: Knowledge Source Activation Scheme..

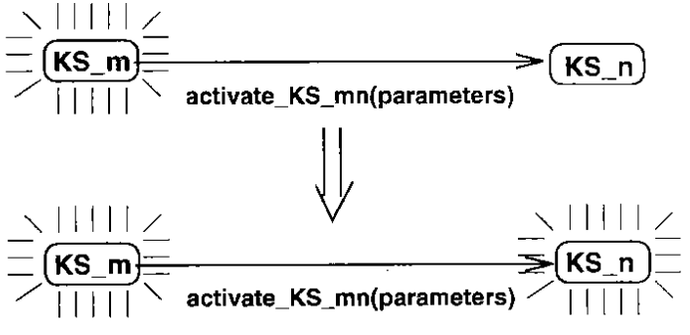

Figure 17: The graph grammar production applied for activating a Knowledge Source.

6 Some Issues Concerning ZEX Physical Design

The class diagram of ZEX is shown in Fig. 18. It depicts top-layer classes defined in the system, and discussed in the paper. At the implementation level these classes are mapped into basic *RTworks* shell [Tal92] processes (see Fig. 19), namely:

RTdaq , which in real-time acquires, filters, and groups the monitored data and sends it to the **RTserver**, which, in turn, distributes it to other processes,

RTie , the inference engine, which reasons (also in real-time) using: rule-based inferencing, statistical functions, trending functions, etc.; it uses working memories of Knowledge Sources as their private stores and the Blackboard as the global memory; it also calls Knowledge Sources implemented as pattern recognizers,

RThci , which provides a sophisticated point-and-click graphical user interface to a shift crew.

Summarizing, the newly-defined broad scope of the ZEUS Expert System activity has required fulfilling the following conditions:

- preparing sound theoretical computer science models, which can handle both the ZEUS complexity and real-time constraints,
- using a proper (for the task defined) and powerful software tool for the system implementation,
- a very accurate logical design, which maps these theoretical models into a physical (implementation) model of the sytem in such a way that it makes use of power of both the software tool and formal models.

The formal models have been prepared on the basis of the theory of artificial intelligence and pattern recognition, and then they were mapped to the physical model with the help of the Object-Oriented methodology. It seems that *RTworks* shell is powerful enough to fulfill the end-user requirements defined for the final version of the ZEUS Expert System. The further results of the project will be published in a subsequent paper.

Acknowledgement

The author thanks Dr. G. Wolf for a support of all the phases of the ZEUS Expert System Project, and for critical reading of the paper. I am, also, very grateful to Prof. E. Lohrmann for the support. I want to thank my EVB group colleagues, U. Behrens, L. Hagge and K. Ohrenberg, for very valuable discussions concerning the ZEX design.

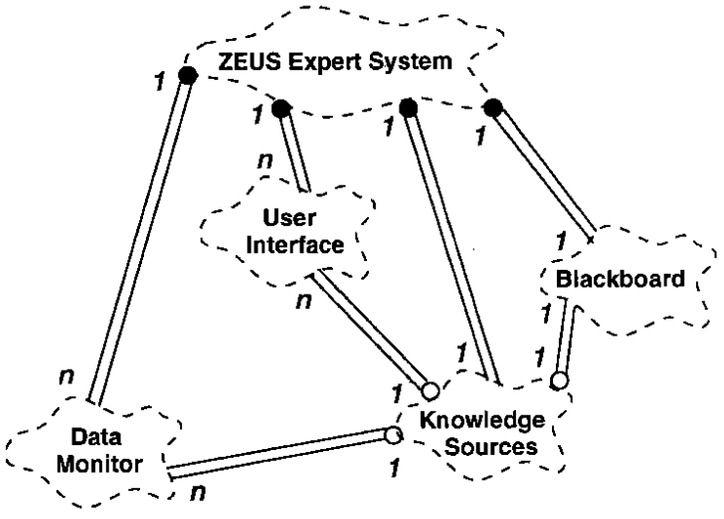

Figure 18: Top-Level Class Diagram.

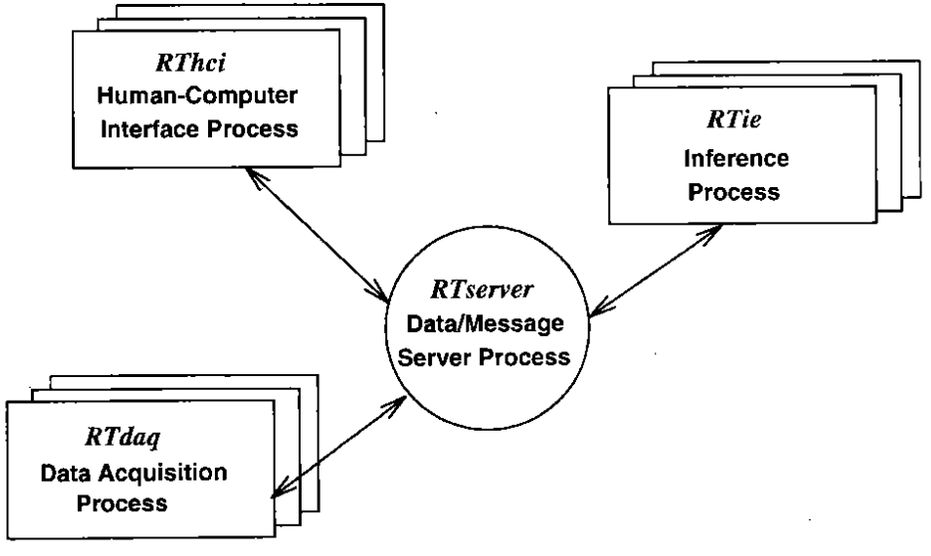

Figure 19: Diagram of RTworks basic processes.

References

[BFH92] U. Behrens, M. Flasiński, L. Hagge, ZEXP - Expert System for ZEUS. Problem analysis, theoretical background and first results, *DESY Reports*, DESY 92-141, October 1992.

[BFHO93a] U. Behrens, M. Flasiński, L. Hagge, K. Ohrenberg, ZEX - An Expert System for ZEUS, *Proc. 8th Conf. Real-Time Comp. Appl. Nucl. Particle Plasma Phys.*, June 8-11, 1993, Vancouver, Canada, 168-173.

[BFHO93b] U. Behrens, M. Flasiński, L. Hagge, K. Ohrenberg, Status and prospects of an expert system for ZEUS, *ZEUS - Note* **93-099**, September 1993.

[BFHO93c] U. Behrens, M. Flasiński, L. Hagge, K. Ohrenberg, ZEX - an expert system for ZEUS, *Proc. 3rd Int. Workshop Software Eng., Artif. Intell., Expert Syst. for High Energy and Nucl. Phys.*, October 4-6, 1993, Oberammergau, Germany.

[Boo91] G. Booch, *Object Oriented Design with Applications*, The Benjamin/Cummings Publ. Comp., 1991.

[Broetal86] L. Brownston, R. Farell, E. Kant, N. Martin, *Programming Expert Systems in OPS5. An Introduction to Rule-Based Programming*, Addison-Wesley, Reading, MA, 1986.

[Cho57] N. Chomsky, *Syntactic Structures*, The Hague, Mouton, 1957.

[Cla85] W.J. Clancey, Heuristic classification, *Artificial Intelligence* **27** (1985), 289-350.

[Dosetal94] U. Dosselli, D. Hanna, G. Iacobucci, L. Labarga, ZEUS 1993: Run Coordinators report, *ZEUS - Note* **94-005**, January 1994.

[EngMor88] R. Englemore, T. Morgan (eds), *Blackboard Systems*, Reading MA, Addison-Wesley, 1988.

[Ermetal80] L.D Erman, F. Hayes-Roth, V.R. Lesser D.R. Reddy, The Hearsay-II speech understanding system: integrating knowledge to resolve uncertainty, *Computing Surveys* **12** (1980), 213-253.

[Fla88] M. Flasiński, Parsing of edNLC-graph grammars for scene analysis, *Pattern Recognition* **21** (1988), 623-629.

[Fla89] M. Flasiński, Characteristics of edNLC - graph grammars for syntactic pattern recognition, *Comput. Vision Graphics Image Process.* **47** (1989), 1 - 21.

[Fla93] M. Flasiński, On the parsing of deterministic graph languages for syntactic pattern recognition, *Pattern Recognition* **26** (1993), 1 -16.

[Fla94] M. Flasiński, The programmed grammars and automata as tools for a construction of analytic expert systems, *submitted for publication.*

[FlaKot89] M. Flasiński, L. Kotulski, Constructing allocators for distributed environment with the help of graph rewriting systems, *Proc. 5th Int. Conf. Perform. Evaluat. Reliab. Exploit. Comp. Syst.*, Książ Castle, Poland, 26-29 Sept. 1989, 115-120, Ossolineum 1989.

[FlaKot92] M. Flasiński, L. Kotulski, On the use of graph grammars for the control of a distributed software allocation. *Computer Journ.* **35** (1992), A165 - 175.

[FlaLew91] M. Flasiński, G. Lewicki, The convergent method of constructing polynomial discriminant functions for pattern recognition, *Pattern Recognition* **24** (1991), 1009 - 1015.

[Fu82] K.S. Fu, *Syntactic Pattern Recognition and Applications*, Prentice Hall, Englewood Cliffs, 1982.

[Har78] M.A. Harrison, *Introduction to Formal Language Theory*, Reading, Massachusetts, Adison-Wesley, 1978.

[Jac90] P. Jackson, *Introduction to Expert Systems*, Addison-Wesley, Reading, MA, 1990.

[Janetal82] D. Janssens, G. Rozenberg, R. Verraedt, On sequential and parallel node-rewriting graph grammars, *Comput. Graphics Image Process.* **18** (1982), 279-304.

[JohHay86] M.V. Johnson, B. Hayes-Roth, Integrating diverse reasoning methods in the BB1 blackboard control architecture, *Techn. Report* KSL 86-76, Knowledge Systems Laboratory, Stanford University, 1986.

[Min75] M. Minsky, A framework for representing knowledge, in *Psychology of Computer Vision*, P.H. Winston (ed.), MIT Press, Cambridge, MA, 1975.

[Neb88] D. Nebendahl, *Expert Systems. Introduction to the Technology and Applications*, Wiley, Chichester, 1988.

[Pos43] E.L. Post, Formal reductions of the general combinatorial decision problem, *Amer. J. Math.* **65**, 197-268.

[Tal92] Talarian Corporation, Mountain View, USA. *RTworks v. 2.1 User Manual*, December 1992.

A Basic Notions Concerning Rule-Based Expert Systems

A good introduction to expert systems can be found in [Neb88]. For further reading, P. Jackson's monography [Jac90] is recommended. For studying rule-based expert systems and rule-based programming techniques, [Broetal86] is suggested. Here only very basic notions, which make understanding of the paper easier, are introduced.

expert system

A computer program that is able to represent and reason over knowledge delivered by a human expert for a restricted area of problems. While reasoning, an expert system tries to simulate a problem solving behaviour of a human expert. The reasoning is made by logical inferencing.

real-time expert system

An expert system, for which the correctness depends not only on the correctness of logical inferencing but also on the time at which the results are produced. Typically such a system consists of: a real-time data acquisition interface, a (graphics) user interface, and a $\xrightarrow{see}$ **knowledge base**.

knowledge base

Abstract representation of an expert system knowledge base is a combination of **entity-level knowledge (solution space)** and **problem-solving knowledge**. The objects (of the form of data structures) in the solution space are: input data, partial solutions, final solutions and control data. The problem-solving knowledge is a set of interpretive procedures, which are used for reasoning over data in the solution space.

rule-based expert system, production system

An expert system, in which interpretive procedures ($\xrightarrow{see}$ **knowledge base**) are sets of *condition-action* rules (productions).

B Selected Items of the Object-Oriented Booch Method

This appendix contains a short (and sometimes a simplified version of) description only of these basic notions and standard notations of Booch Object-Oriented approach, which have been used in the paper. For the introduction to this approach and full description of the method see [Boo91].

B.1 Basic definitions and notations

Object and *class* are basic notions of any OO approach.

object (instance)

A conceptual entity representing an individual (real-world or abstract) item with a well-defined role in the problem domain. An icon for an object is shown in Fig. 20a.

class

A set of objects that have a common structure and a common behaviour. An icon for a Class is shown in Fig. 20 b.

A complex system in the OO approach is described with the help of: objects, classes, and relationships among them.

containing (aggregation) relationship among objects

A relation, in which some objects are part of another object. In Fig. 20 c, objects: b,c,d are part of an object a.

using relationship among objects

A complex system is treated in the OO aproach as a structure consisting of objects, which are acted upon, and themselves act upon other objects to perform a given task. The actions that one object performs upon another to elicit a required reaction are called **messages**. If two objects are in a **using relationship**, then they may exchange messages unidirectionally (one of them can "use" another) or bidirectionally (they can "use" one another). In Fig. 20 d, objects: a and b are in a using relationship; in Fig. 20 e, an object c uses an object d (c sends messages m1 to d); in Fig. 20 f, objects e and f use one another (e sends messages m1 to f, and f sends messages m2 to e).

inheritance relationship among classes

A relation, in which one class inherits features (the structure or behaviour), which are defined in another class (**single inheritance**), or defined in more than one classes (**multiple inheritance**). In Fig. 20 g, the class B inherits features from the class A (single inheritance); in Fig. 20 h, the class E inherits features from the classes C and D (multiple inheritance).

using relationship among classes

A relation, in which one class uses the resources of another class (their instances are in $\xrightarrow{\text{see}}$ **using relationship among objects**). A class A can use a class B in two different ways. Firstly, an implementation of A can use resources of B (in this case B is hidden as part of the secret of A), which is denoted as in Fig. 21 a.

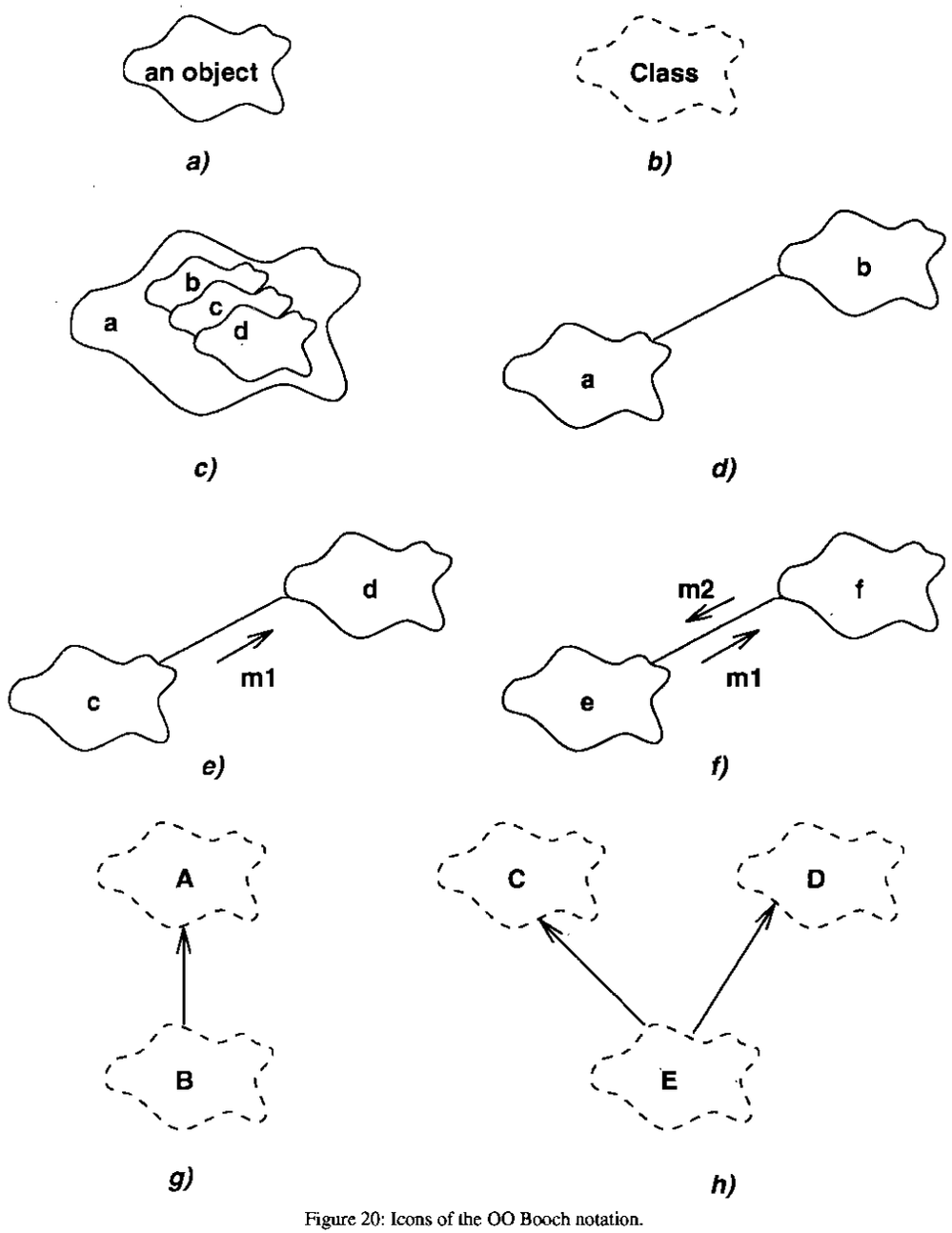

Figure 20: Icons of the OO Booch notation.

Secondly, an interface of A can use resources of B (in this case B is visible to other clients), which is denoted as in Fig. 21 b.
In general, m instances of the class A can use n instances of the class B, which is denoted by puting "m" at the A-end of a class using relationship symbol and by puting "n" at the B-end of the symbol. In Fig. 21 c, one instance of the class A uses one instance of the class B; in Fig. 21 d, one instance of the class A uses n instances of the class B.

instantiation relationship among classes

It is convenient, sometimes, to define a **parametrized (generic) class**, which can serve as a template for other classes. This template can be then parametrized by other classes, objects, or operations. For example, the Set class (with the set of its typical operations, e.g. union, intersection, difference, is-empty, etc.) can be defined as a pattern (container) class for classes AppleSet, OrangeSet, LemonSet, etc. Before creating the corresponding objects a parametrized class has to be instantiated (i.e. its parameters have to be filled in). Sets, queues, rings, graphs, trees and stacks are typical examples of pattern (container) classes. In Fig. 21 e, the class D is a pattern/container class for the class C

With the help of relationships we can build hierarchical/structural representations of complex systems. *Object structure* and *class structure* are the most important structures used in the OO approach.

object structure

A structure defined with $\xrightarrow{see}$ **containing relationships among objects.** In Fig. 21 f, an object a is built up of three (sub)objects: b, c, d, and a (sub)object b consists, in turn, of (sub)objects: f, e.

class structure

A structure defined with $\xrightarrow{see}$ **inheritance relationships among classes.** In Fig. 21 g, a class D inherits from classes: G, B (multiple inheritance), and a class B inherits in turn from a class A (single inheritance).

B.2 Basic OO principles

Object-Oriented approach is based on the following four basic principle.

abstraction

A generalized model of a complex phenomenon, system, etc, in which inessential (from the point of view of a given observer) details are ignored.

encapsulation

A process of hiding all of the details of an object behaviour implementation that do not contribute to object essential characteristics.

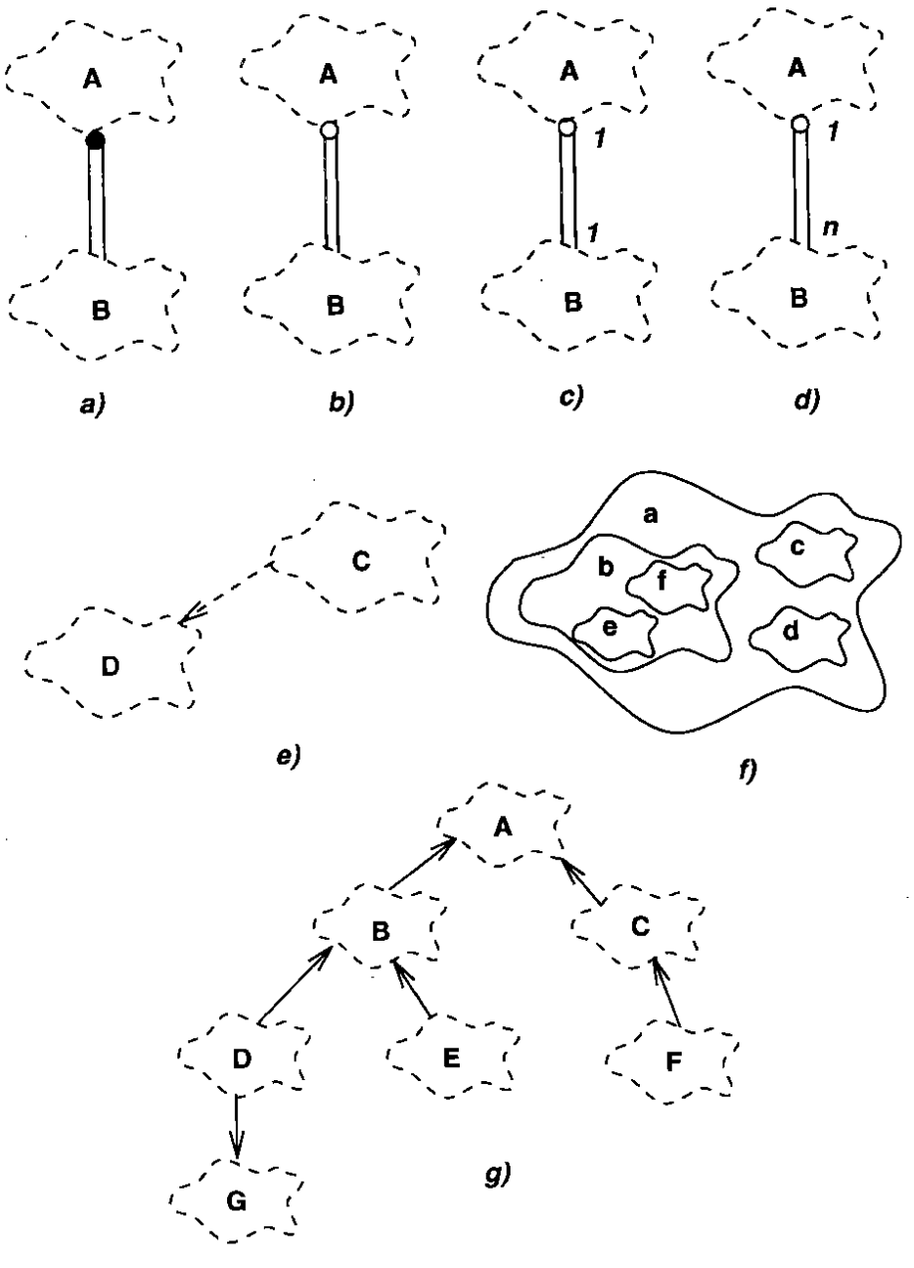

Figure 21: Icons of the OO Booch notation.

modularity

Decomposing a complex system into a set of cohesive and loosely coupled modules in order to reduce its complexity.

hierarchy

Representing a complex system with a set of hierarchical structures (hierarchies) by ranking/ordering of abstractions. For a complex system, two most important hierarchies are: its $\xrightarrow{\text{see}}$ **object structure** and its $\xrightarrow{\text{see}}$ **class structure**.

B.3 Basic diagrams of the Booch OO approach

Two basic types of diagrams are used in the Booch OO approach.

object diagram

A diagram, which serves both to depict object decomposition of a system and to show a system behaviour in terms of the inter-object behaviour. It is used to show: objects, relationships among them, and message passing between them.

class diagram

A diagram, which is used for the logical design of a system. It shows classes and their relationships.

C Basic Notions of the Theory of Formal Languages and Automata

In 1943 Post [Pos43] studying the general combinatorial decision problem, introduced the so-called *canonical systems*, which were then modified for linguistics purposes and defined as *formal grammars* in the Chomsky's model [Cho57]. We present here, introductory notions of this model. For further reading, Harrison's book [Har78] is recommended.

Let Δ be an **alphabet**. The set of all words (including the empty word) over Δ, denoted by Δ^*, is called a **language** over Δ. For example, if $\Delta = \{a, b, c\}$, then the following are the words over Δ

$$a, ab, abc, aaabb, bbbaa, cbacbacba$$

If there are two alphabets: Δ_1 and Δ_2, then by $\Delta_1^*\Delta_2^*$ we denote a language consisting of all the words xy, which are created by a concatenation of words belonging to Δ_2 to words of Δ_1 (i.e. $x \in \Delta_1^*$, $y \in \Delta_2^*$).
By λ we denote an empty word.

A **formal grammar** is a quadruple

$$G = (\Sigma, \Delta, P, S)$$

where Σ is a finite, nonempty set of nonterminal (auxiliary) symbols, called a *nonterminal alphabet*,
Δ is a finite nonempty set of terminal symbols, called a *terminal alphabet*,
P is a finite set of productions (rules) of the form:

$$L \longrightarrow R, \text{ where } L \in (\Sigma \cup \Delta)^*\Sigma(\Sigma \cup \Delta)^* \text{ , } R \in (\Sigma \cup \Delta)^*,$$

$S \in \Sigma$ is the initial symbol, called *axiom*.
L and R are called left- and right- sides (respectively) of a production. A production is sometimes denoted as a pair (L, R).

A language generated by a grammar G, denoted $L(G)$, is a set of all the words over Δ, which can be derived by application of the grammar productions starting from the axiom.

For example, if $\Delta = \{a, b, c\}$, $\Sigma = \{S\}$, and we define the following simple set of rules:

1. $S \longrightarrow aSb$,
2. $S \longrightarrow c$,

then we can generate the words of the form a^ncb^n, where a^n denotes n-ary repetition of a, e.g.:

$$S \longrightarrow aSb \longrightarrow aaSbb \longrightarrow aaaSbbb \longrightarrow aaacbbb.$$

A **finite state automaton with output** is a six-tuple

$$A = (Q, \Delta_1, \Delta_2, \delta, q_0, F)$$

where Q is a finite set of states,
Δ_1 is a finite nonempty set of input symbols, called *input alphabet*,
Δ_2 is a finite nonempty set of output symbols, called *output alphabet*,
$\delta : Q \times \Delta_1 \longrightarrow Q \times \Delta_2$ is the transition function,
$q_0 \in Q$ is the initial state,
$F \subset Q$ is the set of final states.

The transition function δ describes how the automaton changes from one state to another. Let us assume that for some input symbol a and some state q: $\delta(q, a) = (q', x)$. It means that, if the automaton is in state q and it receives the input a, then the automaton goes to the state q' and writes a symbol x to its output. The automaton starts its work in the state q_0, and it stops after reaching a state from the set F.